HW1 - Robot Autonomy Plus

Professor: Oliver Kroemer, TA's: Steven Lee, Ami Sawhney

Due: November 17 2020, 11:59pm

1 Introduction

This homework will focus on the topics of kinematics and control from the lectures. You will explore force and impedance control on a simple scene in the VREP simulator for the first part, and then implement forward and inverse kinematics for the Locobot and visualize your implementations using VREP.

Software installation and setup:

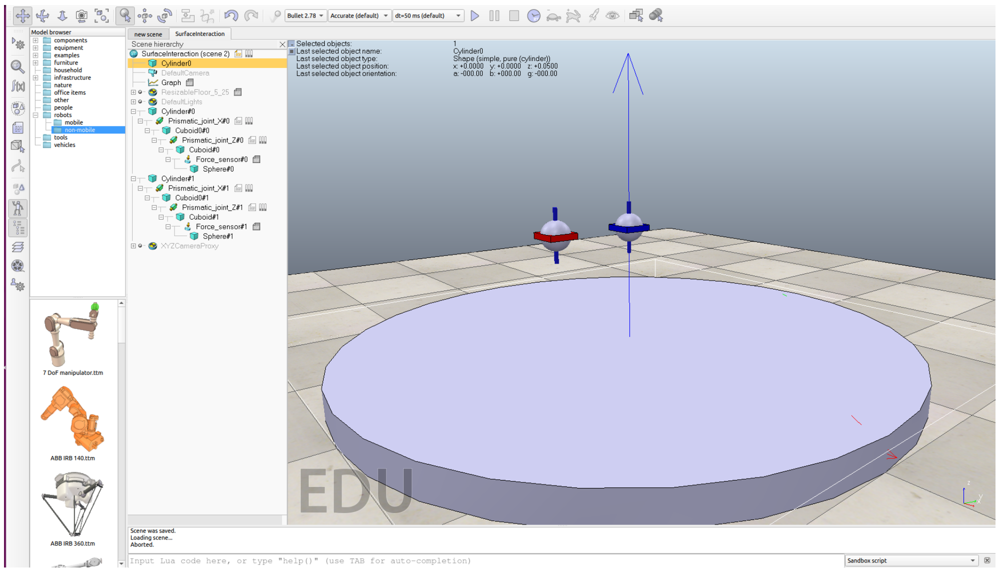
In order to complete this and future assignments, you will need to either download the Virtual Machine file posted on Slack OR install Pyrobot/VREP on your computer. Please see the Slack post from Steven for the link containing detailed setup instructions.

2 PID Control

In this section, you will implement PID controllers with a simple scene in VREP (aka Coppelia Sim) to help familiarize yourself with the simulation environment and understand the basics of PID control.

First, launch VREP and then click File, Open Scene, then open the .ttt file in the HW1/Control folder called SurfaceInteraction.ttt.

The scene should look like this:



```
Selected objects:                    1
Last selected object name:           Cylinder0
Last selected object type:           Shape (simple, pure (cylinder))
Last selected object position:       x: +0.0000   y: +0.0000   z: +0.0500
Last selected object orientation:    a: -000.00   b: +000.00   g: -000.00
```

```
Scene was saved.
Loading scene...
Aborted.
Input Lua code here, or type "help()" (use TAB for auto-completion)
```

If you look in the Scene Hierarchy tree on the left of the window, you will see the two elements below:

$Prismatic\_joint\_Z\#0$: Red sphere

$Prismatic\_joint\_Z\#1$: Blue sphere

Problem 1:

