# Optimal and Learning Control for Robotics
# Exercise Series 3

Bo Peng, New York University

Spring 2020

## Exercise 1

**Solution** a) The python code to run MPC at every time step is shown here:

```
    def MPCstep(walking_model, xt, plan_horizon, Q_nominal, R_nominal, foot_position):
        """
        Replan at time step t with state xt.

        Args:
            walking_model: the enironment;
            xt: the initial state at time t;
            plan_horizon: the replanning horizon length;
            Q_nominal: nominal Q matrix;
            R_nominal: nominal R matrix;
            foot_position: a list of foot posistions;

        Returns:
            ut: the control output at time step t
        """

        ## initialize matrices
        G_bounds = []
        h_bounds = []
        Q = []
        q = []
        R = []
        r = []

        for i in range(plan_horizon):
            Q.append(Q_nominal)
            R.append(R_nominal)

            q.append(Q_nominal.dot(np.array([[-foot_position[i]],[0.]])))
            r.append(R_nominal.dot(np.array([-foot_position[i]])))

            G_bounds.append(np.array([[0,0,1],[0,0,-1]]))
            h_bounds.append(
                np.array([[walking_model.foot_size+foot_position[i]],
                          [walking_model.foot_size-foot_position[i]]]))

        ## solve mpc with given codes
        x_plan, u_plan = solve_mpc_collocation(
                             walking_model.A,
                             walking_model.B,
```

```
                            Q,
                            q,
                            R,
                            r,
                            G_bounds,
                            h_bounds,
                            plan_horizon,
                            xt)

        ## take the first step
        ut = u_plan[:, 0]

        return ut


    def MPController(walking_model, horizon_length,
                     plan_horizon, Q_nominal, R_nominal, foot_position):
        """
        A model predictive controller for stochastic walking environment.

        Args:
            walking_model: the environment;
            horizon_length: the total length of horizon;
            plan_horizon: the replanning horizon length;
            Q_nominal: nominal Q matrix;
            R_nominal: nominal R matrix;
            foot_position: a list of foot positions;

        Returns:
            controller: a controller function for MPC.
        """

        def controller(x, i):
            horizon = min(plan_horizon, horizon_length-i)
            foot_steps = foot_position[i:i+horizon]

            u = MPCstep(walking_model, x, horizon, Q_nominal, R_nominal, foot_steps)
            return u

        return controller

```

Listing 1: MPC Algorithm

The MPController function returns a MPC controller that runs MPCstep function at each time step, which solve the original OC problem for a shorter horizon (plan_horizon) and apply only the first control output $u_t$.

b) To find the shortest time horizon I run an experiment that use different plan_horizon to solve the OC problem. In each experiment, the same plan_horizon is used for 200 times since we are using a noisy environment.

To check if the controller stabilizes the system, I define a metric "cost", which is the distance between original plan states and real states:

$$\text{cost} = \sqrt{(x_{\text{real}} - x_{\text{plan}})^T (x_{\text{real}} - x_{\text{plan}})} \tag{1}$$

The idea is that if the controller could stabilize the system in a noisy environment, then the system should behave as if it is in a non-noise environment, which is the planned state. Therefore, if the cost is low, then the controller stabilize the system in the noisy environment.

The experiment results are shown below (note that the experiment results may change because of the noise in the environment):

Table 1: Plan Horizon Experiment Results

| Plan Horizon | Mean Cost | Minimum Cost | Maximum Cost |
|---|---|---|---|
| 1 | 2.059134e+09 | 2.154437e+09 | 1.953782e+09 |
| 2 | 1.965600e+09 | 2.074292e+09 | 1.872367e+09 |
| 3 | 1.865734e+09 | 1.953721e+09 | 1.780687e+09 |
| 4 | 1.714734e+09 | 1.778597e+09 | 1.630459e+09 |
| 5 | 1.509684e+09 | 1.581568e+09 | 1.420585e+09 |
| 6 | 1.241064e+09 | 1.288699e+09 | 1.167756e+09 |
| 7 | 9.014475e+08 | 9.564713e+08 | 8.431085e+08 |
| 8 | 5.677452e+08 | 6.328668e+08 | 5.134207e+08 |
| 9 | 3.008015e+08 | 3.526146e+08 | 2.464213e+08 |
| 10 | 9.021696e+07 | 1.335314e+08 | 3.131681e+07 |
| 11 | 1.306313e+05 | 4.278392e+06 | 9.459968e-01 |
| 12 | 1.954138e+05 | 1.954131e+07 | 6.288541e-01 |
| 13 | 4.634063e+05 | 2.165111e+07 | 4.381214e-01 |
| 14 | 3.763186e-01 | 6.592180e-01 | 3.154441e-01 |
| 15 | 2.979701e-01 | 4.668112e-01 | 2.368805e-01 |
| 16 | 2.418781e+05 | 1.303302e+07 | 1.844776e-01 |
| 17 | 1.759316e+04 | 1.759294e+06 | 1.379911e-01 |
| 18 | 6.661043e+04 | 6.661023e+06 | 1.092199e-01 |
| 19 | 1.851220e-01 | 5.788336e-01 | 9.333695e-02 |
| 20 | 1.702376e-01 | 3.820062e-01 | 9.580803e-02 |

If a minimum cost is closer to 0, it means that the controller could stabilize the system in at least 1 trails for the used plan horizon. So the shortest time horizon is 11. Note that it is very likely to fail if we use this plan horizon directly. A failure example is shown below, and the animation can be found here (the planned curve is the original planned one in the non-noise environment):

