ARO Practical Guidebook 2022

Project: Nextage Robot

Lecturer: Steve Tonneau
School of Informatics, University of Edinburgh

General Instructions

- This practical consists of 3 major tasks each of which is a building block for the upcoming tasks.
- You should use Python for implementing your solutions as this will provide a consistent experience with the labs. Set up the environment by following the instructions in the “Setting Up” section and download the skeleton from the Git repository.
- In order to further help with some of the implementations, we have provided a set of tools and skeleton code for you.
- The evaluation criteria on which you will be judged includes the quality of the textual answers and/or any plots asked for, and code will be needed to generate the results.
- Read the instructions carefully, implement what is required and only that. Keep your code human readable and efficient. Make comments to help people review your code. The running time of your algorithms should be around 10 milliseconds.
- For answers involving floating point numbers, it is not required to round up to some decimal places.

Important dates

> You **must** submit this practical (**report and code**) by **Friday 25 Nov, 12:00**.

Late submission is penalised, e.g. 5 percent of your awarded mark will be deducted for every calendar day or part thereof it is late, up to a maximum of 7 calendar days.

Please refer to the ITO Website for further details:
Late Coursework & Extension Requests
Academic Misconduct

Part zero: Preliminaries

This section is for people who have little experience with Bash commands and Python, feel free to skip if you are already familiar with those.

- A detailed introduction to Linux bash commands: Simple Linux Commands
- Essential skills in Python: ARO Tutorial 1
- Python library: *Pybullet* simulation library

Setting Up

Ubuntu 20.04 LTS is recommended for this practical. The provided code should be also compatible with Windows 10, MacOS and most other Linux distributions with minor tweaks. You are strongly advised to use DICE environment for this practical. We do not guarantee support if you are working on your own device. If you need more guidance, remember to post on Piazza!

Get the code from:

https://github.com/avro22/ARO_Practical_2022

Please follow the instructions from the Github repository carefully and verify your environment with the “hello-world” programs provided. You should see the Nextage robot in colours.

Once finished, you are good to go!

Submission

To submit this practical, make a .zip file containing all the tasks. The template code is not required, and will not affect your final grade if you do include them in your submission. Check out Files to submit. Once you are ready to submit, log on Learn and submit via:

> **Learn > Assessment > Assignment Submission > “Coursework 2 and Practical Test: report and python code”**

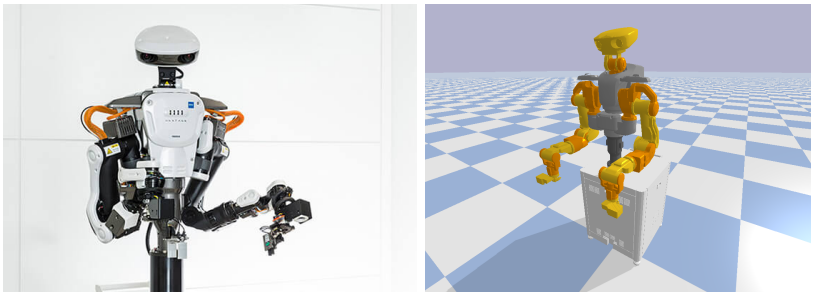
Before you start

Warnings

> **DO NOT** modify anything in the template code. More detail in below sections. You could utilise any API provided to debug your code, but make sure that some prohibited APIs are **NOT** included in your final solution before submission. For more details, please see Appendix 3: Prohibited APIs.

Nextage robot

You are given a 3D-model of a **Nextage robot (**Picture 1, 2**)** with physics configurations: visual mesh data, mass, collision geometry etc.



Picture 1 (Left): The Nextage Robot (Image credit: Kawada)
Picture 2 (Right): The 3D-model of Nextage Robot

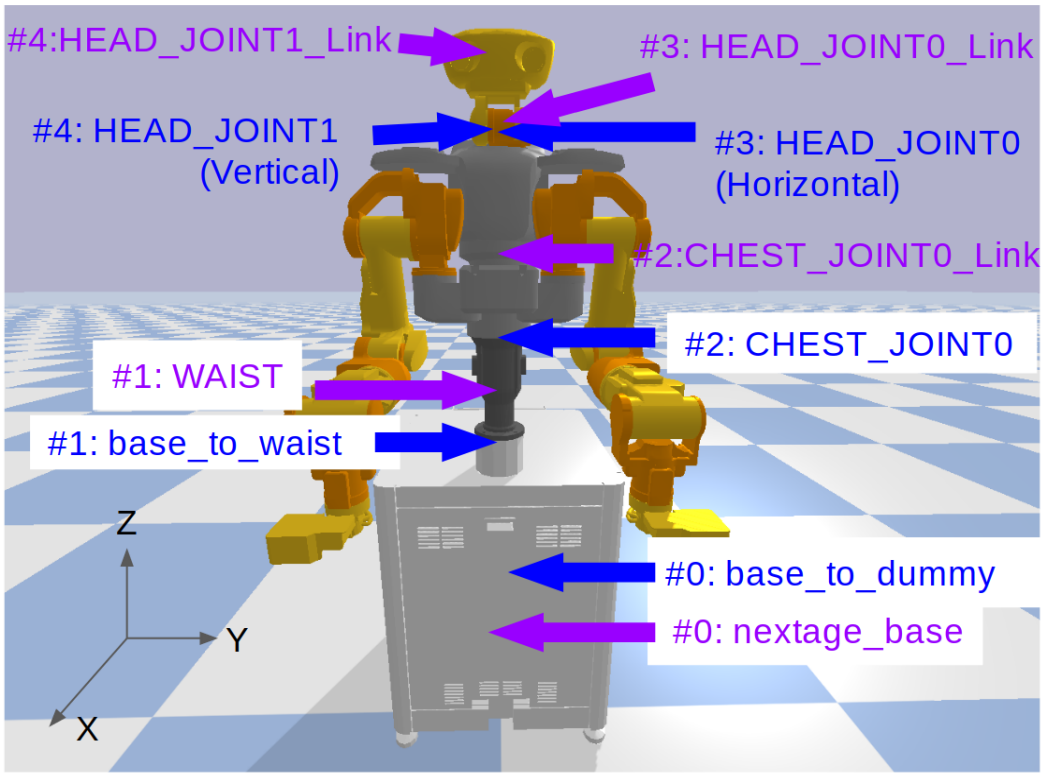
The model consists of **17** joints in total:

- 1 for dummy joint / waist joint / chest joint each
- 2 for head joints
- 6 for joints for left/right arm

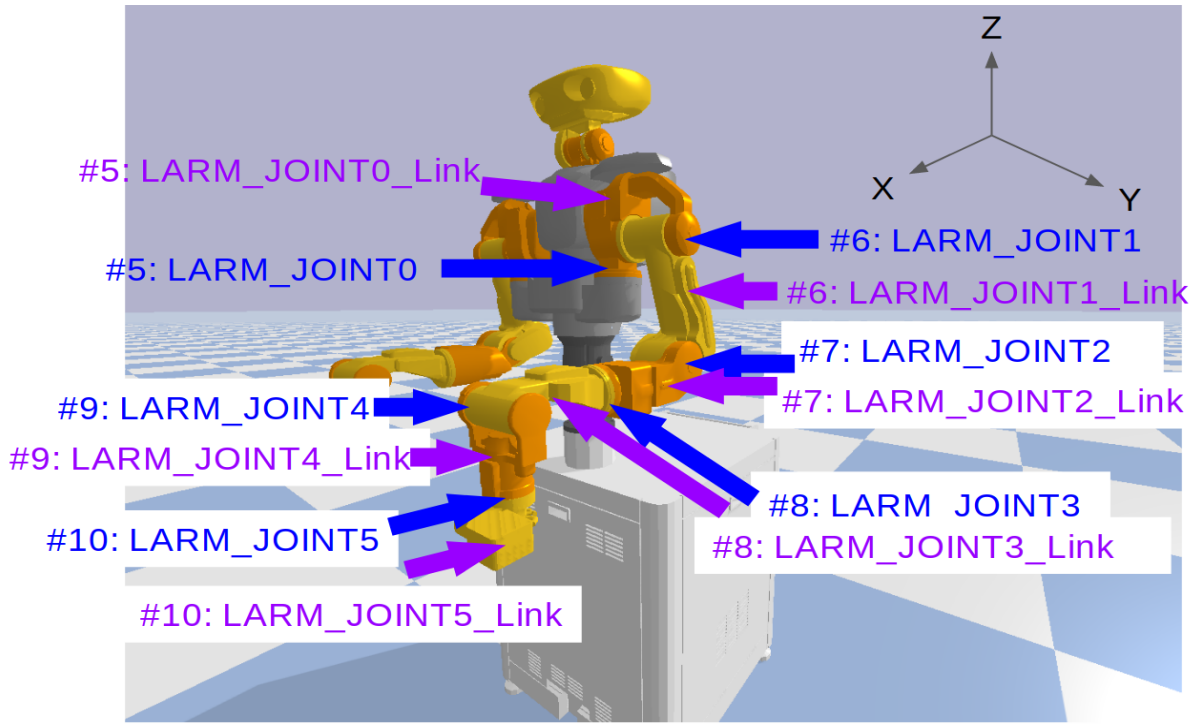
There are exactly 17 links and 17 joints. That is, there are 17 link-joint pairs where the joint and the link share the same id. Except the first joint, "base_to_dummy" which is a virtual joint, all the others are revolute joints.

In Picture 3,4, and , the links are all coloured differently from adjacent ones. The blue arrows in the picture indicate the position of the joints along with a label in the format of "*#id of the joint*: *name of the joint*". Similarly, the purple arrows and labels are for the corresponding links.