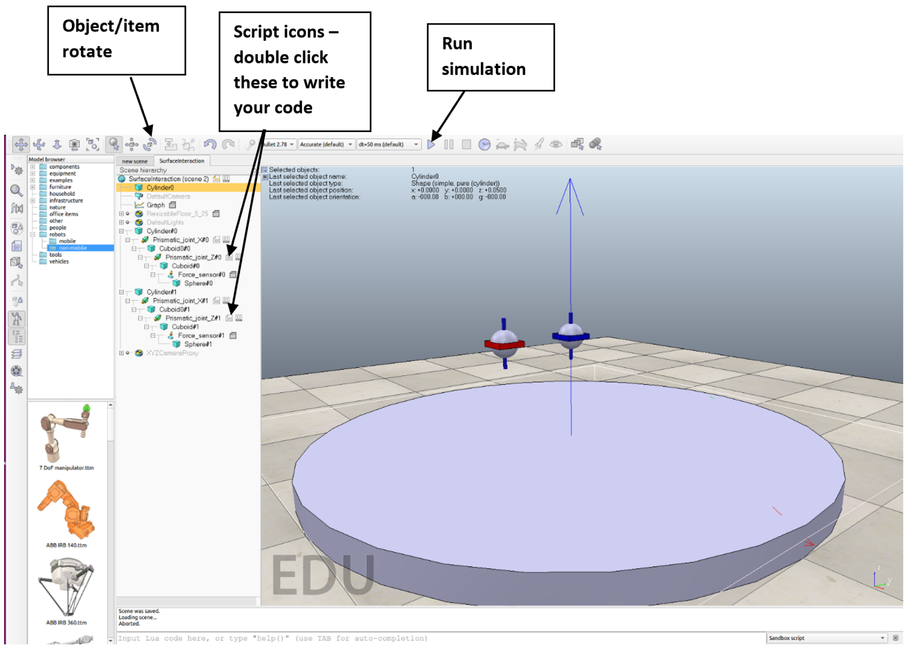
Implement an Impedance controller on the Red sphere ($Prismatic\_joint\_Z\#0$) to maintain a constant force of -10 N. Since this is impedance control, you will need to determine what target position to set to achieve the desired force.

Problem 2:

Implement a Force controller on the Blue sphere ($Prismatic\_joint\_Z\#1$) to maintain a constant force of 10 N.

Implementation guidance:

Double click on the script icon next to each of the above objects in the tree - this script is where you'll be implementing your impedance and force controllers (in Lua). We've provided you with some starter code to work with. The comments in the code will guide you through what parts of the code you should modify and write yourself. You only need to edit the function called sysCall_jointCallback() to implement your controllers.



To help you with selecting PID gains for your force and impedance controllers, we've provided you with a general range of what you might want to set your gain values to:

Force control gains:
P = 0.05 - 0.15, I = 5 - 15, D = 0 (don't use a D term here)

Impedance control gains:
P = 150 - 250, I = 0 (don't use an I term here), D = 2 - 7

Problem 3: Run the simulation (press the start simulation/play button in the top toolbar) with your controllers and plot the forces - are they behaving as expected? Discuss your results and submit screenshots of the force plots.

Problem 4: Now change the orientation of the plane - to do this click the object/item rotate icon in top toolbar, select the orientation tab, and change the beta angle to a nonzero angle (e.g. 45 degrees) and run the simulation.

Discuss (in 1-3 sentences) the results you observe. Discuss the differences in the behavior of the force and impedance controllers.

3 Forward Kinematics for Locobot

In this section you will implement the forward kinematics for the Locobot arm and compute the Jacobian. Please use Python for your implementation. We

Intuitively the impedance controller is trying to force its way into a set point it cannot gain access to hence it overshoots wildly when not constrained to a single axis.

The force controller essentially has access to the "ground truth" so it can correct for the force it applies.

have given you given you starter code in LocoBot.py and RobotUtil.py in the HW1/Kinematics folder. You may use the functions in RobotUtil.py to aid your implementation or you can implement your own functions. Please write your code to compute the FK and Jacobian in the FK method in the ForwardKin() method in the Locobot class in Locobot.py. Your FK method should output the computed transforms for each of the joints based on the input angles and the Jacobian matrix.

The lecture slides should provide guidance to help you with the implementation. Remember that you will need to adapt your code from what is shown in the lecture slides to take into account the fact that not all joints for this robot are defined to rotate about the z axis. To determine the axis of rotation for each joint, you should look at the URDF file called interbotix_locobot_description.urdf. In the urdf under each joint you will see (for example): (axis xyz="0 1 0"/), which defines the axis of rotation.