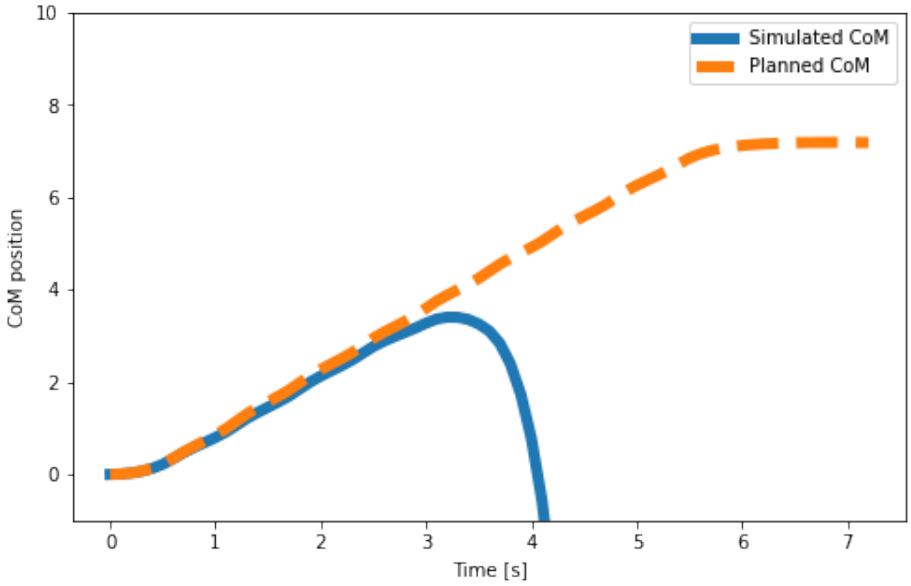


Figure 1: CoM Position (horizon=11)

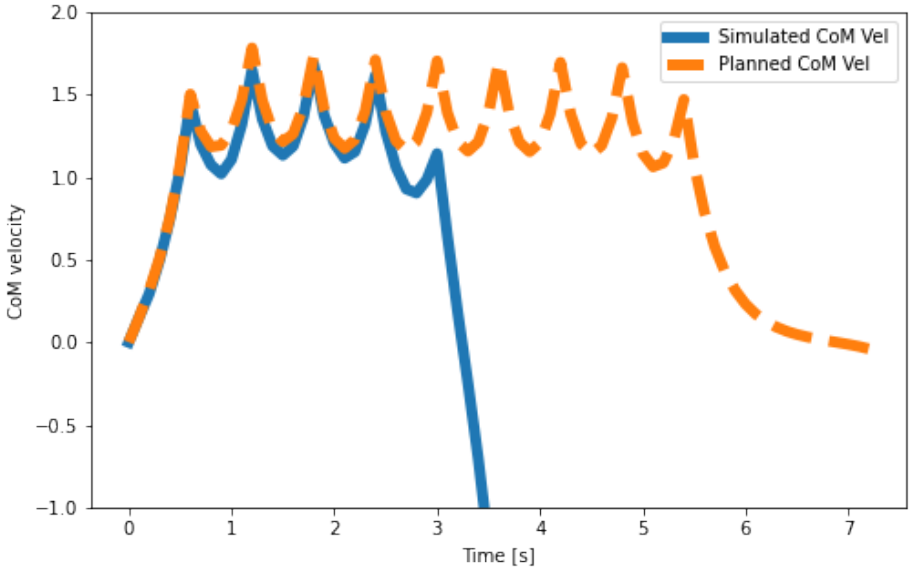


Figure 2: CoM Velocity (horizon=11)

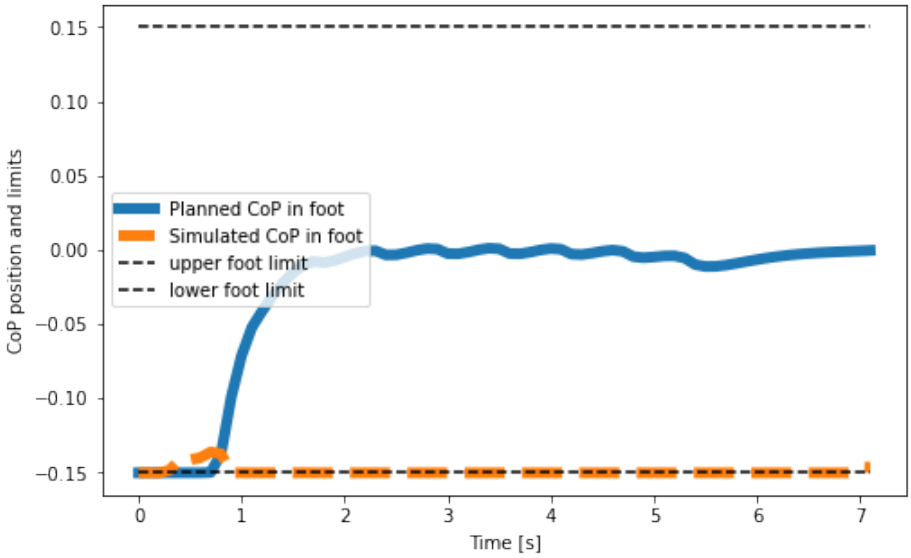


Figure 3: CoP Position and Limits (horizon=11)

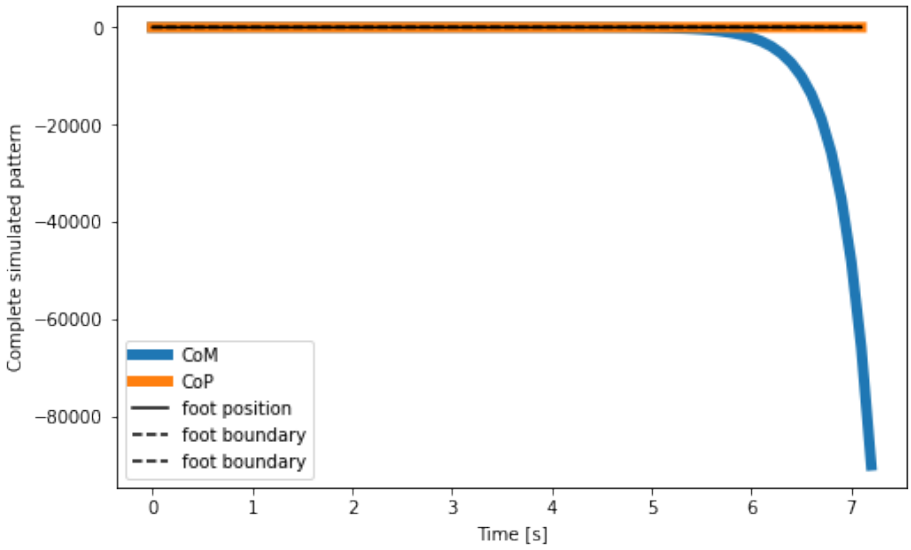


Figure 4: Complete Simulated Pattern (horizon=11)

A successful example is shown below, and the animation can be found here.