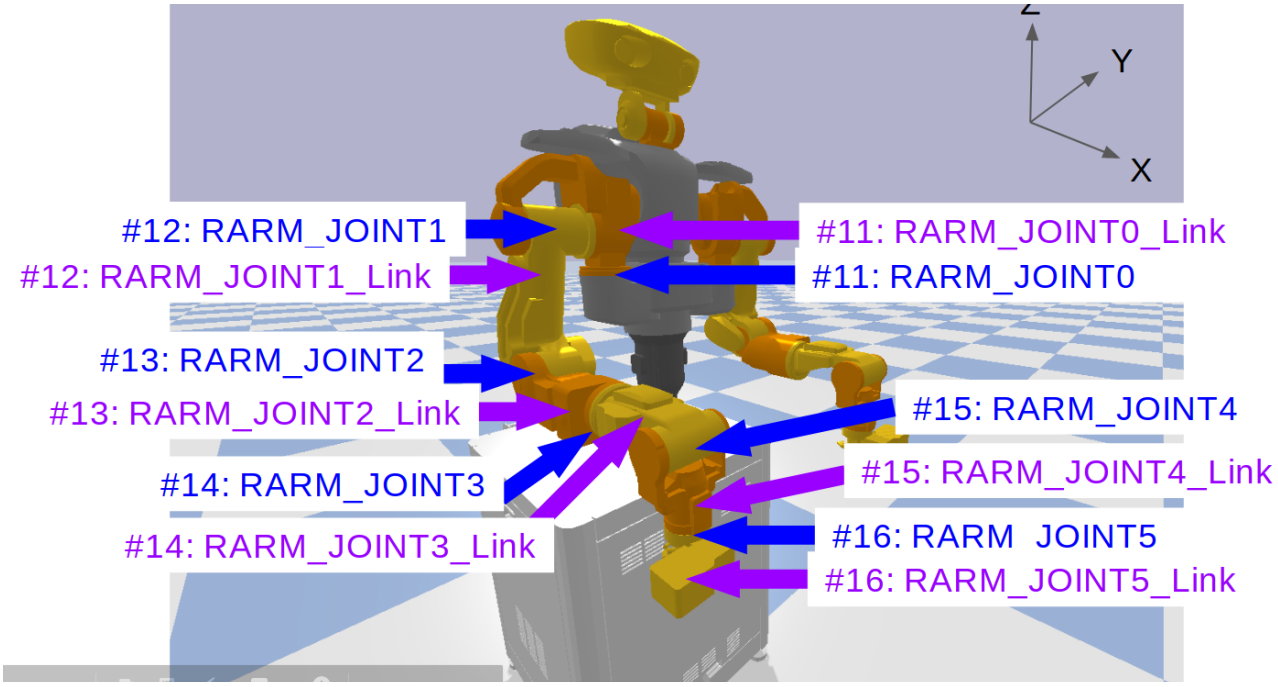
For more details, you can check with *Simulation.getJointInfo()*.



Picture 3: Nextage robot - Front view



Picture 4: Nextage robot - Left view

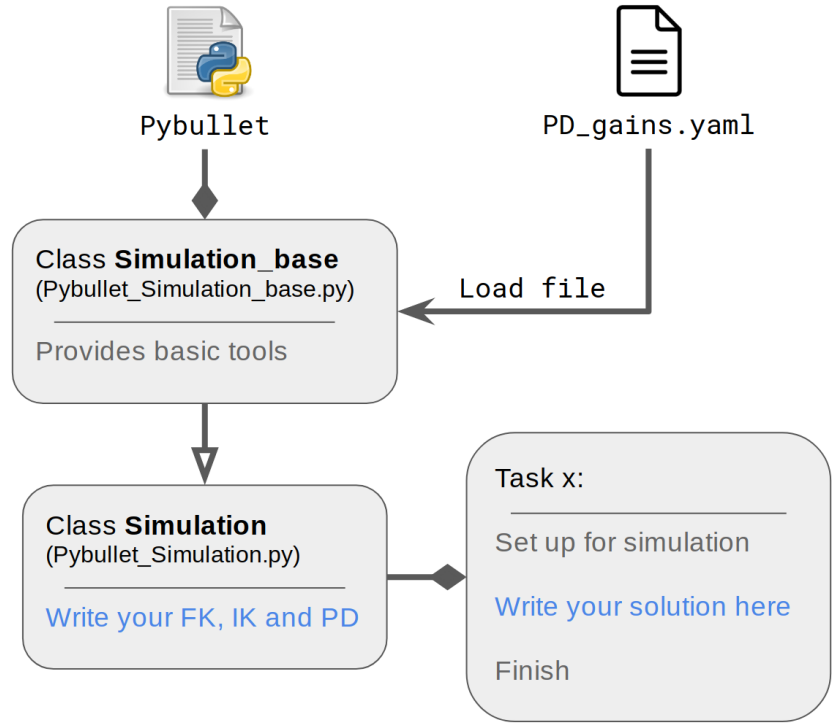


Picture 5: Nextage robot - Right view

The skeleton

Let's have a look at the skeleton files.

```
./ARO_Practical_2022
├── core
│   ├── nextagea_description
│   │   └── Nextagea_robot_model
│   ├── Pybullet_Simulation_base.py
│   ├── Pybullet_Simulation_template.py
│   ├── Pybullet_Simulation.py
│   ├── PD_gains_template.yaml
│   └── PD_gains.yaml
├── taskx
│   ├── lib
│   │   └── ...
│   ├── taskx_template.py
│   └── taskx.py
├── environment.yml
├── requirement.txt
├── hello_Nextage.py
├── hello_world.py
└── README.md
```



Picture 6 (left): File tree of the template
Picture 7 (right): Template UML diagram

The file tree (Picture 6) shows an overall structure of the files. The files in coloured in blue are the template files, the files in purple are the files that are not provided, but you should create from the corresponding template file on your own. They are critical for the simulation and submission. There is more information about submission from each task. The library files, in **black**, helps you to set up and run the simulation.

All files with a black filename in the picture should **NOT** be modified

At the same time,

You are encouraged to read through all files to **learn and experiment**

All the core functionalities are under the *'core'* folder, including the 3D model of the robot, core simulation class and the PD parameter file. For each task, there is a folder called *'taskx'* where *x* is the task number.

The UML diagram (Picture 7) on the right shows the relationship between files and classes: class *Simulation_base* wraps up all the essential APIs from *Pybullet* library for this practical but missing some functionalities. The class *Simulation* inherits class *Simulation_base* and adds more to (colour coded in blue) the parent class. You are responsible for implementing these new methods.

Let's start

You should have had a brief intuition of the set up now, let's dive into the code.

Key parameters

Let's start with *hello_Nextage.py*. This file contains the simplest example with *Simulation*. Line 1~15 imports all necessary libraries in order to run the simulation, including the libraries from the *'core'* folder (Picture 8).

RSS_Practical_2020 > hello_Nextage.py > ...

You, 2 days ago | 1 author (You)

```
1  import subprocess
2  import math
3  import time
4  import sys
5  import os
6  import numpy as np
7  import pybullet as bullet_simulation
8  import pybullet_data
9
10 # setup paths and load the core
11 abs_path = os.path.dirname(os.path.realpath(__file__))
12 root_path = abs_path
13 core_path = root_path + '/core'
14 sys.path.append(core_path)
15 from Pybullet_Simulation_template import Simulation
16       You, 3 days ago • first commit
```

Picture 8: Importing libraries

Line 17~37 defines two config dictionaries *pybulletConfigs* and *robotConfigs* (Picture 9, Table 1) where the keys are self-explanatory. Feel free to play around and see the effects.

```
17  pybulletConfigs = {
18      "simulation": bullet_simulation,
19      "pybullet_extra_data": pybullet_data,
20      "gui": True,
21      "panels": False,
22      "realTime": False,
23      "controlFrequency": 1000,
24      "updateFrequency": 250,
25      "gravity": -9.81,
26      "gravityCompensation": 1.,
27      "floor": True,
28      "cameraSettings": 'cameraPreset1'
29  }
30  robotConfigs = {
31      "robotPath": core_path + "/nextagea_description/urdf/NextageaOpen.urdf",
32      "robotPIDConfigs": core_path + "/PD_gains_template.yaml",
33      "robotStartPos": [0, 0, 0.85],
34      "robotStartOrientation": [0, 0, 0, 1],
35      "fixedBase": False,
36      "colored": True
37  }
```

Picture 9: Config parameters

Parameters in ***pybulletConfigs***

Parameter	Description
simulation	*Pybullet* simulation object
pybullet_extra_data	Directory of *Pybullet* common models
gui	GUI or non-GUI
panels	With panels or not
realTime	Real-time mouse interaction
controlFrequency	1/dt
updateFrequency	Program supervision frequency
gravity	Gravity constant
gravityCompensation	Naive gravity compensation ratio
floor	If load floor
cameraSettings	“cameraSettings{1..5}”

Parameters in ***robotConfigs***

Parameter	Description
robotPath	Robot URDF path
robotPIDConfigs	Robot PID gains config
robotStartPos	Robot initial position
robotStartOrientation	Robot initial orientation
fixedBase	If keep the base fixed
colored	If apply colour

Table 1: Essential parameters

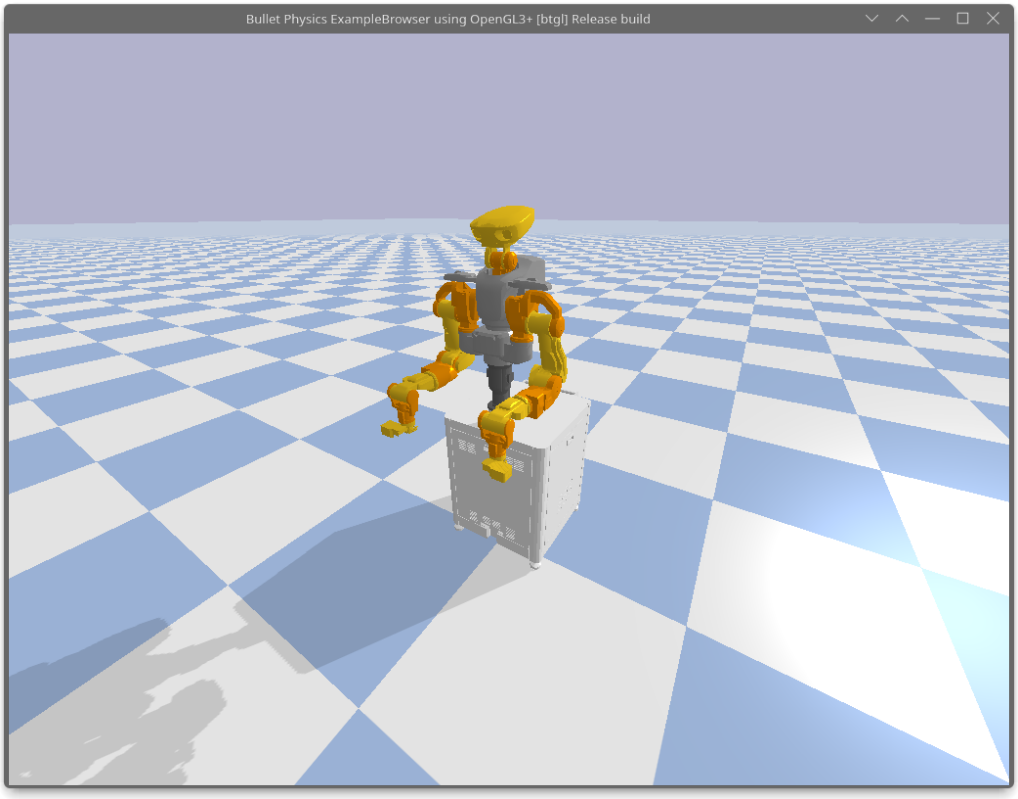
Line 39~44 (Picture 10) instantiates a *Simulation* object as *sim* and closes after 10 seconds or a time specified by the user.

```
39  sim = Simulation(pybulletConfigs, robotConfigs)
40
41  try:
42      time.sleep(float(sys.argv[1]))
43  except:
44      time.sleep(10)
```

