import pybullet as p

import time

import pybullet\_data

import numpy as np

import matplotlib

matplotlib.use('Qt5Agg')

import matplotlib.pyplot as plt

class Biped3D(object):

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

self.physicsClient = p.connect(p.GUI) # or p.DIRECT for non-graphical version

p.resetDebugVisualizerCamera(cameraDistance=0.6, cameraYaw=0,

cameraPitch=0, cameraTargetPosition=[0, 0, 0.2])

#修改了相机的参数，有利于看出来两只脚的离地高度，虽然还没有研究调整成正交而不是投影，也没有弄两个机位，但是可以看出来摆动脚是没有碰到地面的

p.setAdditionalSearchPath(pybullet\_data.getDataPath()) # optionally

self.ground = p.loadURDF("plane.urdf")

self.robot = p.loadURDF("/yanshee/robots/yanshee.urdf", [0, 0, -0.1], [0, 0, 0, 1], useFixedBase=False)

self.joints = self.get\_joints()

self.n\_j = len(self.joints)

self.simu\_f = 200 # Simulation frequency, Hz

p.setTimeStep(1.0/self.simu\_f)

self.motion\_f = 2 # Controlled motion frequency, Hz

self.world\_joint = 0

self.robo\_pose = 0

self.l1 = 0.05 # leg length 1

self.l2 = 0.06 # leg length 2

self.support\_leg = 'left' # init left

self.step\_length = 0.04

self.T = 0 # determined at run time

self.n\_T = 1

self.mid\_speed = 0.1 # speed at mid point

self.omega = np.sqrt(9.81 / 0.1) # $angular velocity = \sqrt{g / z}$

self.x\_0 = -self.step\_length/2

self.x\_dot\_0 = np.sqrt(self.mid\_speed \*\* 2 + self.omega \*\* 2 \* (self.step\_length / 2) \*\* 2)

self.cosh = lambda x : (np.exp(x) + np.exp(-x)) / 2

self.sinh = lambda x : (np.exp(x) - np.exp(-x)) / 2

# 一些全局的数据。虽然这个lab并不很重视软件工程，但是这里使用了一个类，在这个构造函数里面有些部分是全局可以访问的数据，比如运行时生成的或者一些flag，比如下面的q\_mat，robo\_pose之类的，有些是则是参数，比如上面的了l1 l2长度。实际上我认为这样写不是特别清楚。感觉python+numpy有时候有点软件工程又有点函数式编程，然后还因为要控制很多外设缝合了很多同步的问题，希望能分开点。

self.iq\_vec = np.zeros(self.n\_j)

self.q\_vec = np.zeros(self.n\_j)

self.dq\_vec = np.zeros(self.n\_j)

self.q\_mat = np.zeros((self.simu\_f \* 3, self.n\_j)) # record data of 3 seconds

self.q\_d\_mat = np.zeros((self.simu\_f \* 3, self.n\_j)) # record data of 3 seconds

self.init\_plot()

def run(self):

for i in range(int(1e3)):

t = i / self.simu\_f

torque\_array = self.controller(t)

self.q\_vec, self.dq\_vec = self.step(torque\_array)

if 0 == i % 20:

self.update\_plot()

time.sleep(5/self.simu\_f)

p.disconnect()

def step(self, torque\_array):

self.robo\_pose = self.set\_motor\_torque\_array(torque\_array, 0.1 + self.robo\_pose) # 修改了self.set\_motor\_torque\_array这个函数，多增加了一个joint，也就是机器人和世界的滑动关节的位置，用来体现机器人行走的结果。

p.resetDebugVisualizerCamera(cameraDistance=0.6, cameraYaw=0,

cameraPitch=0, cameraTargetPosition=[self.robo\_pose-0.1, 0, 0.2]) # 根据机器人的位置更新相机的位置

p.stepSimulation()

self.q\_mat[:-1] = self.q\_mat[1:]

self.q\_mat[-1] = self.q\_vec

return self.get\_joint\_states()

def get\_joints(self):

all\_joints = []

for j in range(p.getNumJoints(self.robot)):

# Disable motor in order to use direct torque control.

info = p.getJointInfo(self.robot, j)

joint\_type = info[2]

if (joint\_type == p.JOINT\_REVOLUTE):

all\_joints.append(j)

p.setJointMotorControl2(self.robot, j,

controlMode=p.POSITION\_CONTROL, force=0)

if (joint\_type == p.JOINT\_PRISMATIC):

self.world\_joint = j

p.setJointMotorControl2(self.robot, j,

controlMode=p.POSITION\_CONTROL, force=0)

joints = all\_joints[0:]

print("All joints are", all\_joints)

# print("Number of All Joints:", p.getNumJoints(self.robot))

# print("Number of All Revolute Joints:", joints)

return joints

def get\_joint\_states(self):

'''