Test your FK implementation: Run your FK method for the following joint configurations to compute the end effector pose. Submit your computed end effector poses for each set of joint positions below (note: these joint positions are also defined as a list called "joint_targets" in the MainTest.py file).

$$q_1 = \begin{bmatrix} 0^\circ \\ 0^\circ \\ 0^\circ \\ 0^\circ \\ 0^\circ \end{bmatrix}, \quad q_2 = \begin{bmatrix} -45^\circ \\ -15^\circ \\ 20^\circ \\ 15^\circ \\ -75^\circ \end{bmatrix}, \quad q_3 = \begin{bmatrix} 30^\circ \\ 60.^\circ \\ -65^\circ \\ 45^\circ \\ 0^\circ \end{bmatrix}$$

You can check that your results with the following answers (Try to get within a few millimeters for translation and 2 or 3 decimal places for the rotation values):

```
computed FK ee position [5.170250e-01 1.000000e-04 4.193268e-01]
computed FK ee rotation
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

$$t_1 = \begin{bmatrix} 0.518 \\ 0 \\ 0.414 \end{bmatrix} R_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
computed FK ee position [ 0.34547502 -0.25488326  0.3462286 ]
computed FK ee rotation
[[ 0.66446302  0.4166168  -0.62041867]
 [-0.66446302 -0.0505914  -0.74560673]
 [-0.34202014  0.90767337  0.24321035]]
```

$$t_2 = \begin{bmatrix} 0.349 \\ -0.251 \\ 0.346 \end{bmatrix} R_2 = \begin{bmatrix} 0.667 & 0.418 & -0.619 \\ -0.664 & -0.050 & -0.746 \\ -0.343 & 0.907 & 0.244 \end{bmatrix}$$

```
computed FK ee position [0.55687573 0.26545164 0.18333242]
computed FK ee rotation
 [[ 0.66341395 -0.5        0.5566704 ]
 [ 0.38302222  0.8660254   0.3213938 ]
 [-0.64278761  0.         0.76604444]]
```

$$t_3 = \begin{bmatrix} 0.554 \\ 0.264 \\ 0.179 \end{bmatrix} R_3 = \begin{bmatrix} 0.662 & -0.499 & 0.559 \\ 0.382 & 0.866 & 0.322 \\ -0.645 & 0 & 0.764 \end{bmatrix}$$

Testing your implementation using Pyrobot simulator:
To visualize your FK implementation, run the file Pyrobot_FK_IK.py. Note, you will have to open another terminal and first start a roscore before running this script (simply type "roscore" in another terminal). This file will load in the Locobot into the VREP simulator and allow you to visualize the robot in the joint configurations defined above.

4 Inverse Kinematics for Locobot

In this section you will implement the inverse kinematics for the Locobot arm using the Jacobian transpose or pseudoinverse method (you may choose to implement one or both). Please use Python for your implementation. Write your code in the IK method in the IterInvKin() method in the Locobot class in Locobot.py.

Test your IK implementation: Run your IK method for starting joint configuration and goal end effector pose defined in the MainTest.py file (called qInit and HGoal in the code, respectively) and compute the joint configurations to achieve the desired end effector pose.

Discuss your results and observations in 1-3 sentences. How well did your Inverse Kinematics implementation perform?

General notes:

Please be concise in your responses to the questions (1-3 sentences maximum).

You are encouraged to work together and discuss questions with your peers, but please write your own code for the implementations.

IK in general can be pretty unstable sometimes, so it may take some time to get hyperparameters that work properly.

```
 # Jacobian Pseudo-Inverse cannot be computed @ Step 4 even after reducing step sizes multiple times
 # Pos Error(m): [0.41623425993423047, -0.8802044262027227, -0.22801142285012657], Rotation Error(rad): 1.1072814432454838

# Switching to Jacobian Transpose Method

# Step: 1000
# Pos Error(m): [0.7712178398298731, -0.6167892348244101, -0.15746137092272164], Rotation Error(rad): 0.767704823590883
# computed IK angles[-0.02715018929923225, -0.5629399578355448, 0.7531900187871879, 0.290465299712724, 0.610573795437234]

# Step: 2000
# Pos Error(m): [0.8471681122638188, -0.507206807657038, -0.1582638424584537], Rotation Error(rad): 0.8643796036834195
# computed IK angles [-0.03371279976628066, -0.45473591137218317, 0.7042210161901095, 0.1981941678980578, 0.7528826325651012]
```

Oscillatory errors observed.