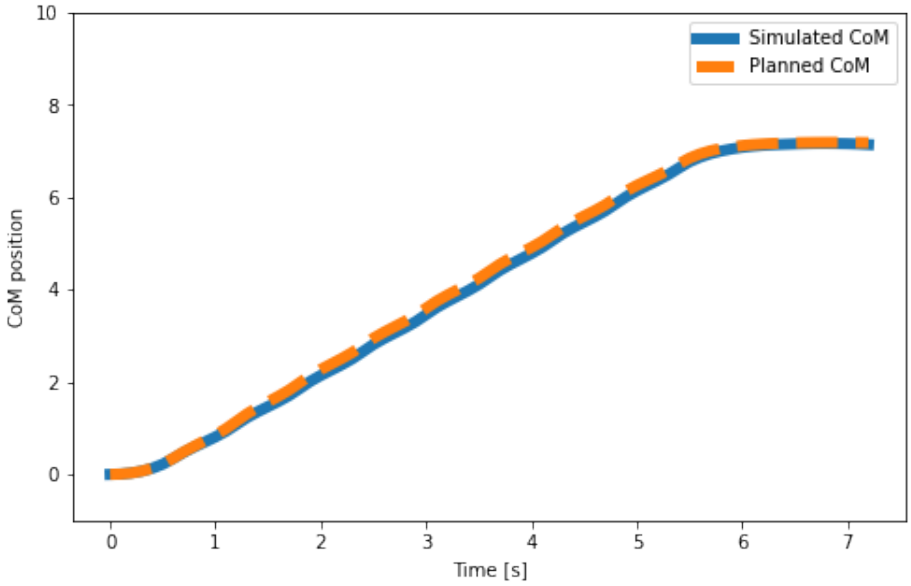


Figure 5: CoM Position (horizon=11)

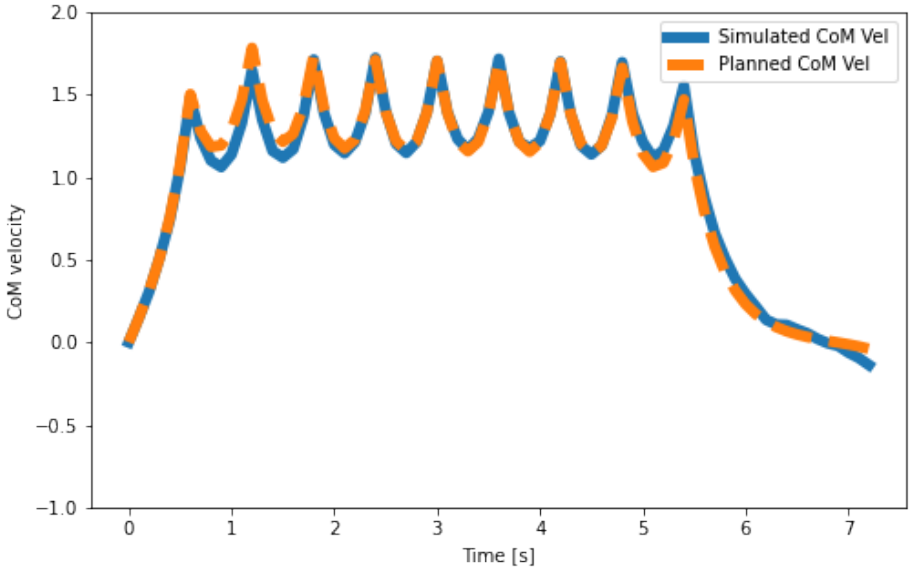


Figure 6: CoM Velocity (horizon=11)

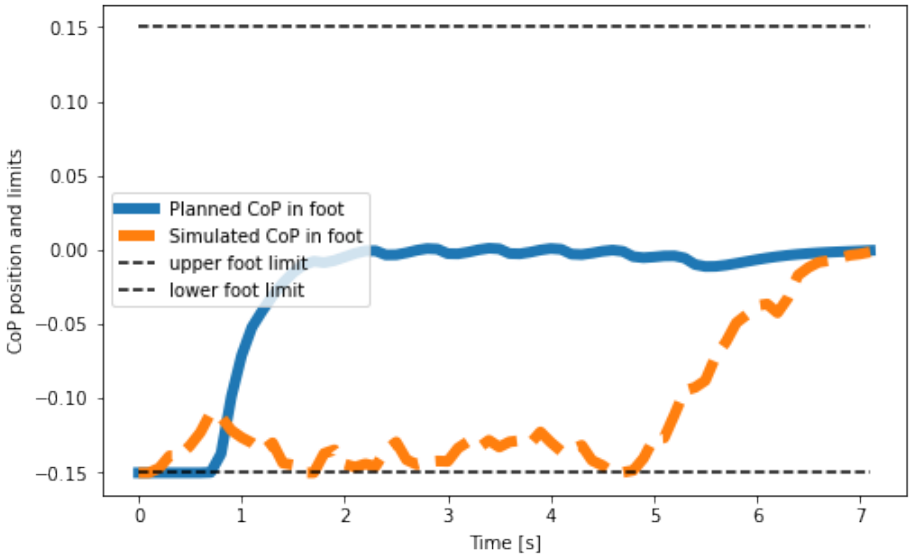


Figure 7: CoP Position and Limits (horizon=11)