:return: q\_vec: joint angle, dq\_vec: joint angular velocity

'''

q\_vec = np.zeros(self.n\_j)

dq\_vec = np.zeros(self.n\_j)

for j in range(self.n\_j):

q\_vec[j], dq\_vec[j], \_, \_ = p.getJointState(self.robot, self.joints[j])

return q\_vec, dq\_vec

def set\_motor\_torque\_array(self, torque\_array = None, robot\_posi=0):

assert torque\_array is not None

for j in range(len(self.joints)):

p.setJointMotorControl2(self.robot, self.joints[j], p.POSITION\_CONTROL, targetPosition=torque\_array[j])

p.setJointMotorControl2(self.robot, self.world\_joint, p.POSITION\_CONTROL, targetPosition=robot\_posi) # 设定机器人与世界的相对位置

return robot\_posi # 函数式编程（doge

def controller(self, t, type='joint'):

if 'joint' == type:

return self.joint\_controller(t)

def joint\_controller(self, t):

# $x(t) = x(0)cosh(\omega t) + \frac{\dot{x(0)}}{\omega} sinh(\omega t)$

# $\dot{x(t)} = x(0)\omega sinh(\omega t) + \dot{x(0)} cosh(\omega t)$

# $\omega$ = `self.omega`

# $x(0)$ = `-self.step\_length/2`

# $\frac{1}{2} \dot{x}^2 - \frac{g}{2z} x^2 = E (Orbital Energy)$

# $\dot{x(0)} = `np.sqrt(self.mid\_speed \*\* 2 + self.omega \*\* 2 \* (self.step\_length / 2) \*\* 2)`

# 上面讲的是公式

x\_0 = self.x\_0

x\_dot\_0 = self.x\_dot\_0

cosh = self.cosh

sinh = self.sinh

y = 9.81 / self.omega \*\* 2

t -= self.T \* self.n\_T

x = x\_0 \* cosh(self.omega \* t) + x\_dot\_0 \* sinh(self.omega \* t) / self.omega

self.robo\_pose = self.n\_T \* self.step\_length - self.step\_length / 2 + x

# 算出x和roboposi

if x > self.step\_length / 2 and self.T == 0:

self.T = t

if self.support\_leg == 'left':

self.support\_leg = 'right'

else:

self.support\_leg = 'left'

if self.T != 0 and t > self.T:

self.n\_T += 1

if self.support\_leg == 'left':

self.support\_leg = 'right'

else:

self.support\_leg = 'left'

# 左右脚切换

x += 0.001

####################### Notations ####################

# q, dq | d

# q stands for angle and dq stand for velocity | d stand for desired

# a, k, h | p, w

# a stands for ankle, k for knee and h for hip | p stand for support and w for swing

q\_d\_a\_p, q\_d\_k\_p, \_ = self.ik\_planner\_2R(x, y)

q\_d\_h\_p = np.pi / 2 - q\_d\_a\_p - q\_d\_k\_p

# print('q\_d\_h\_p = ', q\_d\_h\_p/np.pi\*180, ', q\_d\_k\_p = ',

# q\_d\_k\_p/np.pi\*180, ', q\_d\_a\_p = ', q\_d\_a\_p/np.pi\*180)

q\_d\_h\_w, q\_d\_k\_w, \_ = self.ik\_planner\_2R(self.step\_length / 2, y / 5, bit=0)

q\_d\_h\_w = q\_d\_h\_w - np.pi/2

# q\_d\_k\_w remain unchanged

q\_d\_a\_w = -q\_d\_h\_w - q\_d\_k\_w - np.pi / 12

if self.T != 0:

if x < 0: \_y = y/5

if x > 0: \_y = y

q\_d\_h\_w, q\_d\_k\_w, \_ = self.ik\_planner\_2R((t/self.T-0.5)\*self.step\_length, \_y, bit=0)

q\_d\_h\_w = q\_d\_h\_w - np.pi/2

# q\_d\_k\_w remain unchanged

q\_d\_a\_w = -q\_d\_h\_w - q\_d\_k\_w - np.pi / 12

print('q\_d\_h\_w = ', q\_d\_h\_w/np.pi\*180, ', q\_d\_k\_w = ',

q\_d\_k\_w/np.pi\*180, ', q\_d\_a\_w = ', q\_d\_a\_w/np.pi\*180)

if self.support\_leg == 'left':

q\_d\_vec = np.array([-q\_d\_h\_p, -q\_d\_k\_p, (np.pi/2 - q\_d\_a\_p)\*0.9,

q\_d\_h\_w, q\_d\_k\_w, q\_d\_a\_w\*0.8])

else:

q\_d\_vec = np.array([-q\_d\_h\_w, -q\_d\_k\_w, -q\_d\_a\_w\*0.8,

q\_d\_h\_p, q\_d\_k\_p, (q\_d\_a\_p - np.pi/2)\*0.9])

self.q\_d\_mat[:-1] = self.q\_d\_mat[1:]