Picture 10: Simulation object instantiation

Bash command:

$ **python3** hello_Nextage.py 5 # keep the window for 5 seconds
> ... # pybullet simulation starts (Picture 11)
> [Simulation] Found 17 DOFs
> ... # wait for 5 seconds
> [Simulation] Leaving



Picture 11: hello_Nextage.py with graphical interface

Note: Hold Ctrl + Left Mouse Button while moving the cursor to move the camera position.
Hold Ctrl + Middle Mouse Button while moving the cursor to move the camera target
Hold Ctrl + Right Mouse Button while moving the cursor to focus
Hold Ctrl while scrolling the Mouse Wheel to move along the camera direction

Class *Simulation_base*

This section is a brief overview of the class *Simulation_base*, please have a glance through the well-documented file *'core/Pybullet_Simulation_base.py'*.

The class *Simulation_base* is a wrapper class for the *Pybullet* simulation and provides you with some useful tools. In this practical, for simplicity and readability, we use **joint names** for indexing all joints & links in every task.

To learn more about *Pybullet*, please see Pybullet Quickstart Guide.

There are 5 groups of methods in *Simulation_base*:

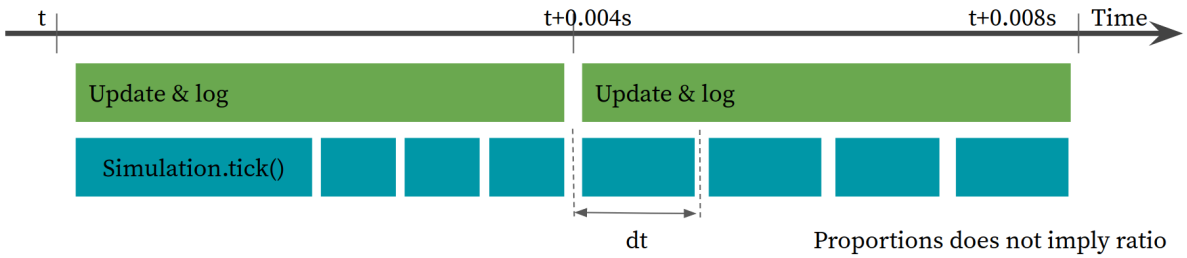
- **Constructor** - Set up the environment and robot
- **Destructor** - Elegantly exit the simulation
- **Set up tools** - Initialising tools
- **Robot config & status tools** - Robot information checkers
- **Robot dynamics and kinematics** (basic) - Basic tools for dynamics and kinematics

Class *Simulation*

This section is a brief overview of the interface class *Simulation*. The template for class *Simulation* is provided in the file *'core/Pybullet_Simulation_template.py'*. You are recommended to keep it unchanged while developing your solution with a copy of this file in the same directory and name it *'Pybullet_Simulation.py'*. Bash commands:

```
$ cd ./core # from the folder ./ARO_Practical_2022
$ cp Pybullet_Simulation_template.py Pybullet_Simulation.py
```

The class *Simulation* inherits from the parent class *Simulation_base* while adding more upon it. There are new skeleton methods, for each task, where you write your solutions. Most importantly, *Simulation* introduces method *tick()* which is critical to control your simulation. You should read carefully and make sure you have fully understood the method *Simulation.tick()* before the next step. Here is a hint for you: Picture 12:



Picture 12: Control frequency and Update frequency

From Picture 12: simulation events in the order of time, the update frequency and control frequency are by default, 250Hz and 1000Hz respectively. The green blocks denote the event of simulation update and logging which is triggered on the left edge of each block. Similarly, the blue blocks denote the event of *Simulation.tick()*. You are free to modify this method, more on this later in the tasks.

PD gains

Lastly, for the same reasons as creating file *'Pybullet_Simulation.py'* from its template, you shall create a file for saving the PD gains as well. Bash commands:

```
$ cd ./core # from the folder ./ARO_Practical_2022
$ cp PD_gains_template.yaml PD_gains.yaml
```

Now you have made a fresh copy of *'PD_gains.yaml'*. More on this later in the tasks.

Part one: Kinematics, Dynamics & Manipulation (25 points)

Introduction

In this section, we are going to implement two major components for the *Nextage* robot: PD controls and Inverse Kinematics solver. In order to achieve those, they are broken down into smaller tasks for you to experiment and develop.

For each task, there is a folder called *'taskx'*. Within each folder, there is a skeleton template named *'taskx_template.py'* for you. Please make a copy of that template and rename it as 'taskx.py'. Please only modify the copied code while leaving the others unmodified. Any modification in the library files will not affect the marking process, but make development harder for yourself and potentially affect the quality of your implementation. You are welcome to add any method you think is necessary within the file *'core/Pybullet_Simulation.py'*.

Point allocation

Part one: Dynamics and Kinematics - **25 points**

- Task 1: Forward and Inverse Kinematics - **10 points**
- Task 2: PD controller - **5 points**
- Task 3: Manipulation (Pushing, grasping & docking) - **10 points**
    - Task 3.1: Pushing the cube - **4 points,**
    - Task 3.2: Grasping & Docking- **6 points**

Part two: Practical Report - **15 points**

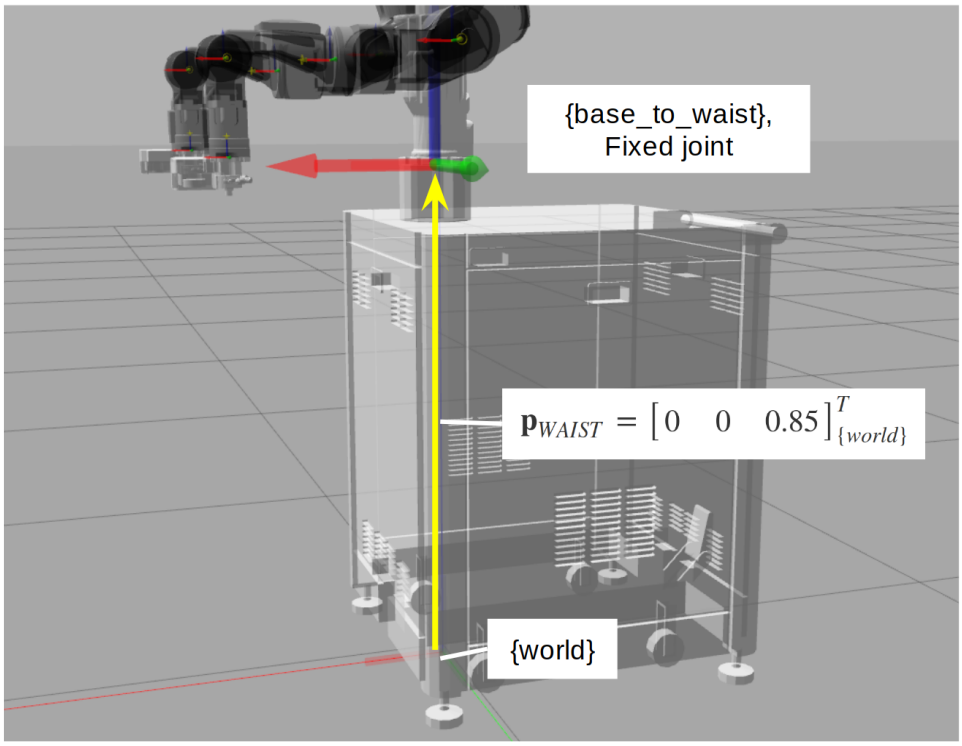
Total: 40 points (40%) of the overall mark of ARO

Task 1: Kinematics (10 points)

On the topology of the robot

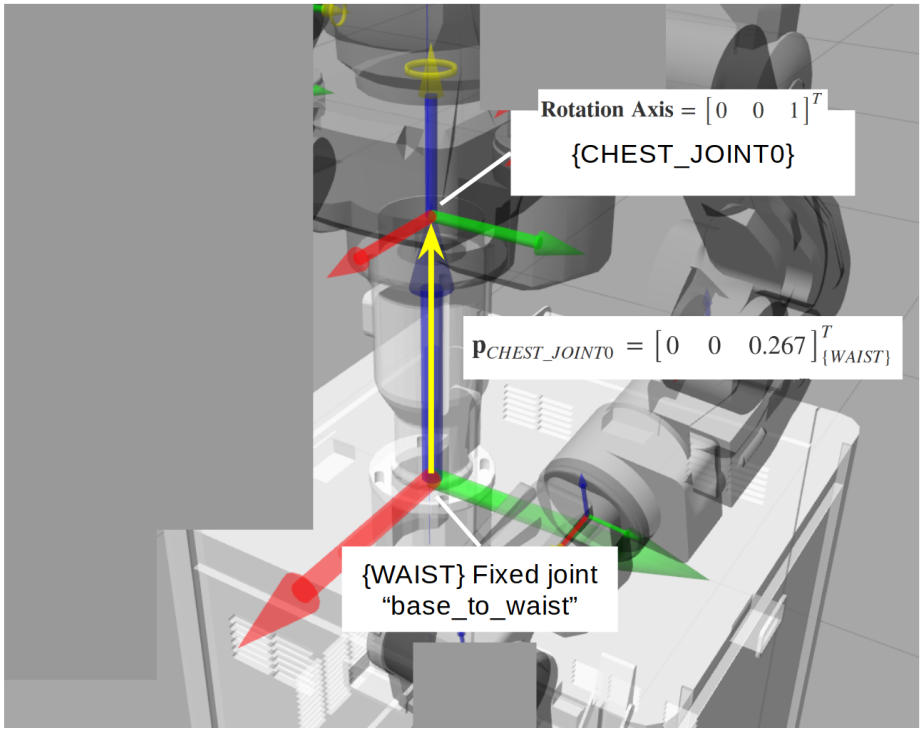
For our robot, the joints are all positioned at the origin with respect to their frames. Let's now have a look at the topology of the robot at its rest position. The rotation axis for each joint has a yellow cone tip surrounded by a yellow ring in the pictures. You should discover that all joint frames are defined with purely translation from the origin of the world frame, without any rotation.