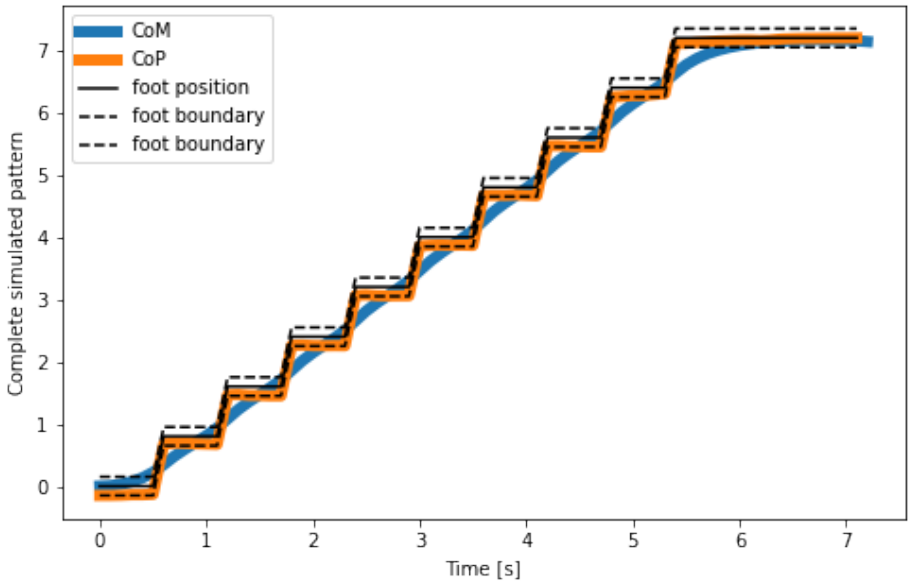


Figure 8: Complete Simulated Pattern (horizon=11)

c) If a mean cost is closer to 0, it means that the controller could stabilize the system in most of the trails. So a stable controller should plan at least 14 steps in advance. An example is shown below, and the

animation can be found here.

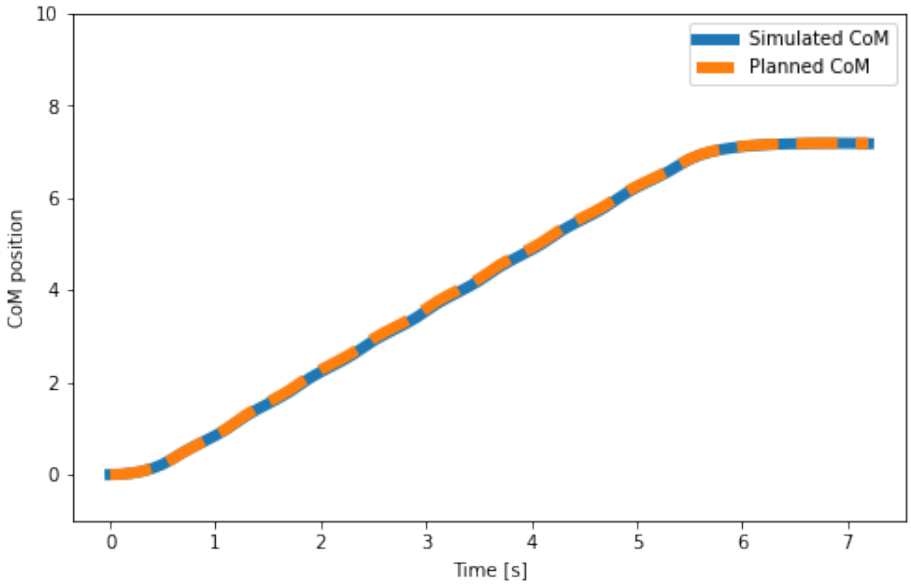


Figure 9: CoM Position (horizon=14)

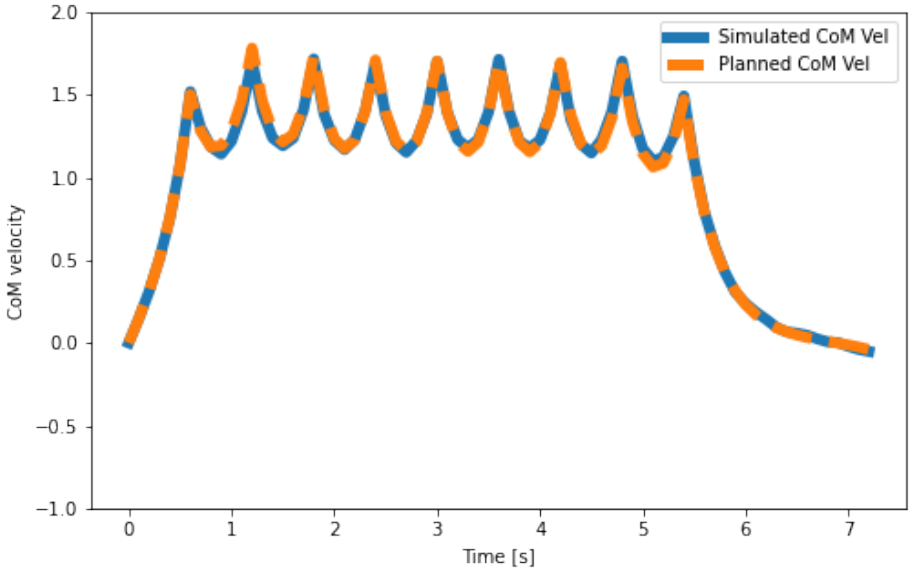


Figure 10: CoM Velocity (horizon=14)

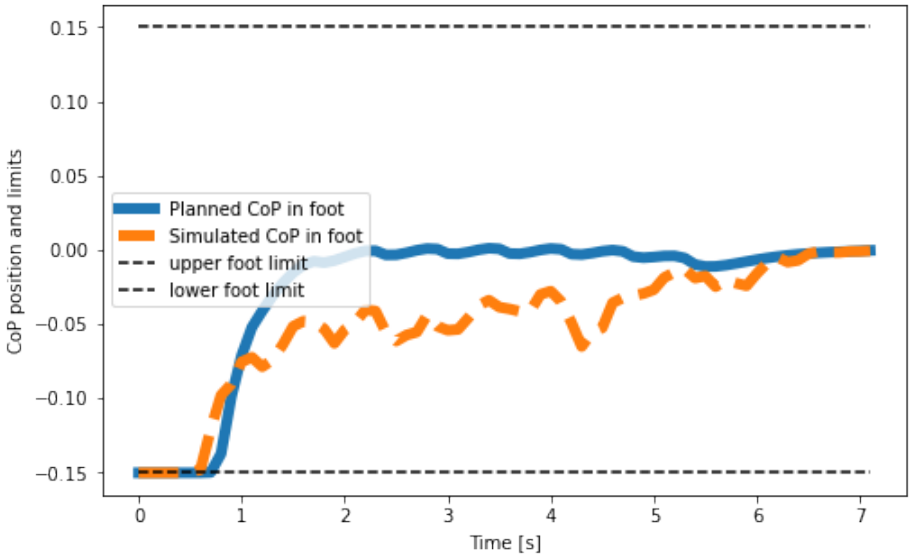


Figure 11: CoP Position and Limits (horizon=14)

