Computer Exercise 2

In this exercise you will implement forward and inverse kinematics solvers for a robotic manipulator. This exercise's workflow is organized as follows:

- We will start by setting up a new ROS workspace.
- We will visualize the robot, understand the ROS nodes involved, and investigate the robot structure via URDF files.
- We will create our own methods for forward kinematics (FK) and inverse kinematics (IK) by applying the algorithms seen in class.

General instructions:

- Submission is in pairs only.
- Submission deadline: 16/12/2021, 23:59
- Questions related to the exercise should be posted in the "Discussion on Computer Exercises" forum on Moodle.
- Submit a zip file containing the code and a written report. The code should include the *launch*, *scripts*, and *URDF* directories under the *hw2* package. The written report should include figures as required in the questions.

Part I: Workspace Setup

Create a new workspace and name it *hw2_ws*:

```
mkdir -p ~/hw2_ws/src
cd ~/hw2_ws/src
catkin_init_workspace
```

Now, paste the contents of the (unpacked) zip file (provided with this exercise) into the folder *~/hw2_ws/src*.

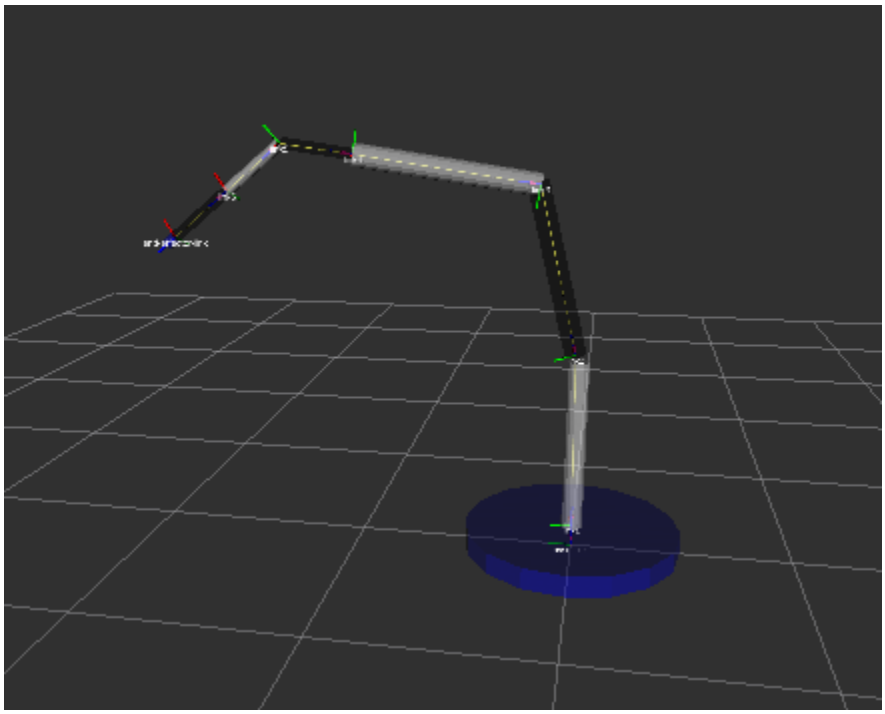
Then, run:

```
catkin_make
source ~/hw2_ws/devel/setup.bash
```

Don't forget to source the new workspace in new every terminal that you open.

Part II: Visualizing and Moving the Robot

Our robotic arm (see figure below) consists of a revolute-revolute-revolute wrist on top of an anthropomorphic arm.



1. Run Rviz by

```
roslaunch hw2 display.launch
```

   Note that you can change the robot's joint configuration with the sliders. Try playing with the *Randomize* and *Center* buttons under the *joints state publisher* UI window. Attach a screen capture of the robot in a few random configurations of the joints.
2. Describe the configuration space of the robot: what is the number of joints and their types?
3. Identify all the reference frames visible in RViz. Each frame is defined by its x, y and z axes, and their respective colors are red, green, and blue. The "center configuration" will denote the robot's configuration where all joint angles are zero.
   Assume the robot is posed in its center configuration. What is the rotation between *base_link* and *end-effector-link*? Reading from RViz is sufficient, code is not required.

In addition to the joint state publisher UI window, joint commands can be sent by code. The code in *fixed_joints_publisher.py* does exactly that – try running this node as you may need to use it in later clauses. The code is provided for your convenience – no action needed from your side.

4. Open in a text editor the file *robot_hw2.urdf* (under *src/hw2/urdf*). Make sure you understand the robot description.

For this clause only, create a copy of *robot_hw2.urdf*. Connect a sphere of radius 1 to the end-effector such that it is touching the end-effector link in a single point. What is the joint type you used? Provide both the modified URDF file, and a screen shot of the new robot in RViz.
Note: revert to the original URDF for the rest of this exercise.

Part III: Denavit-Hartenberg

1. Mark the frame axes on the robot joints in a diagram according to the DH conventions. In your diagram, align your first frame with the base link frame and your last frame with end-effector-link frame. Draw the diagram for the robot at its center configuration and recall that at this position all joint angles are zero (i.e., $q_i = 0\ \forall i$). Make sure that the z axes keep the rotation in the same direction as indicated in the URDF file (verify that positive rotation in the diagram corresponds to positive rotation in RViz).
2. Let *j1* be the configuration of joints where each joint is set to 90°. Provide a screen capture of RViz at *j1*. Draw the DH diagram again, this time for the *j1* configuration.
3. Write the DH parameter table. Notice that to obtain the translation components between consecutive DH frames, you will need to extract link radii, lengths, and origins from the robot's URDF.
4. What are T_0^B and T_{EE}^n in this case?
5. *tf* is a ROS package that reads the robot URDF and handles the computation of link poses as well as transformations between different frames. See also the tf tutorials.
   (a) Set the robot to the center position. Use the *tf_echo* command to output the pose of the end-effector-link origin relative to the base_link frame. Report the command, the result, and the axis-angle representation of the quaternion.
   Note: in class we defined quaternions to have a positive value in the first (real) component.
   (b) Repeat the previous clause for the robot is in *j1*.