World ➔ "base_to_waist"



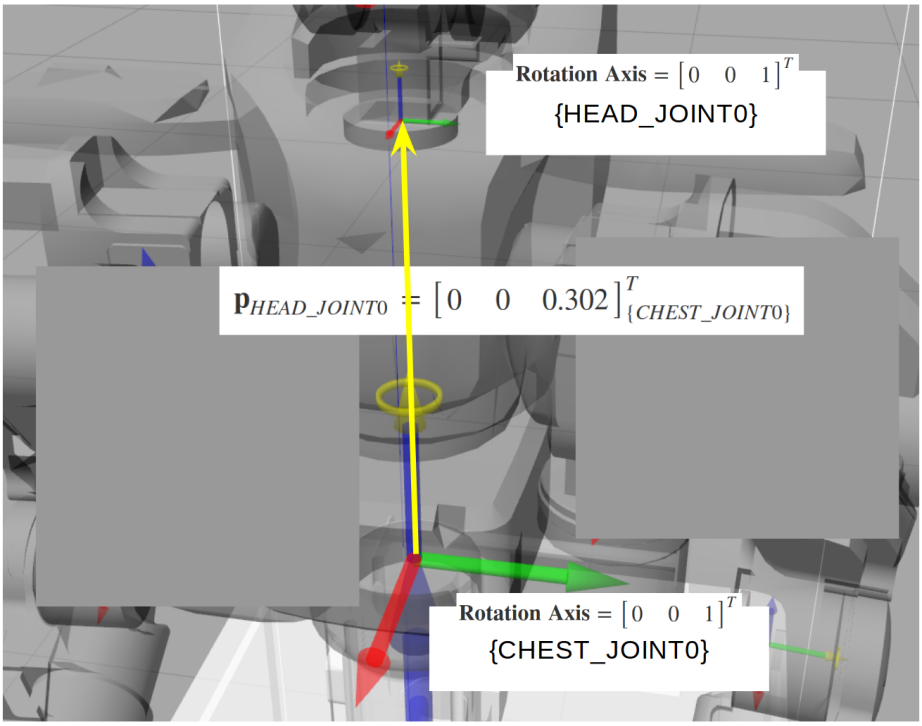
Picture 13: Transformation from world to WAIST

“base_to_waist” ➔ “CHEST_JOINT0”



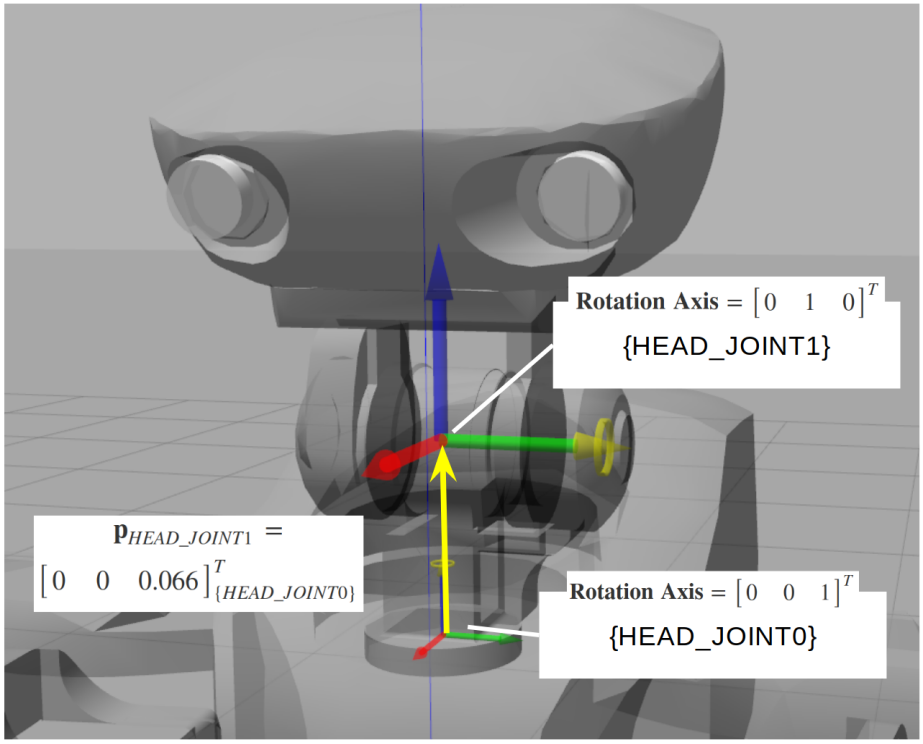
Picture 14: Transformation from “base_to_waist” to “CHEST_JOINT0”

“CHEST_JOINT0” ➔ “HEAD_JOINT0”



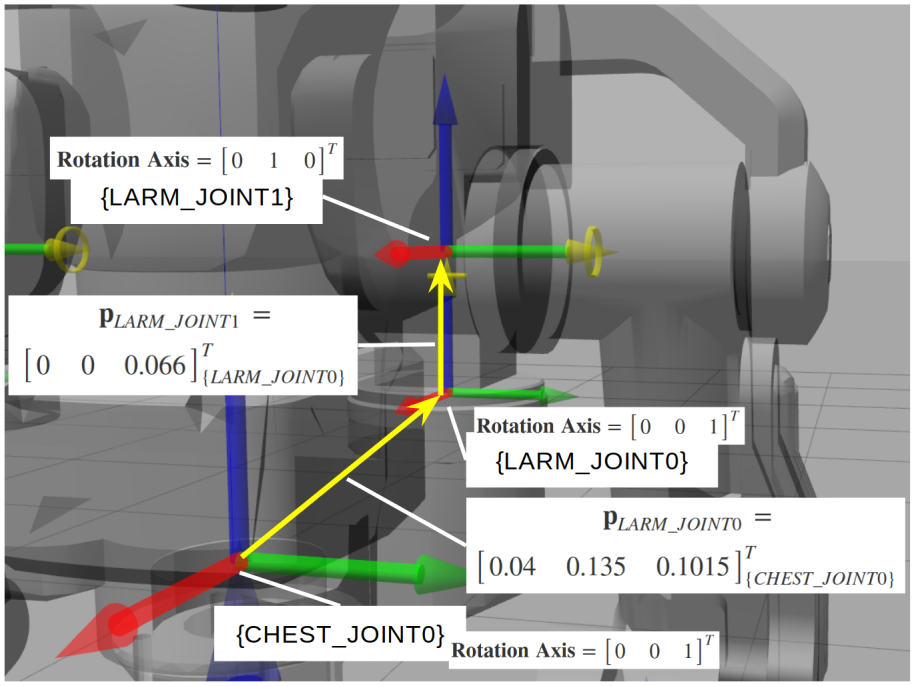
Picture 15: Transformation from chest to neck

"HEAD_JOINT0" → "HEAD_JOINT1"



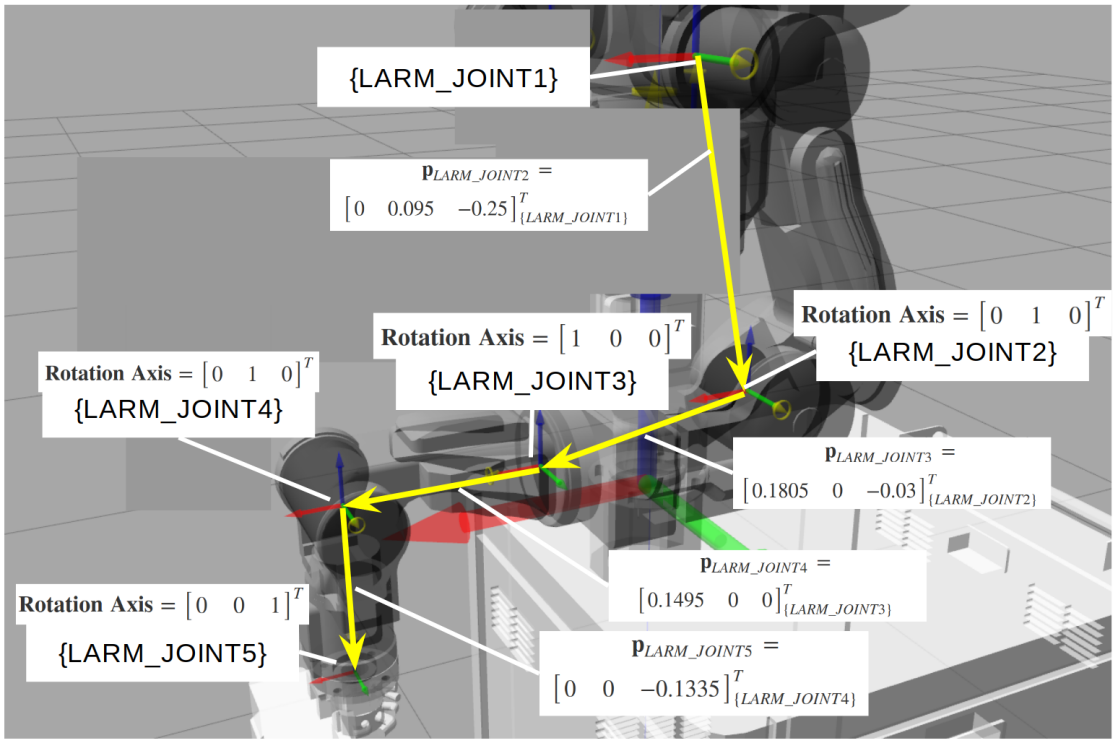
Picture 16: Transformation from neck to head