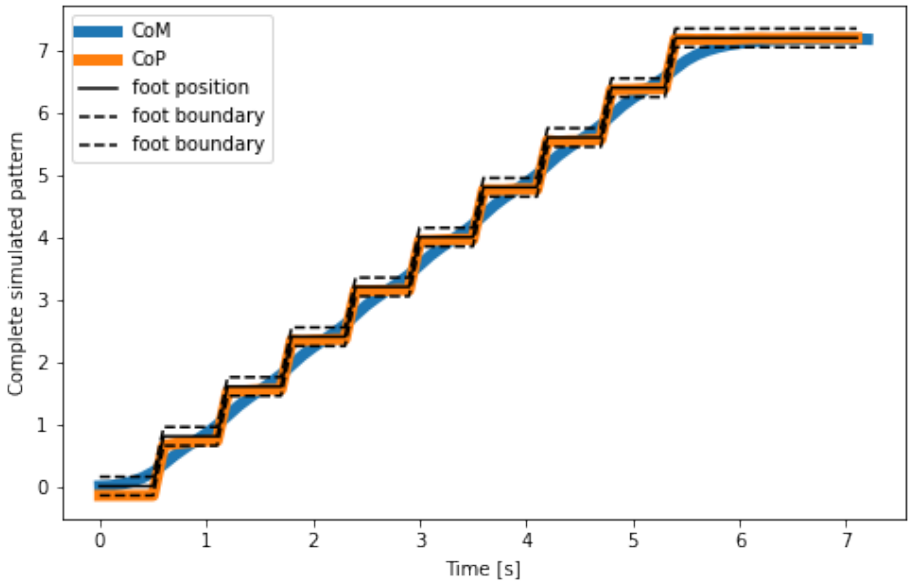


Figure 12: Complete Simulated Pattern (horizon=14)

d) The comparison between the original planned control and executed control is shown below. In general, the executed control is a little bit lower than the planned control sequence to adjust to the environment

noise. The longer plan horizon, the smaller difference there will be.

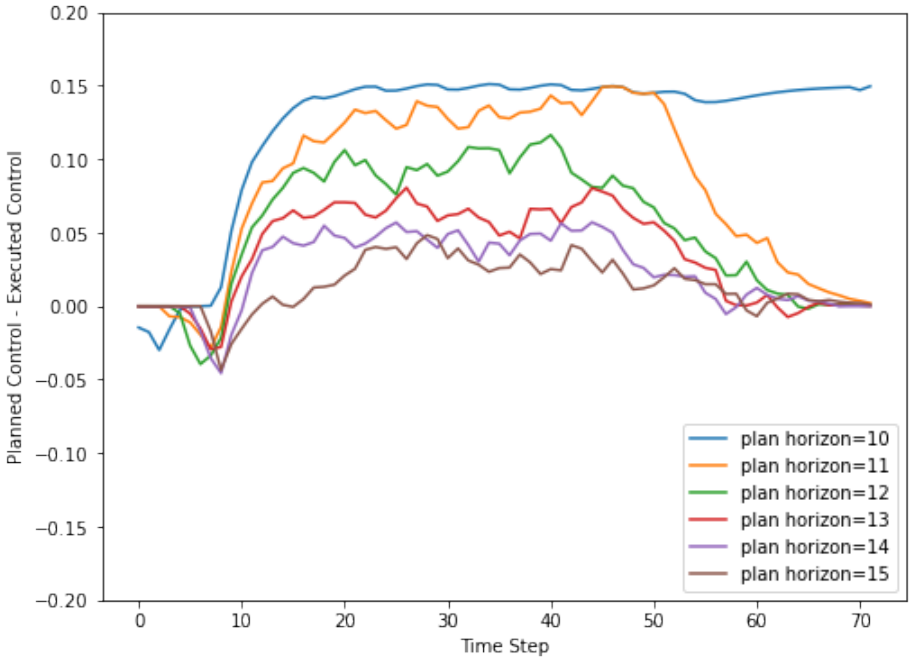


Figure 13: Comparison between Control Sequence

e) When I increase the cost weight in cost function, there's no big difference in shortest plan horizon or stable plan horizon. Besides, if I multiply the cost weight by 1,000, the controller fails to guide the robot to walk. I think maybe that's because when we increase the cost weight the controller will force the robot to follow the trajectory, which maybe very hard in the noisy environment. If the cost weight is too high, it could be impossible to follow the trajectory and lead to failure.

Similarly, if decrease the cost weight there's no big difference either. If I divide the cost weight by 1,000, the controller also fails to guide the robot. I think that's because when we decrease the weight, the constraints on the target will become less important. So the robot won't need to follow the trajectory to get low cost. In conclusion, only change the cost weight may not decrease the plan horizon, and may lead to failure in extreme cases.

When I increase the terminal cost, the shortest control horizon decrease to 7. The simulation results are shown below, and the animation can be found here. It seems that if we increase terminal cost, the controller would force the robot to stay at the terminal, which is helpful for planning.

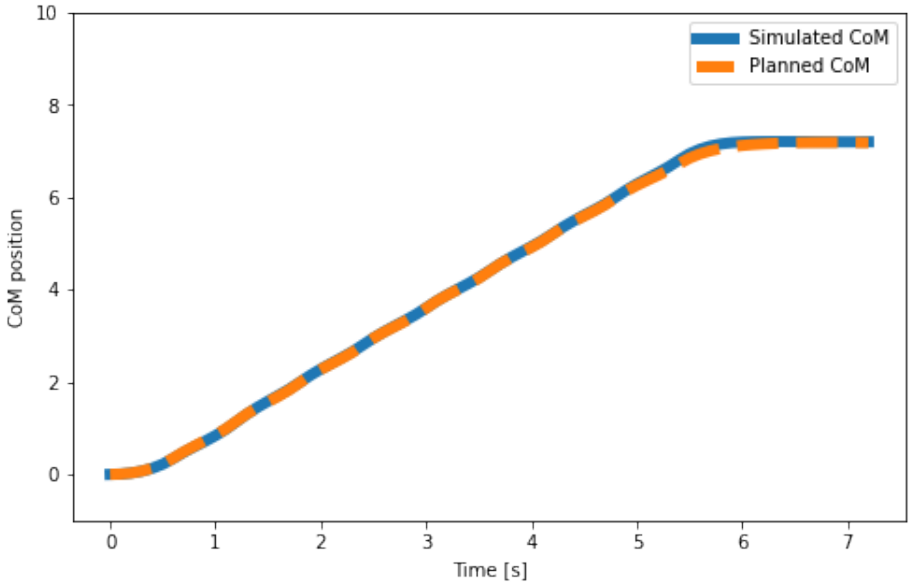


Figure 14: CoM Position (horizon=7)

