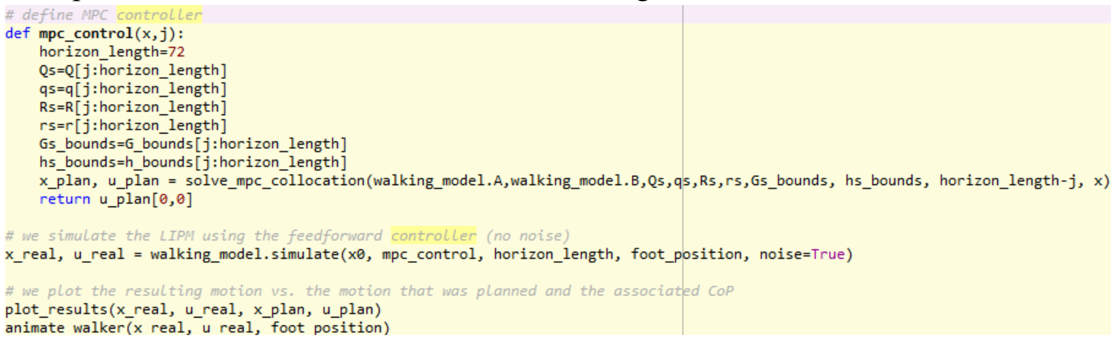
Exercise 1:

a) The example code only did the MPC controller at first state x_0. What we need to do is using the Q, q, R, r, G, h which have been given to do residual iterations and update control. The iteration code is following:



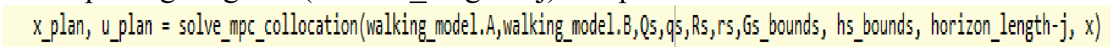
```
# define MPC controller
def mpc_control(x,j):
    horizon_length=72
    Qs=Q[j:horizon_length]
    qs=q[j:horizon_length]
    Rs=R[j:horizon_length]
    rs=r[j:horizon_length]
    Gs_bounds=G_bounds[j:horizon_length]
    hs_bounds=h_bounds[j:horizon_length]
    x_plan, u_plan = solve_mpc_collocation(walking_model.A,walking_model.B,Qs,qs,Rs,rs,Gs_bounds, hs_bounds, horizon_length-j, x)
    return u_plan[0,0]

# we simulate the LIPM using the feedforward controller (no noise)
x_real, u_real = walking_model.simulate(x0, mpc_control, horizon_length, foot_position, noise=True)

# we plot the resulting motion vs. the motion that was planned and the associated CoP
plot_results(x_real, u_real, x_plan, u_plan)
animate_walker(x_real, u_real, foot_position)
```

We defined a new controller to update control u at every iteration.

b) After getting an average output COP, the shortest time horizon is 12.

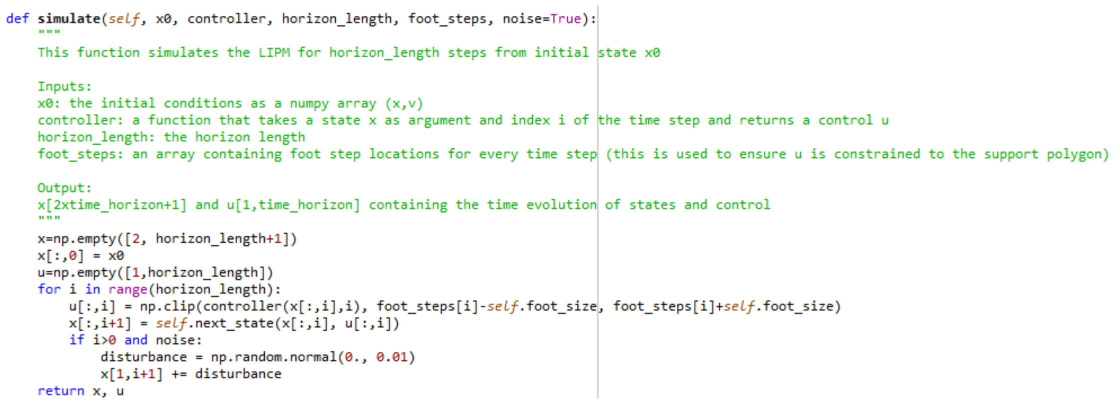
c) According to the codes which is showed, when we want to get the stable controller in current state, we need to predict all controls from current to the end. That is how solve_mpc_collocation function works when we calculate controller in every state.

