**Record about the project**

**1. Open AI gym**

installation can be found in Youtube; Also, the mujoco environment installation;

**2. mujoco environment**

**2.1 rope issue**

Now the biggest issue is the rope.

We have three ways to define the rope model: composite rope; tendon; chain

Now we use the MIT model as our model (chain).

**2.2 render issue**

**2.3 time issue**

When using mj\_data.time simulating the rope, it always occurs error in run function. For example, it would get stuck in uncertain loop. Through checking the code file, I found in the mujoco, time might occur error, which it means if the real time is 0.8s, and it is not the end time, but mj\_data.time would reset itself, and cound from 0.05s.

Fix the issue through breaking points, and checking each step to add an if to prevent the issue.

**3. algorithm**

**4. plot figures**

**5. evaluation methods**

For all the algorithms performance, we had better use the same method to evaluate them. When I looked through TD3 algorithm, I noticed the author would evaluate xxx epochs in the whole training process per several hundred times, and using the average reward plotting the figures. Also, the evaluation should no more exploration, just exploitation.

Code for oiac:

import numpy as np

import numpy.linalg as la

class ada\_imp\_con( ):

"""Online impedance adaptation"""

def \_\_init\_\_(self, dof):

self.DOF = dof# degree of freedom of a robot arm

self.k\_mat = np.mat(np.zeros((self.DOF, self.DOF)))#stiffness parameter matrix

self.b\_mat = np.mat(np.zeros((self.DOF, self.DOF)))#damping parameter matrix

self.ff\_tau\_mat = np.mat(np.zeros((self.DOF, 1)))

self.q = np.mat(np.zeros((self.DOF, 1)))#real joint angle matrix

self.q\_d = np.mat(np.zeros((self.DOF, 1)))#desired joint angle matrix

self.dq = np.mat(np.zeros((self.DOF, 1)))#real joint velocity matrix

self.dq\_d = np.mat(np.zeros((self.DOF, 1)))#desired joint velocity matrix

self.a = 0.2

self.b = 5.0

self.k = 0.05

def update\_impedance(self, q, q\_d, dq, dq\_d):#tune stiffness and damping matrices, see Eq.(2)

#copy inputs

self.q = np.mat(np.copy(q)).T#real joint angle matrix

self.q\_d = np.mat(np.copy(q\_d)).T#desired joint angle matrix

self.dq = np.mat(np.copy(dq)).T#real joint velocity matrix

self.dq\_d = np.mat(np.copy(dq\_d)).T#desired joint velocity

#update stiffness K and damping B

self.k\_mat = (self.gen\_track\_err() \* self.gen\_pos\_err().T)/self.gen\_for\_factor()

self.b\_mat = (self.gen\_track\_err() \* self.gen\_vel\_err().T)/self.gen\_for\_factor()

return self.k\_mat, self.b\_mat

def gen\_pos\_err(self):#position error, see Eq. (1)

return (self.q - self.q\_d)

def gen\_vel\_err(self):#velocity error, see Eq. (1)

return (self.dq - self.dq\_d)

def gen\_track\_err(self):#tracking error, see Eq. (3)

return (self.gen\_vel\_err() + self.k \* self.gen\_pos\_err())

def gen\_ad\_factor(self):#adaptation scalar, see Eq. (3)

return self.a/(1.0 + self.b \* la.norm(self.gen\_track\_err()) \* la.norm(self.gen\_track\_err()))

"""

#Pseudocode

if \_\_name\_\_ == "\_\_main\_\_":

self.ada\_imp = aic.ada\_imp\_con(dof) # degree of freedom of robot arm

while(t<t\_max):

...

#get robot arm joint angles and velocity

#data-driven learning or opmtimization of feedforward joint torque(s) tau\_ff

self.ada\_imp.(q, q\_d, dq, dq\_d) #undate stiffness (self.k\_mat) and damping (self.b\_mat) matrices

tau\_fb = self.k\_mat\* (q\_d-q) + self.k\_mat\*(dq\_d-dq) #compute feedback joint torque(s)

tau = tau\_ff + tau\_fb #compute total joint torques

...

"""