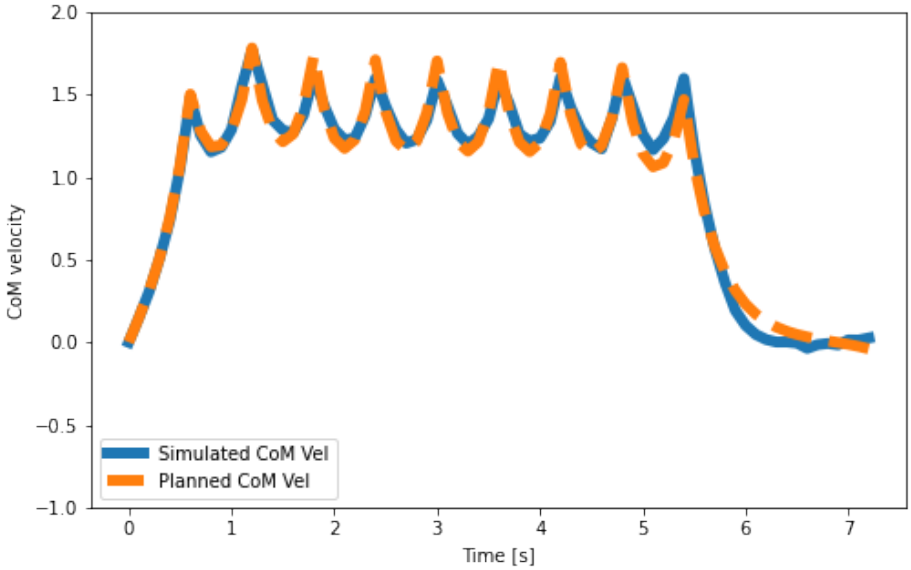


Figure 15: CoM Velocity (horizon=7)

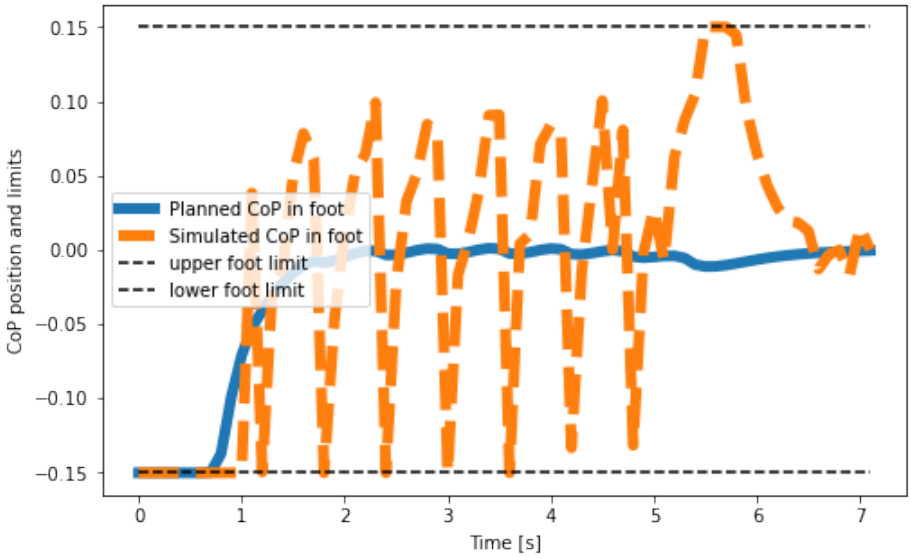


Figure 16: CoP Position and Limits (horizon=7)

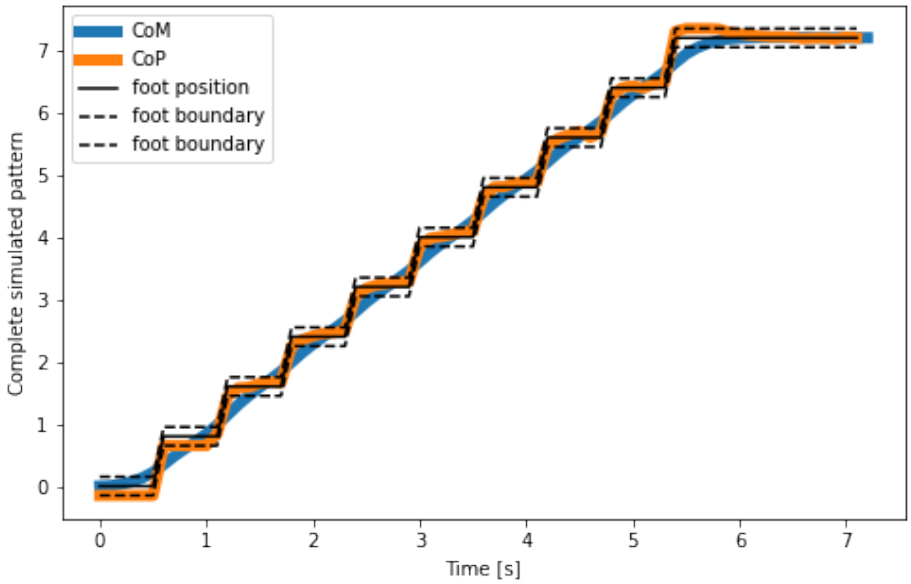


Figure 17: Complete Simulated Pattern (horizon=7)