self.q\_d\_mat[-1] = q\_d\_vec

return q\_d\_vec

# call了一个运动学逆解，用来计算支撑脚和摆动脚的关节角度

# dq\_d\_vec = np.array([0, 0, 0, 0, 0, 0])

# # PID controller gains

# P\_vec = np.array([100, 50, 50, 100, 50, 50])

# I\_vec = np.array([0.05, 0.05, 0.05, 0.05, 0.05, 0.05])

# D\_vec = np.array([0.05, 0.05, 0.05, 0.05, 0.05, 0.05])

# return self.joint\_PID\_controller(P\_vec, I\_vec, D\_vec, self.q\_vec, self.dq\_vec, q\_d\_vec, dq\_d\_vec) # 这些没用了，因为是位置控制

def ik\_planner\_2R(self, x, y, bit=1):

l1 = self.l1

l2 = self.l2

assert y > 0

assert l1 > 0

assert l2 > 0

theta2 = np.pi - np.arccos((l1\*\*2+l2\*\*2-x\*\*2-y\*\*2)/(2\*l1\*l2))

theta\_avg = np.arctan2(y, x)

theta\_delta = np.arctan2(l2\*np.sin(theta2), l1+l2\*np.cos(theta2))

if bit == 1:

theta1 = theta\_avg - theta\_delta

else:

theta1 = theta\_avg + theta\_delta

theta2 = -theta2

return theta1, theta2, bit

# 因为支撑脚和摆动脚可以抽象成二维平面的2R机械臂的运动学逆解的问题，接下来就是用身体竖直和脚要水平来算出来剩下的一个关节角度：

# 支撑脚的脚踝关节是坐标原点，然后利用hip关节的LIP的轨迹算出来ankle和knee的角度。再利用身体要竖直这一特点计算出hip的角度。

# 摆动脚的hip是坐标原点，然后因为身体在支撑脚前面的时候，摆动脚要逐渐伸到身体前面准备切换，因此让两个脚关于身体对称，计算尺hip和knee的角度。然后再利用脚面与地面平行的几何约束确定出ankle的角度。

def joint\_PID\_controller(self, P\_vec, I\_vec, D\_vec, q\_vec, dq\_vec, q\_d\_vec, dq\_d\_vec):

self.iq\_vec = self.iq\_vec + (q\_d\_vec - q\_vec)

return P\_vec\*(q\_d\_vec-q\_vec) + I\_vec\*self.iq\_vec + D\_vec\*(dq\_d\_vec-dq\_vec)

# 这个pid controller没有用

def init\_plot(self):

self.fig = plt.figure(figsize=(5, 9))

joint\_names = ['left\_hip', 'left\_knee', 'left\_ankle', 'right\_hip', 'right\_knee', 'right\_ankle']

self.q\_d\_lines = []

self.q\_lines = []

for i in range(6):

plt.subplot(6, 1, i+1)#多画一点

q\_d\_line, = plt.plot(self.q\_d\_mat[:, i], '-')

q\_line, = plt.plot(self.q\_mat[:, i], '--')

self.q\_d\_lines.append(q\_d\_line)

self.q\_lines.append(q\_line)

plt.ylabel('q\_{} (rad)'.format(joint\_names[i]))

plt.ylim([-3, 3])

plt.xlabel('Simulation steps')

self.fig.legend(['q\_d', 'q'], loc='lower center', ncol=2, bbox\_to\_anchor=(0.49, 0.97), frameon=False)

self.fig.tight\_layout()

plt.draw()

def update\_plot(self):

for i in range(6):

self.q\_d\_lines[i].set\_ydata(self.q\_d\_mat[:, i])

self.q\_lines[i].set\_ydata(self.q\_mat[:, i])

plt.draw()

plt.pause(0.001)

if \_\_name\_\_ == '\_\_main\_\_':

robot = Biped3D()

robot.run()

整个lab的分析过程也附在后面了，因为不是实验报告，因此更接近草稿的程度。这里总结一下思路和一些优缺点：

思路：虽然不是真正的仿真，但是body的运动轨迹是按照2D LIP计算出来的。虽然不是控制关节力矩，但是由于关节角度与使用LIP轨迹逆解出来的，因此力矩自然是符合LIP的。这样一个支撑脚的phase就确定了。关于摆动脚，则需要在换脚的时候到达身体前面半个步长。这样让支撑脚和摆动脚到身体的距离相同，就非常的自然了。摆动脚距离地面的高度不像是支撑脚是恒定的，摆动脚距离身体越近，高度就越高，就是这样。

优点：摆动脚的切换方式与轨迹规划成功利用了同一套ik代码

缺点：经常有一些小地方，是通过人为调参，比如角度增加几度，位置增加一点点，这样凑出来一个好一点的结果的。说明现在机器人的模型和控制的抽象还是不够，无法完美抽象出来。