"CHEST_JOINT0" → "LARM_JOINT0" → "LARM_JOINT1"



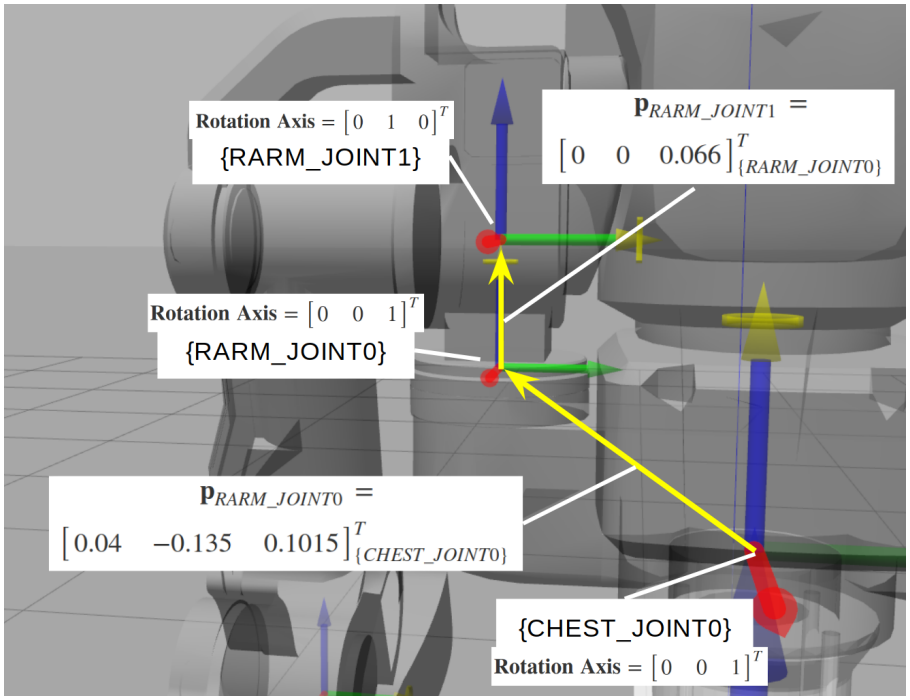
Picture 17: Transformation from chest to left arm

"LARM_JOINT1"➔ ... ➔ "LARM_JOINT5"



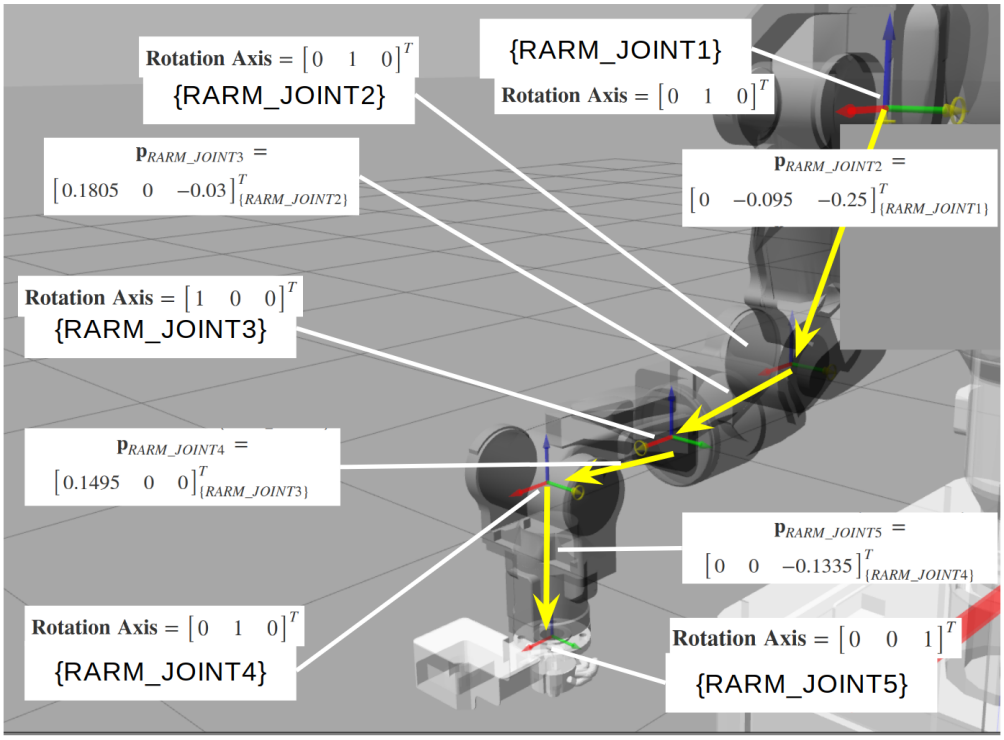
Picture 18: Transformation of the left arm

"CHEST_JOINT0" ➔ "RARM_JOINT0" ➔ "RARM_JOINT1"



Picture 19: Transformation from chest to right arm

"RARM_JOINT1"➔ ... ➔ "RARM_JOINT5"



Picture 20: Transformation of the right arm

Task 1.1: Forward Kinematics (5 points)

Task 1.1.1: Homogeneous Transformation Matrix

Forward kinematics (FK) calculates the position and orientation of every joint, link and the end-effector. First, let's calculate the homogeneous transformation matrices for each frame. Please check out variables:

- *Simulation.jointRotationAxis*
- *Simulation.frameTranslationFromParent*

In *Pybullet*, the pose of any object is represented with a position vector and a quaternion. You can convert between a quaternion, **q**, and the rotation matrix, R, by following methods: Picture 21. This is purely for debugging and experimenting, you do not need in-depth knowledge about quaternions to solve this practical.

$$\mathbf{q} = [q_x, q_y, q_z, q_w]$$

`Simulation.matrixToPose()`
`pybullet.getQuaternionFromEuler()`

`Simulation.getRotationMatrix()`
`Simulation.poseToMatrix()`
`pybullet.getEulerFromQuaternion()`

$$\mathbf{R} = \begin{bmatrix} 1-2s\left(q_y^2+q_z^2\right) & 2s\left(q_xq_y-q_zq_w\right) & 2s\left(q_xq_z+q_yq_w\right) \\ 2s\left(q_xq_y+q_zq_w\right) & 1-2s\left(q_x^2+q_z^2\right) & 2s\left(q_yq_z-q_xq_w\right) \\ 2s\left(q_xq_z-q_yq_w\right) & 2s\left(q_yq_z+q_xq_w\right) & 1-2s\left(q_x^2+q_y^2\right) \end{bmatrix}$$

Picture 21: Converting between quaternion and rotation matrix

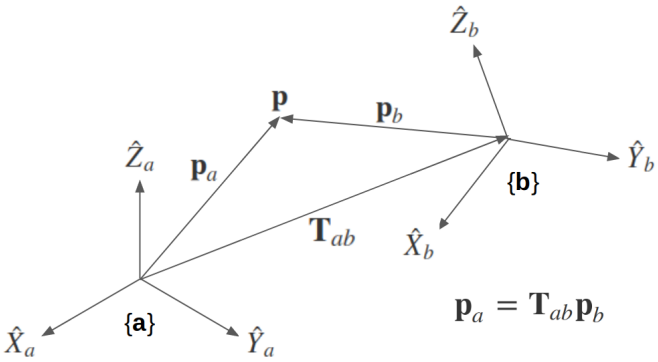
Rotational matrices about three axes following the right-hand rule:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The right-hand rule to obtain a rotation matrix from rotation matrices about the three axes as defined above is:

$$R = R_z R_y R_x$$

To transform a point from frame {**a**} to frame {**b**}, we can use the homogeneous transformation matrix **T**, Picture 22:



$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{p}_{translation} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Picture 22: (Left) Homogeneous transformation of point **p** from frame {**a**} to frame {**b**}
(Right) Homogeneous transformation matrix

You should now implement a method to calculate the rotational matrices and the transformation matrices for every single joint from the robot in *'core/Pybullet_Simulation.py'*. Note, you should sense

joint (revolutional) positions with method *Simulation.getTransformationMatrices()*. There is also an optional skeleton method *Simulation.getJointRotationalMatrix()* where you could calculate the rotational matrices all at once before calculating the transformation matrices. You could skip the optional skeleton as you wish.

Task 1.1.2: Kinematic Map

A kinematic map, ϕ, maps a robot configuration, q, to the position of the end-effector, y.

$$\phi : q \mapsto y$$

To get the (revolutional) joint positions (angles), you should use *Simulation.getJointPos()*. Please implement the method *Simulation.getJointLocationAndOrientation()* which takes the name of a joint and returns its pose in the world frame.

For avoiding unwanted exceptions during the simulation, please couple the waist joint and the base joint. Manipulating 'fixed' joints will crash the program, so let's avoid those joints from the beginning. (For more details, please see the robot URDF descriptions)

Task 1.2: Inverse Kinematics (5 points)

Task 1.2.1: Jacobian Matrix

$$J_{\text{pos}}(q) = \begin{pmatrix} [a_1 \times (p_{\text{eff}} - p_1)] & [a_2 \times (p_{\text{eff}} - p_2)] & \dots & [a_n \times (p_{\text{eff}} - p_n)] \end{pmatrix} \in \mathbb{R}^{3\times n}$$

Position Jacobian

$$J_{\text{vec}}(q) = \begin{pmatrix} [a_1 \times a_{\text{eff}}] & [a_2 \times a_{\text{eff}}] & \dots & [a_n \times a_{\text{eff}}] \end{pmatrix} \in \mathbb{R}^{3\times n}$$

Vector Jacobian

Let's calculate the Jacobian matrix for the robot. Please implement the method *Simulation.jacobianMatrix()*.