## Exercise 2

**Solution** a) To run the value iteration algorithm, first we need to code the environment. The environment is shown below:

```
    def env(x, u):
        """
        The grid-world environemnt.

        Args:
            x: the current state;
            u: the current action.

        Returns:
            x_next: the next state;
            cost: the current cost.
        """

        def get_coord(x):
            """
            A helper function to transfer state x to coordinate in grid-world.

            Args:
                x: the current.

            Returns:
                i, j: the coordinate in grid-world.
            """

            return x//5, x%5

        i, j = get_coord(x)
        cost = 0

        if (i==0 and j==4) or (i==3 and j==4):
            cost = -1
        elif (i==0 and j==2) or (i==1 and j==2) or (i==2and j==3) or (i==3 and j==0):
            cost = 1
        elif (i==1 and j==4) or (i==2 and j==1) or (i==3 and j==3):
            cost = 10

        di, dj = 0, 0
        if u == 0:
            di = -1
        elif u == 1:
            di = 1
        elif u == 2:
            dj = -1
        elif u == 3:
            dj = 1

        ## obstacles
        if (i+di==1 and j+dj==1) or (i+di==2 and j+dj==2):
            di, dj = 0, 0

        ## make a move
        i += di
        j += dj

        ## boundry condition
```

```
        if i < 0:
            i = 0
        elif i > 3:
            i = 3

        if j < 0:
            j = 0
        elif j > 4:
            j = 4

        x_next = int(i * 5 + j)

        return x_next, cost

```

Listing 2: Environment Dynamics

The value iteration algorithm is shown below:

```
    def ValueIteration(env, states, actions, alpha, tol=1e-6):
        """
        Value iteration algorithm.

        Args:
            env: the environment function;
            states: the state indices;
            actions: the action indices;
            alpha: the discount factor;
            tol: converge tolerance.

        Returns:
            v: the value function for each state.
        """

        v = np.zeros_like(states, dtype=np.float)
        i = 0
        while True:
            v_new = np.zeros_like(v)
            for s in states:
                q = np.zeros_like(actions, dtype=np.float)
                for a in actions:
                    s_next, cost = env(s, a)
                    q[a] = cost + alpha * v[s_next]

            v_new[s] = np.min(q)

            if np.all(np.abs(v_new-v) < tol):
                break
            v = v_new
            i += 1

        print(f"Converge in {i} iterations.")

        return v

```

Listing 3: Value Iteration Algorithm

In this grid-world case, the algorithm converge in 1,375 iterations. The optimal value function and optimal policy are shown below:

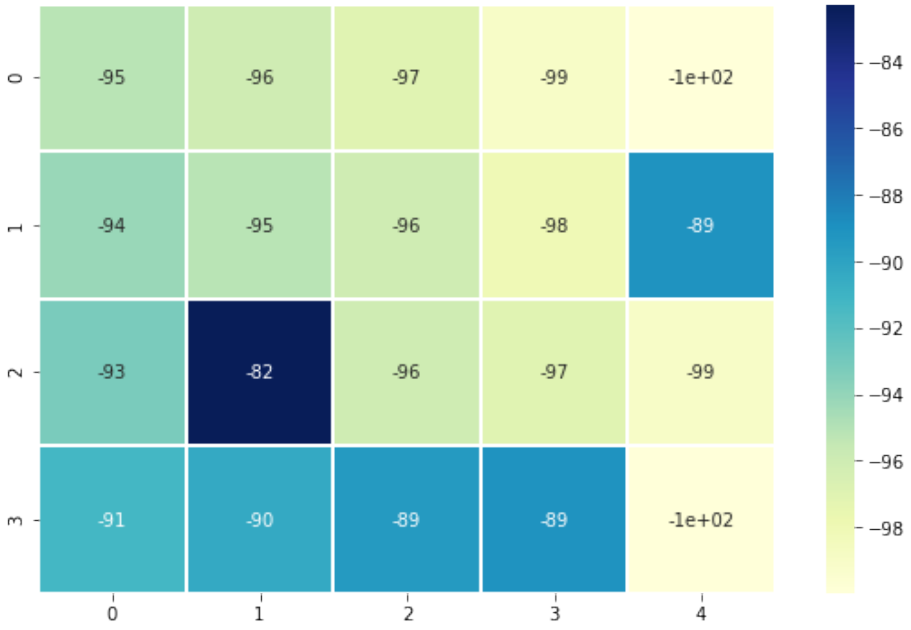


Figure 18: Optimal Value Function (Value Iteration)

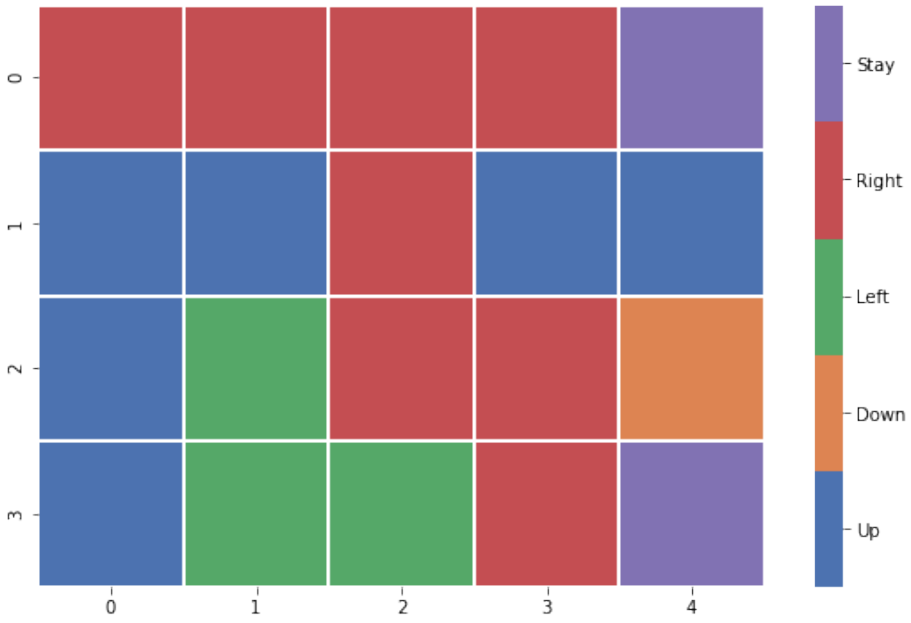


Figure 19: Optimal Policy (Value Iteration)