The inputting length is (horizon_length – j). J represents current state.



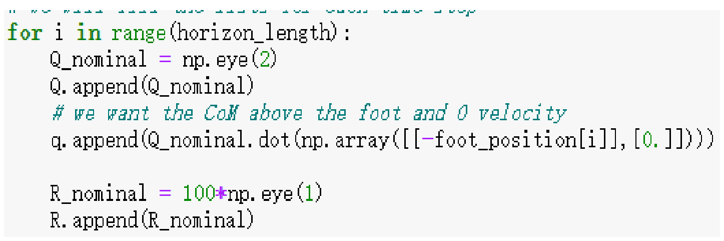
```
    x_plan, u_plan = solve_mpc_collocation(walking_model.A,walking_model.B,Qs,qs,Rs,rs,Gs_bounds, hs_bounds, horizon_length-j, x)
```

d) For original planned one, it only did once MPC calculation (solve_mpc_collocation function). But the new one did MPC calculation at every single state, even though it only returns the current control. But when we executed simulate function, it will update next state x with current state x and current new control which gets from MPC. Thus, in new one, the control u is changed at every state due to every state's MPC calculation. Compared to that, the original one only has a series of fixed control, due to once MPC calculation. If there is no noise or disturbing, the result of them are same. But if the noise is coming, only doing MPC calculation at first state is not enough to deal with the noise. The result would leave away from planned position.



```
def simulate(self, x0, controller, horizon_length, foot_steps, noise=True):
    """
    This function simulates the LIPM for horizon_length steps from initial state x0

    Inputs:
    x0: the initial conditions as a numpy array (x,v)
    controller: a function that takes a state x as argument and index i of the time step and returns a control u
    horizon_length: the horizon length
    foot_steps: an array containing foot step locations for every time step (this is used to ensure u is constrained to the support polygon)

    Output:
    x[2xtime_horizon+1] and u[1,time_horizon] containing the time evolution of states and control
    """
    x=np.empty([2, horizon_length+1])
    x[:,0] = x0
    u=np.empty([1,horizon_length])
    for i in range(horizon_length):
        u[:,i] = np.clip(controller(x[:,i],i), foot_steps[i]-self.foot_size, foot_steps[i]+self.foot_size)
        x[:,i+1] = self.next_state(x[:,i], u[:,i])
        if i>0 and noise:
            disturbance = np.random.normal(0., 0.01)
            x[1,i+1] += disturbance
    return x, u
```

e) If the weight of the cost function is changed, (Matrix Q changed) the length of the back-off level will increase. The heavier the weight, the longer the required back-off level, and the lighter the weight, the longer the required back-off level. However, if the weight exceeds, the back-off level will increase in length. Faster than weight loss.