Task 1.2.2: Inverse Kinematics

Algorithm *Inverse Kinematics*:

```
function IK(endEffector, targetPosition, orientation, interpolation_steps, maxIterPerStep, threshold)
    curr_q = current joint positions
    starting_EFpos = FK(curr_q)[0]

    traj = [curr_q]
    for i in 1..interpolation_steps do
        curr_target = ith-interpolation point on the trajectory
        for iteration in 1..maxIterPerStep do
            ### TODO: find the desired displacement of the endEffector
            dy = # fix me!
            jacobian = jacobianMatrix(curr_target)
            ### TODO: calculate the joint increment using Jacobian matrix
            dTheta = # fix me!
            curr_q = curr_q + dTheta
            traj.append(curr_q)

            EF_position = FK(curr_q)[0]
            if | EF_position - curr_target | < threshold then
               break

    return traj
```

Please implement the method *Simulation.inverseKinematics()*. For more hints, please check out the kinematics tutorial on Learn. With your IK solver, now implement the method:

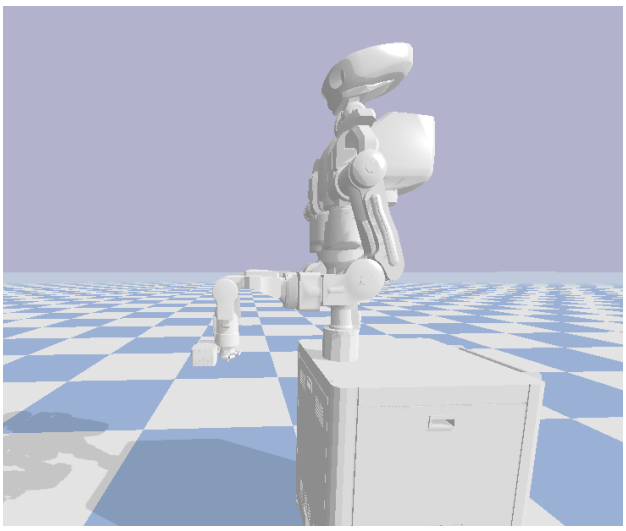
- *Simulation.move_without_PD()*

This method moves the end-effector to a target position. You should directly (re)set the joint positions to be the desired values using a method such as *Simulation.p.resetJointState()* or else.
For this task, only position control is required. For task 3.2, you are required to also add orientation control, so you can consider adding it now.

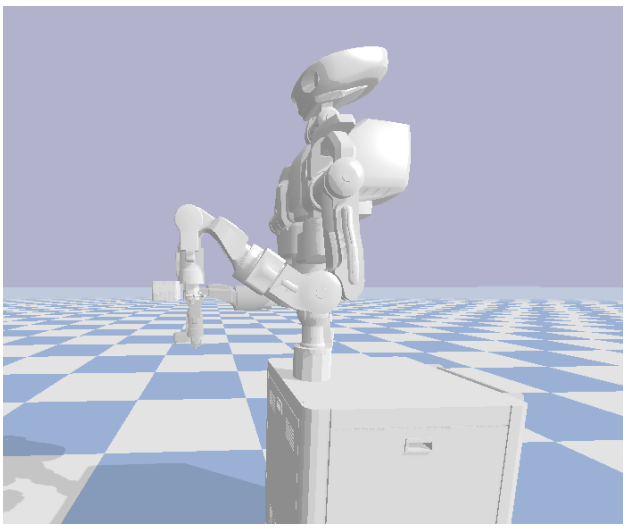
If you test your implementation of *move_without_PD()* using the values provided in the template code for Task 1,

```
# This is an example target position for the left end effector. This target
# position assumes your world frame is located at the base. If your world
# frame is located at the waist, you will need to transform this vector using
# the base_to_waist translation.
endEffector = "LARM_JOINT5"
targetPosition = np.array([0.37, 0.23, 1.06385])  # x,y,z coordinates in world frame
```

Then the final result should look like this (**note**: assuming your world frame is located at the base):



The initial position



The final position after moving LARM_JOINT5 to position [0.37, 0.23, 1.06].

Task 1.3: The Nextage Robot Simulation

Now, you have built all the core functionalities to start your simulation. However, the “brain” of the robot is still not connected to the manipulators. The last implementation, fix the method *Simulation.tick_without_PD()*. You should send the control signals to the manipulators using *Simulation.tick_without_PD()*.

Checklist for task1

You should have implemented / modified:

- *core/Pybullet_Simulation.py:jointRotationAxis*
- *core/Pybullet_Simulation.py:frameTranslationFromParent*
- *core/Pybullet_Simulation.py:getJointRotationalMatrix()* - optional
- *core/Pybullet_Simulation.py:getTransformationMatrices()*
- *core/Pybullet_Simulation.py:getJointLocationAndOrientation()*
- *core/Pybullet_Simulation.py:jacobianMatrix()*
- *core/Pybullet_Simulation.py:inverseKinematics()*
- *core/Pybullet_Simulation.py:move_without_PD()*
- *core/Pybullet_Simulation.py:tick_without_PD()*
- *task1/task1.py*

Task 2: Dynamics (5 points)

Task 2.1: PD controller

You are going to implement a closed-loop controller to manipulate every single joint of the robot. Your controller should take the real and the target joint positions and joint velocities to generate a torque, in Newtons. You should use your knowledge from the lectures to tune the Kp, Ki and Kd parameters which yield the best performance. Here is the equation of the dynamic system:

$$\boldsymbol{u}(t) = k_p(x_{ref} - x(t)) - k_d\dot{x}(t)$$

Where,

- t is time
- k_p, k_d are PD gains, respectively
- $x(t)$ is the current position
- $\boldsymbol{u}(t)$ is the output of your control

You should implement your code in *'core/Pybullet_Simulation.py:calculateTorque()'*. The PD gains should be tuned and saved to file *'core/PD_gains.yaml'*.
For more guidance, please see the course slides and the tutorials on Learn.

Optional task: Gravity compensation (no reduction in the final marks for skipping this task)

From the class Simulation, we have provided a naive gravity compensation, which is applying a vertical force at CoM of each joint. This is good enough for this practical, however, as you could imagine, it is not applicable in reality. If you are interested in designing a complete PD control with gravity compensation, have a look at this paper.

Task 2.2: Manipulate a joint

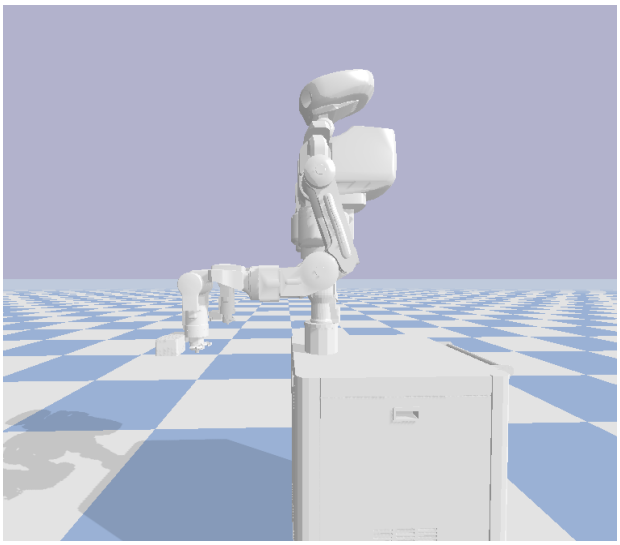
You should have developed your PD controller in Task 2.1. Now, let's manipulate a joint with it. Here are some methods you may find useful:

- getJointPos / getJointPoses - sense the joint revolutional position
- getJointVel / getjointVelArr - retrieve the joint velocity (Using this will result in a reduction of marks)

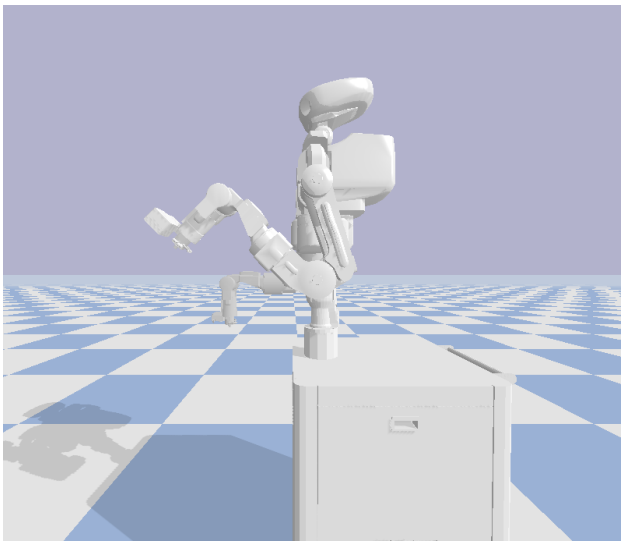
You should implement your code to *Simulation.moveJoint()*. You could run *'task2/task2.py'* to debug and test your code and plot a graph for the report. In your report, you should explain the graph and identify the flaws in the current controller.

You can test and tune your PD controller by running the template code provided for Task 2. For example, the following settings set a desired angular position of [-45 degrees] for "LARM_JOINT2" and the screenshots below show the initial and final arm configuration when using these settings:

```python
# This is an example target (angular) position for the joint LARM_JOINT2
task2_jointName = "LARM_JOINT2"
task2_targetPosition = np.deg2rad(-45)  # joint (angular) position in radians
task2_targetVelocity = 0.0  # joint (angular) velocity in radians per second
```



The initial position



The final position after moving LARM_JOINT2 by [-45 degrees].

You can use this template code and modify the name of the joint you are using to tune the PD gains of each joint individually. You are encouraged to produce plots to assist you in the tuning process. Notice how the *'moveJoint()'* method only moves a single joint and keeps all others fixed, which is good for tuning the PD gains.

Within the template method *'moveJoint()'*, there is a simplified version of *'tick()'* called *'toy_tick()'*. The more complicated method *'tick()'* will be used later to manipulate every joint of the robot at once. But before that, *'toy_tick()'* could give you a chance to get familiar with the workflow of *Pybullet* simulation.

Once you have solved Task 2 and tuned the PD gains for all joints, please reuse the line from *'toytick()'* which calculates the torque using the PD controller and paste it in the method '*tick()'*. This will allow the method *'tick()'* to perform PD control for all joints.