2) The policy iteration algorithm is shown below:

```
def PolicyIteration(env, states, actions, policy, alpha):
    """
    Policy iteration algorithm.

    Args:
        env: the environment function;
        states: the state indices;
        actions: the action indices;
        policy: the initial policy;
        alpha: the discount factor;
        tol: converge tolerance.

    Returns:
        v: the value function for each state.
    """

    def PolicyEvaluate(policy):
        """
        A helper function to evaluate the given policy.
        """

        N = len(states)
        T = np.zeros((N, N))
        g = np.zeros_like(states)

        for s in states:
            a = policy[s]
            s_next, g[s] = env(s, a)
            T[s, s_next] = 1

        J = np.linalg.inv(np.eye(N) - alpha * T) @ g

        return J

    def PolicyUpdate(policy, J):
        """
        A helper function to update current policy.
        """

        new_policy = np.zeros_like(policy)
        for s in states:
            q = np.zeros_like(actions, dtype=np.float)
            for a in actions:
                s_next, cost = env(s, a)
                q[a] = cost + alpha * J[s_next]

            new_policy[s] = np.argmin(q)

        return new_policy


    i = 0
    while True:
        ## policy evaluation
        J = PolicyEvaluate(policy)
        ## policy update
        new_policy = PolicyUpdate(policy, J)

        if np.all(policy == new_policy):
```

```
                break

            policy = new_policy
            i += 1

        print(f"Converge in {i} iterations.")

        return policy, J

```

Listing 4: Policy Iteration

In this grid-world case, the algorithm converge in 8 iterations. The optimal value function and optimal policy are shown below:

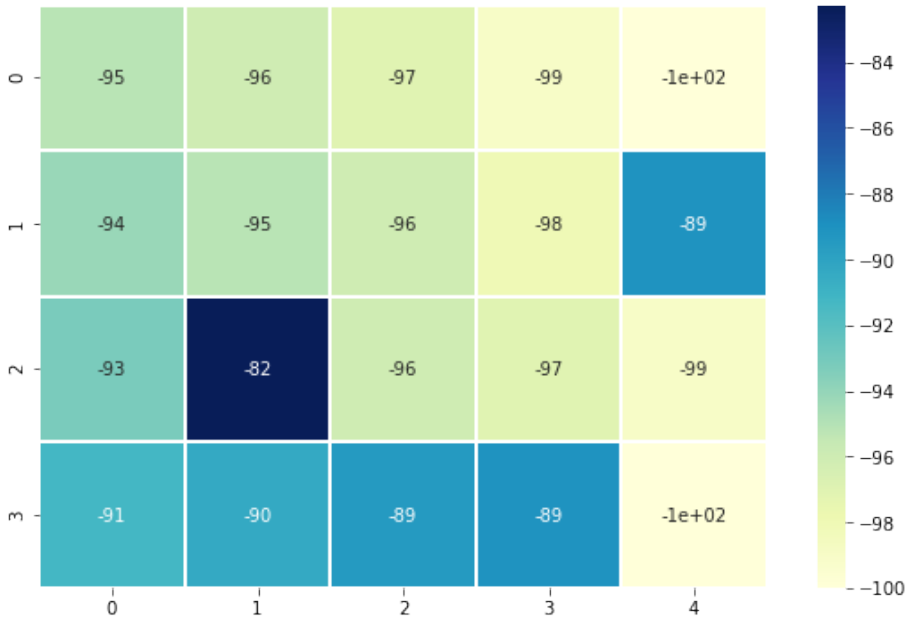


Figure 20: Optimal Value Function (Policy Iteration)

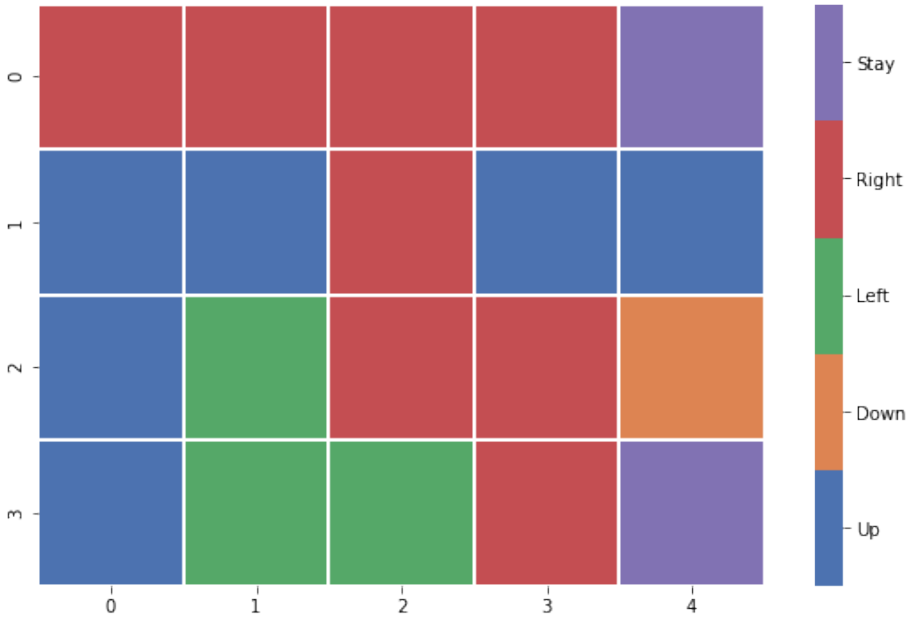


Figure 21: Optimal Policy (Policy Iteration)

c) For the solution, both value iteration and policy iteration algorithm could find the optimal value function/ policy. The optimal value function/ policy are the same in this case, and they are already shown in the previous sections.

For the convergence/ complexity, both value iteration and policy iteration algorithm could converge, and the corresponding optimal value/ policy are the same. But the convergence speed is different. The value iteration algorithm may take a relative longer time to converge, in this case is 1,375, while policy iteration only needs 8 iterations. So policy iteration could accelerate the convergence. Note that we are using the linear equation when we evaluate the policy. If we use the iterative method, the policy iteration algorithm may need longer time to converge.