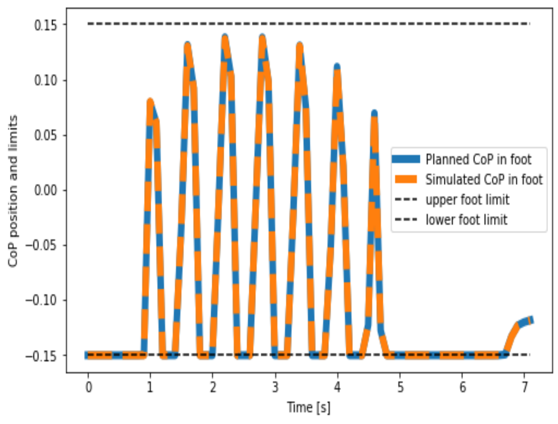


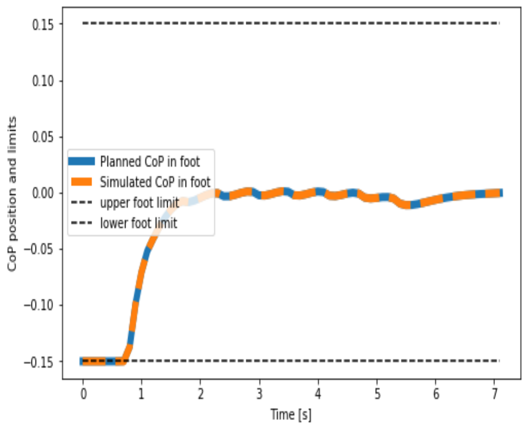
```
for i in range(horizon_length):
    Q_nominal = np.eye(2)
    Q.append(Q_nominal)
    # we want the CoM above the foot and 0 velocity
    q.append(Q_nominal.dot(np.array([[-foot_position[i]],[0.]])))

    R_nominal = 100*np.eye(1)
    R.append(R_nominal)
```

Changing the scaler of Q and R, these two matrices would change

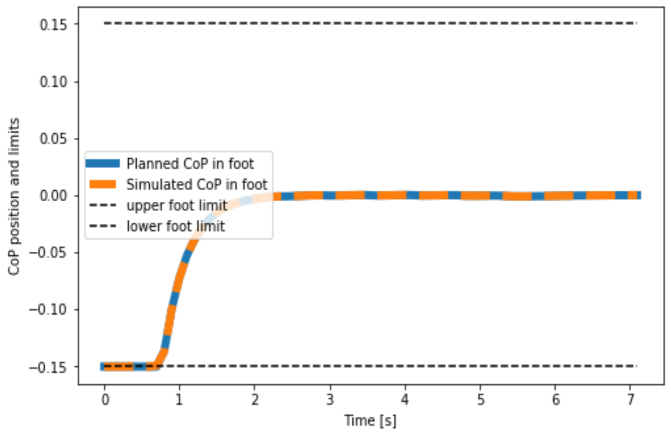
Whatever we don't consider about noise, as the scaler of R increasing, nothing will have a big change. As the scaler of Q increasing, the horizon length will decrease. Because COP is the output of the system, and when Q changing, the COP got 7 times fluctuation before 5s. Which means horizon length has decreased.

Without noise:



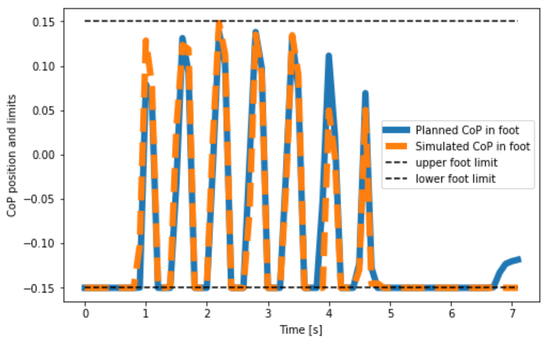


Q became to 10^9 Q has no change

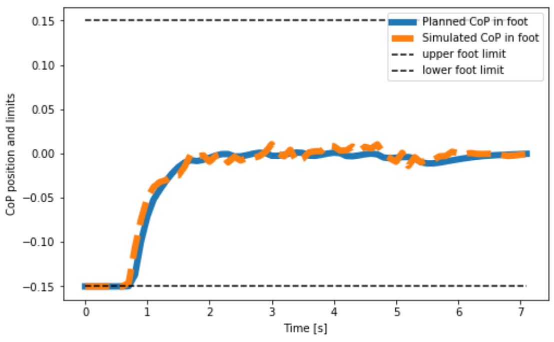


R became to 1000

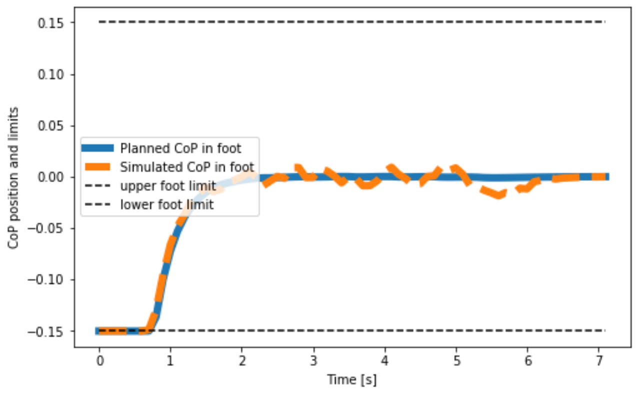
With noise:



Q became to 10^9



Q has no change



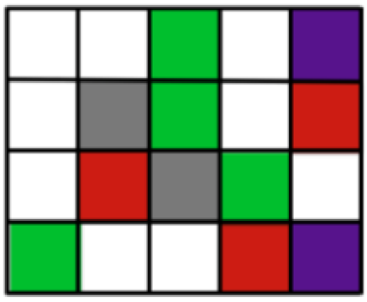
R became to 1000

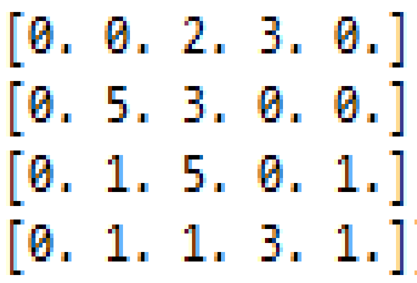
Exercise 2:

a) The iteration times = 1554

The final policy: 0:up 1:down 2:left 3:right 5:block part

Threshold: 10^{-6}



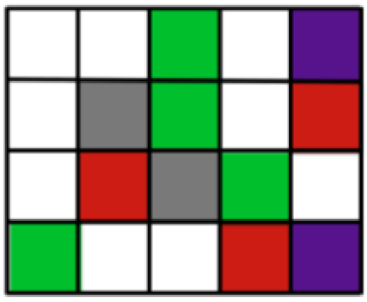
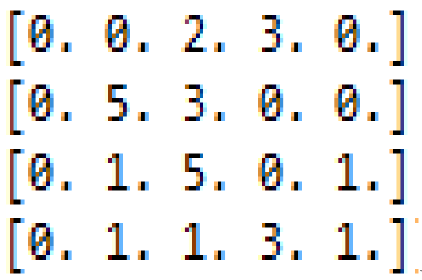


```
[0. 0. 2. 3. 0.]
[0. 5. 3. 0. 0.]
[0. 1. 5. 0. 1.]
[0. 1. 1. 3. 1.]
```

Policy

b) The policy iteration is only twice, but in every policy iteration it used 1791 times iterations at first, and 1554 times iterations at second to make the cost converge.

Threshold: 10^{-6}

```
[0. 0. 2. 3. 0.]
[0. 5. 3. 0. 0.]
[0. 1. 5. 0. 1.]
[0. 1. 1. 3. 1.]
```

Policy

c) In Policy Iteration algorithms, you start with a random policy, then find the value function of that policy (policy evaluation step), then find a new (improved) policy based on the previous value function, and so on. In this process, each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Given a policy, its value function can be obtained using the Bellman operator.

In Value Iteration algorithms, you start with a random value function and then find a new (improved) value function in a iterative process, until reaching the optimal value function. Notice that you can derive easily the optimal policy from the optimal value function. This process is based on the Optimality Bellman operator.