Checklist for task2

You should have implemented / modified:

- *core/Pybullet_Simulation.py:calculateTorque()*

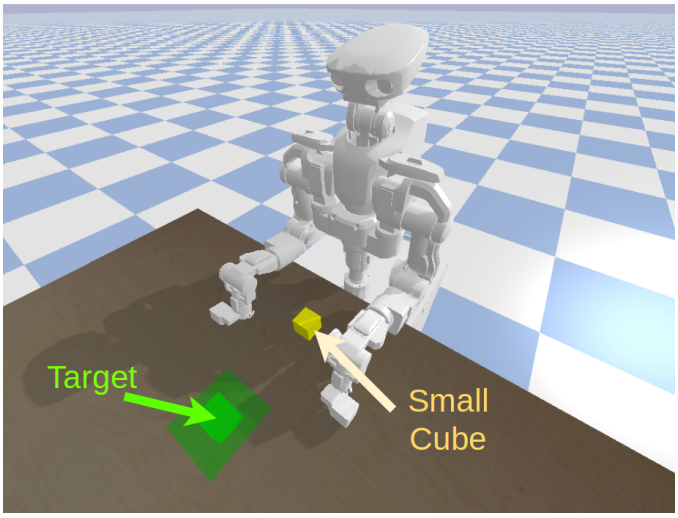
- *core/Pybullet_Simulation.py:moveJoint()*
- *core/PD_gains.yaml*
- *core/Pybullet_Simulation.py:tick()*
- *task2/task2.py*

Task 3: Nextage Robot Manipulation (10 points)

As a recommended (but not strictly necessary) step before approaching the following tasks, you are encouraged to develop a function which performs polynomial interpolation between a set of given points, as this will be a valuable tool to generate the trajectories to solve the following tasks (particularly Task 3.2). A recommendation is to use a **cubic** spline (i.e. 3rd order polynomial), with natural boundary conditions at the end points (i.e. the second derivative is zero). You are allowed to use existing methods found in *scipy.interpolate* to sample points to generate the trajectory.

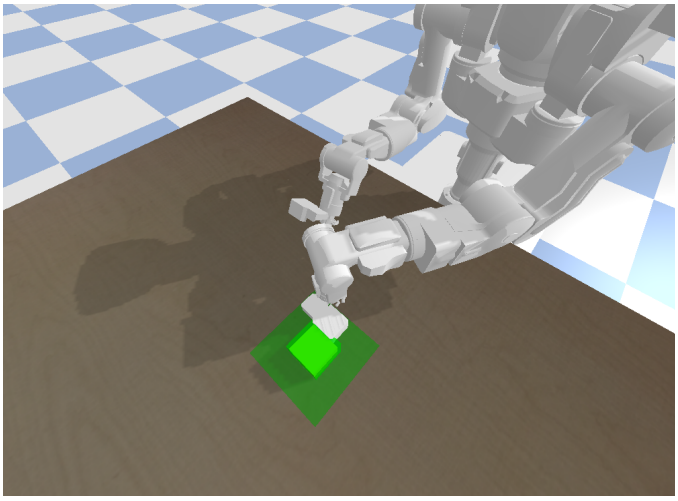
Task 3.1: Pushing a cube (4 points)

Objective: pushing a cube to a target location
In this sub task, there is a table in front of the robot and there is a small cube on the table. What you should do is to push that cube to a target position in green colour (Picture 23).



Picture 23: pushing the cube setup

You are free to move any joint of the robot pushing the cube to anywhere you intended to. You can use the left arm or right arm or even both arms. The quality of the task is purely judged on the final distance from the centre-of-mass of the cube to the target centre. An example shown below in Picture 24.

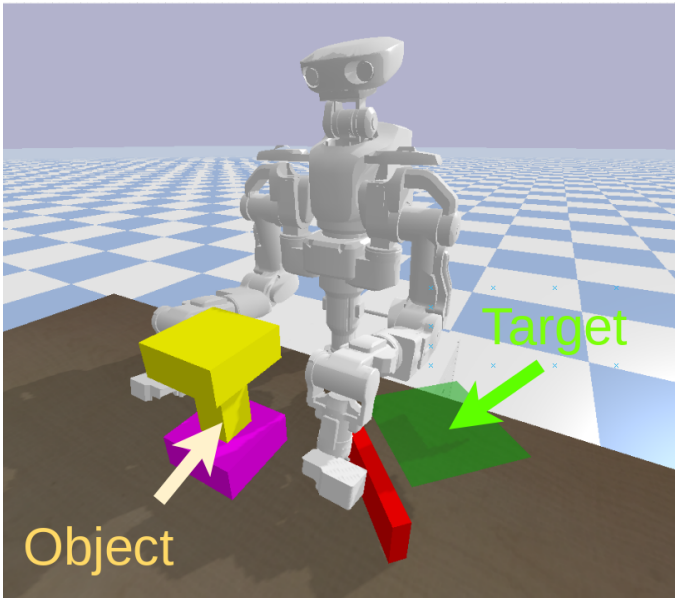


Picture 24: the cube is pushed to the target position

There is a template method *Simulation.moveEndEffectorToPosition()* where you could add your code for this task. But you are free to add other methods you need.

Task 3.2: Grasping & Docking (6 points)

Objective: grasp and dock an object to a target.
For this task, you are required to add orientation control for the end-effector.
Extending from task 3.1, this time, you are asked to dock an object to a target with some obstacle in between. So, you probably will have to pick it up in some way. We have changed the cube to a "dumb bell" for you to make the task easier, because the robot has only one finger per hand. Here is an example of setup. (Picture 25)



Picture 25: docking setup

There is a template method *Simulation.dockingToPosition()* where you could add your code for this task. But you are free to add other methods you need.

Checklist for task3

You should have implemented / modified:

- *core/Pybullet_Simulation.py:moveEndEffectorToPosition()*
- *core/Pybullet_Simulation.py:dockingToPosition()*
- *task3_1/task3_1_pushing.py:solution()*
- *task3_2/task3_2_docking.py:solution()*

Precision requirements:

This project will not ask you to beat the industrial standards, but please push yourself to the limit (and maybe break through it).

PD error:

- <= 0.035 rads (about 2 degrees) for full marks
- > 0.035 rads reduction of 1 mark
- Note, a higher precision will make task 3 easier.

Pushing/docking error in Task 3: (distance from CoM of the cude/dumbbell to their targets)

- < 150mm for the full marks
- >= 300mm for 0 marks
- in between - uniform reduction

After you have finished all tasks

- Check the prohibited APIs in your solutions
- Validate your work with non-GUI simulation (**!Important!**)
- Sanitise and review your code while adding comments

Part two: Practical Report (15 points)

Due: (report and code) on **Friday 25 Nov, 12:00**. Hard deadline.

In the report, you should explain what you have learnt from the course that is related to this practical. For each task, you should explain and demonstrate your algorithm(or system schematics) in good detail, e.g. with pseudo code or block diagram. List the mathematical equations you used in the computation. List all the parameters you used. The completeness and performance of your implementation in each task should be evaluated with at least one metric. For example, you can use the error between target end-effector pose and real-time end-effector pose to validate your IK solver. Make sure that a person with no robotics background can understand your report.

Remember to export your report as a **.pdf** file.

Advice on writing the lab report

The final report will be written individually and it will contain a complete description of the algorithms developed during the course, e.g. introduction, problem description, evaluation, conclusion.
Note that the write-up of the final report is an equivalence (short) of how robot projects are reported in scientific papers. Here's an outline of what your report should contain:

Title: A 4-12 word title that would allow an unfamiliar reader to know what your report is about.

Abstract: You MUST preface the report with a 100-200 word summary of what it contains. This is usually easier to write when you have finished the report. It should briefly explain the task, the approach used, the results and the conclusions drawn. Avoid making entirely generic statements that could apply to almost anything, e.g., (BAD) "This report describes the construction of a robot to perform a task. We describe the design decisions and outline the control program, then explain the results and possible improvements". Instead, make it specific to what you have done, e.g., (GOOD) "We have developed the robot capable of grasping objects in a confined environment. It uses trajectory planning in the task space to avoid obstacles, and inverse kinematics to resolve the joint angles given the Cartesian space trajectories. We implement a PD control architecture in conjunction with the inverse kinematics to close the loop. The robot was tested in five different trials and was able to dock objects to 3 target locations.".

Introduction: This should explain the task, and give an overview of how you approached it. In a normal scientific report, this would include the reference to previous work (your own or others). You are not required in this case to refer to other work (although you may wish to do so if for example something you read about influenced your approach to the task). So this section is likely to be quite short (400 words).

Methods: A good rule of thumb here is that someone reading your report should be able to replicate your approach. So you need to provide a good description of the physical architecture, particularly

the type, number and position of sensors and actuators. Include labelled photographs or diagrams, and make sure the dimensions are clear. Comment on factors that led to the design, explaining the decisions you made. For the control program, you should provide a flow diagram or pseudo-code description, and again explain the reasoning that led to this solution. This is likely to be the longest section of the report. Do not include code except for short snippets that help explain a crucial part of the program you created. Avoid repetition and refer to other peoples' work instead of describing well-known algorithms. (1400 words)

Results: This section contains the results of 3 tasks, therefore, you can present results by sub-sections as: Task 1, Task 2, and Task 3, with data plots, snapshots of simulations, and descriptions.
This should contain some quantitative evaluation of the robot performance. For example: that it can find a resource site from a distance of x metres, and recognise and leave within t seconds; etc. If your robot is not capable of doing the final task, you should evaluate what it does correctly, and try to analyse what it does wrong. The reader should be left with an accurate understanding of exactly what your robot is capable of, even if this is not as good as you hoped. Bad results are results too. You get marked based on how you approached the problem and how you evaluated the results. (800 words)

Discussion: Start by summarising the results, and giving your evaluation of how well it works. Explain what are the most successful elements of your approach, and what was the limitations. Include ideas about how the system could be improved. (200 words)

Length: The final report should be no more than 3000 words long. It can be shorter if you think that you can do a satisfactory description in fewer words.
Note: please use the snapshots of your simulation (simple style) to aid the explanation and elaboration of your methods and results. Please pre-record the demonstration of your task completion and include the **links** to your videos in the report.

Files to submit

Please submit the relevant files only and make sure the Python scripts run out-of-box. In more words, please include 1 experiment (.py script) per task, and also remove all the other irrelevant files such as pictures, to reduce file size (keep ~5MB).

```
./ARO_Practical_2022
├── core
│   ├── nextagea_description
│   │   └── Include all files as provided
│   ├── Pybullet_Simulation_base.py
│   ├── Pybullet_Simulation.py
│   └── PD_gains.yaml
├── task1
│   └── task1_solution.py
├── task2
│   └── task2_solution.py
├── task3_1
│   ├── lib
│   │   └── Include all files as provided
│   └── task3_1_solution.py
├── task3_2
│   ├── lib
│   │   └── Include all files as provided
│   └── task3_2_solution.py
└── report.pdf
```

An example of files to submit, the **files** contain your solution

To make a .zip file, you could use:

```
$ zip -r <YOUR_UUN>.zip ARO_Practical_2022
```

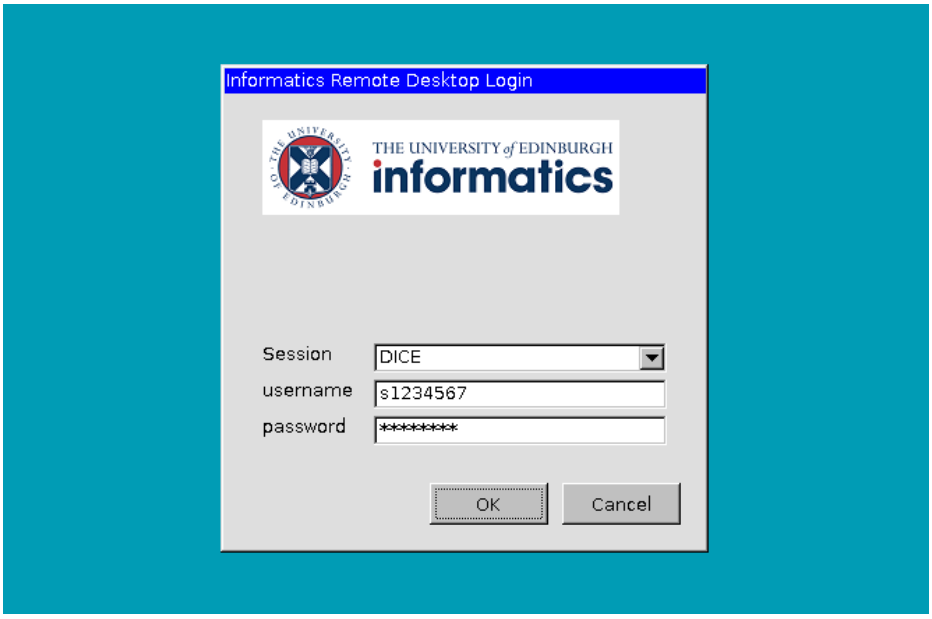
Submit via
Learn > Assessment > Assignment Submission > "Coursework 2 and Practical Test: report and python code"

Appendix 2: Remote DICE with GUI

To establish a connection to remote DICE with GUI, you can follow the instructions from Informatics remote DICE. It works for all platforms (Windows, MacOS and Linux).

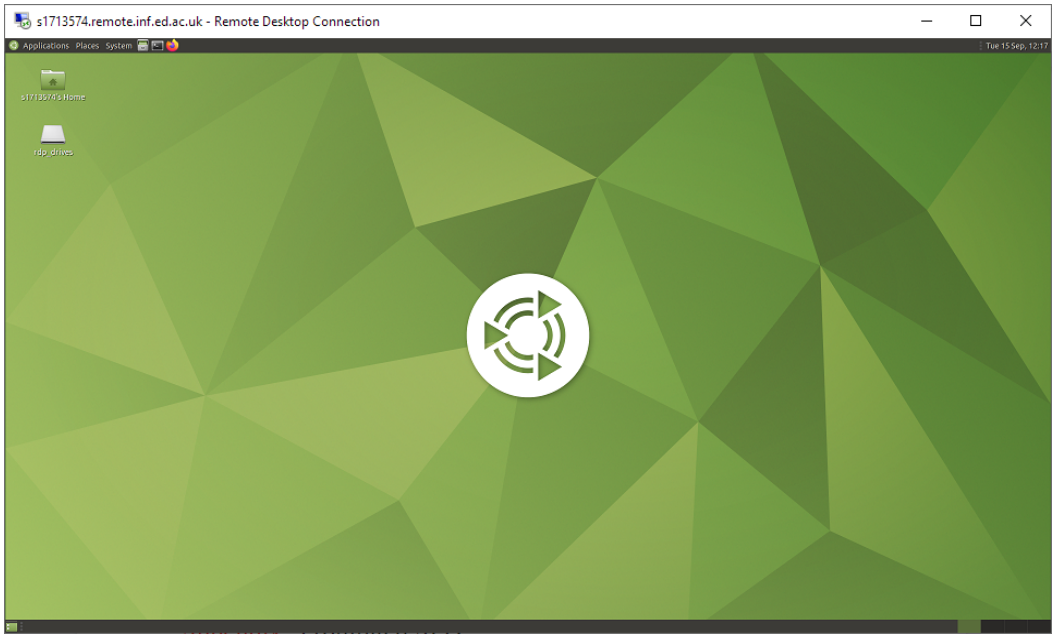
	Format	**Example**
Server Address	<YOUR_UUN>.remote.inf.ed.ac.uk	s1234567.remote.inf.ed.ac.uk
Username	<YOUR_UUN>	s1234567
Password	your DICE password (May be different from your MyEd/Learn/UoE Windows 10 passwords)	

Once you have correctly connected to remote DICE, you should see a login page: (Appendix Picture 1)



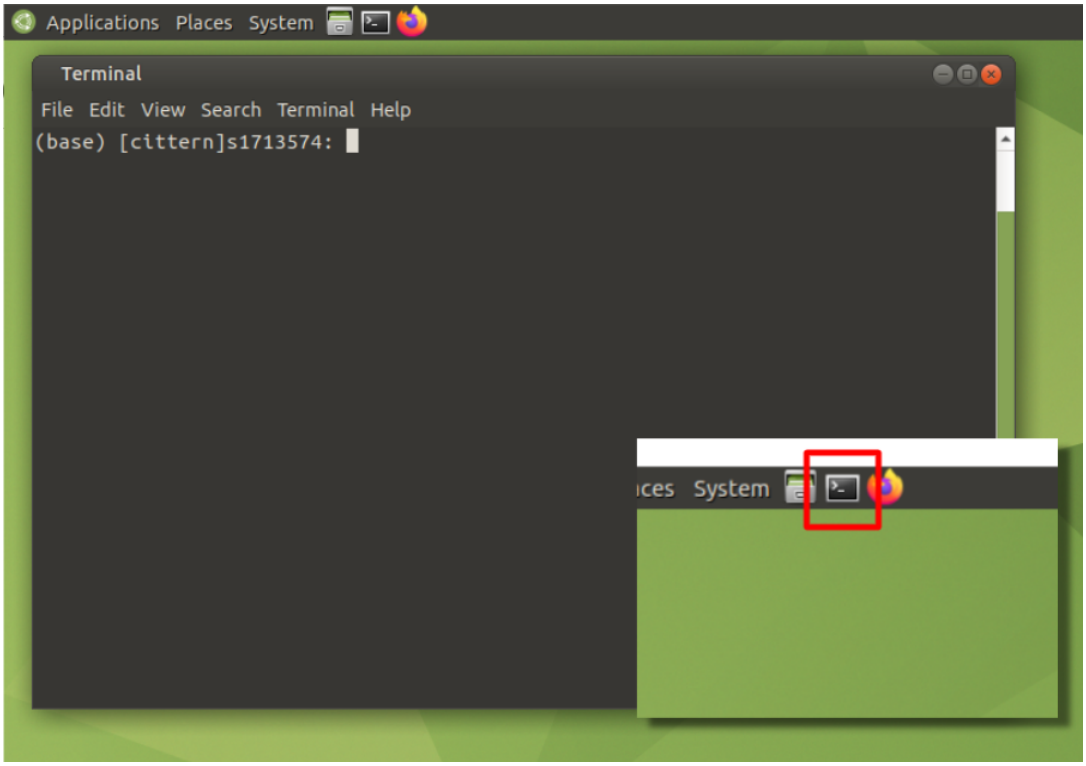
Appendix Picture 1: Remote DICE login page

Enter your username and password and click the OK button. You should see a similar screen (Appendix Picture 2) if you have successfully connected to remote DICE:



Appendix Picture 2: Example of remote DICE

Open a terminal by clicking the icon on the toolbar (Appendix Picture 3):



Appendix Picture 3: Open terminal on DICE

Now you can follow the instructions on the Git repo. Or submit your solution.

Appendix 3: Prohibited APIs

The class *Simulation* provides a set of friendly APIs for those who are not familiar with *Pybullet*. Those methods should be enough for you to achieve everything practical. Still, you are welcome to **learn and debug** with the help of built-in APIs from *Pybullet*. However, some APIs are "overpowered" for this practical, therefore, directly or indirectly utilising them in your solution is regarded as **CHEATING!** All solutions will be marked by hand, any activity in any of these categories are identified as cheating:

- Any algorithm/library that solves dynamics & kinematics problems
- Any algorithm/library that manipulates the robot components "bypassing" physics. i.e. direct joint/link position update.

Prohibited APIs from *Simulation* class:

- setJoints
- getJointState
- getJointVel
- getJointVelArr
- getLinkState
- getLinkDynamicsInfo

Prohibited APIs from *Pybullet* library

- restoreState
- getBasePositionAndOrientation
- resetBasePositionAndOrientation
- getJointState(s), resetJointState
- getLinkState(s)
- getBaseVelocity, resetBaseVelocity
- applyExternalForce/Torque
- getDynamicsInfo/changeDynamics
- calculateJacobian
- calculateInverseDynamics(2)
- calculateInverseKinematics(2)

The **ONLY EXCEPTION** to the above list is **Task 1**, where you are allowed to use the following method when implementing FK and IK: resetJointState().

To learn more about *Pybullet*, please see Pybullet Quickstart Guide