   From now on, when you implement a forward kinematics solver, you can compare your results to *tf*.

Part IV: Forward Kinematics

You will now implement a forward kinematics solver using the provided skeleton in *hw2_services.py*.

1. First, let's look at the node initialization in the *__main__* part. For now, keep the line

```
# solve ik(gs)
```

in comment as we will only need it in a later clause.
Use the member variable *current_joints* (in *GeometricServices*) to save the joint states by subscribing to the *joint_states* topic. Complete *TODO1* in the code (2 occurrences).

2. Create a service named *get_tf_ee* that when called, reads the value of *current_joints* and by using *tf*, returns the pose of the end-effector. You should use *tf.TransformListener* as described in the [Writing a tf listener](#) tutorial. Complete *TODO2* in the code (3 occurrences).
   What are the returned values for *j1*?
3. We will now implement a forward kinematics service named *get_ee_pose* and verify its result against the result of the *get_tf_ee* service. For this clause, use *numpy* for the numerical computations.
   (a) Start by populating the DH parameters in *TODO3*.
   (b) Implement the *_generate_homogeneous_transformation* static method that returns a two-dimensional *numpy* array representing a single joint transform (*TODO4*).
   (c) Since *tf* represents orientation as quaternions, we need to output quaternions as well. Implement the static method *_rotation_to_quaternion* that outputs a quaternion representation based of a rotation matrix *r* (*TODO5*).
   (d) The service *get_ee_pose* is already initialized in the skeleton code. You are required to implement its core logic in the method *get_ee_pose* in *TODO6*. This method gets the joint configuration and should use the *_generate_homogeneous_transformation* and *_rotation_to_quaternion* methods to return the transformation matrix, translation, and quaternion representation of the end-effector for these joints.
4. Now that you have completed writing the forward kinematics solver, verify that your result for joints configuration *j1* matches the results of *tf*.
   You should call the new ROS service you created and report the resulting message in the written report.

Part V: Inverse Kinematics

Now that we have computed the transformation matrices, we are ready to implement an inverse kinematic solver. Our solver will use ZYZ Euler angles as orientation representation for the poses. In order to enable the inverse kinematic code, please uncomment the third line in *__main__*:

```
solve_ik(gs)
```

Our goal is to find joint configurations that set the end-effector to position

$$p = (-0.770;\ 1.562;\ 1.050)$$

and orientation (quaternion)

$$q = (0.392;\ 0.830;\ 0.337;\ -0.207)$$

1. First, we need to express the required end-effector orientation as ZYZ Euler angles. Complete the implementation of *convert_quaternion_to_zyz* in *TODO7*. Report the ZYZ Euler angles for

$$q = (0.392;\ 0.830;\ 0.337;\ -0.207)$$

2. We will now implement the inverse kinematics solver in three steps:
   (a) Implement the calculation of geometric Jacobian in *get_geometric_jacobian* (*TODO8*). You may find it helpful to reuse *_generate_homogeneous_transformation*.

(b) Complete the computation of analytical Jacobian with ZYZ Eular orientations in the *get analytical_jacobian* method (*TODO9*).

(c) Finally, use the analytical Jacobian to implement the *compute_inverse_kinematics* method (*TODO10*). The parameters for this method are as follows (initial values are given in the *solve_ik* method):

- *end_pose*: the required pose of the end effector.
- *max_iterations*: an iteration count that limits your algorithm from running forever. You may adjust this parameter freely or ignore it depending on your implementation (this parameter is given as a recommendation only).
- *error_threshold*: the required mean squared error threshold between the pose of your solution and end pose.
- *time_step*: Δt that should be used by the solver. You may adjust this parameter freely or ignore it depending on your implementation (this parameter is given as a recommendation only).
- *initial_joints*: the initial joints to search from. We start from a value of 0.1 in every joint.
- k: damping factor (see Lecture 5). You may adjust this parameter freely or ignore it depending on your implementation (this parameter is given as a recommendation only).

Note: values for joints should always be between the joint limits (as defined in the URDF). You are free to implement any of the iterative algorithms shown in Lecture 5 (or any combination of the two).

For this part, provide:

- A plot showing the difference in x, y and z displacement over the iterations of the algorithm. Draw one figure with a different colored line plot for each of the 3 coordinates.
- A plot showing the difference in the ZYZ angles ϕ, ν, ψ) over the iterations of the algorithm. Draw one figure with a different colored line plot for each of the 3 angles.
- Your final position and orientation errors.

Note: your solution should have a squared error less than 0.001 as indicated by the *error_threshold* parameter.

Hint: make sure to verify your IK solution by either running it in your FK solver, or by setting the robot to the solution and querying TF for the pose.

Before you start the next part, please make sure to run the following line to install the necessary packages.

```
sudo apt-get install ros-melodic-effort-controllers
```

Also, you need to download the additional zip file from moodle. After extracting the contents, copy and paste all of them into the `rrbot_ws/src` directory. To finish off, run the following line:

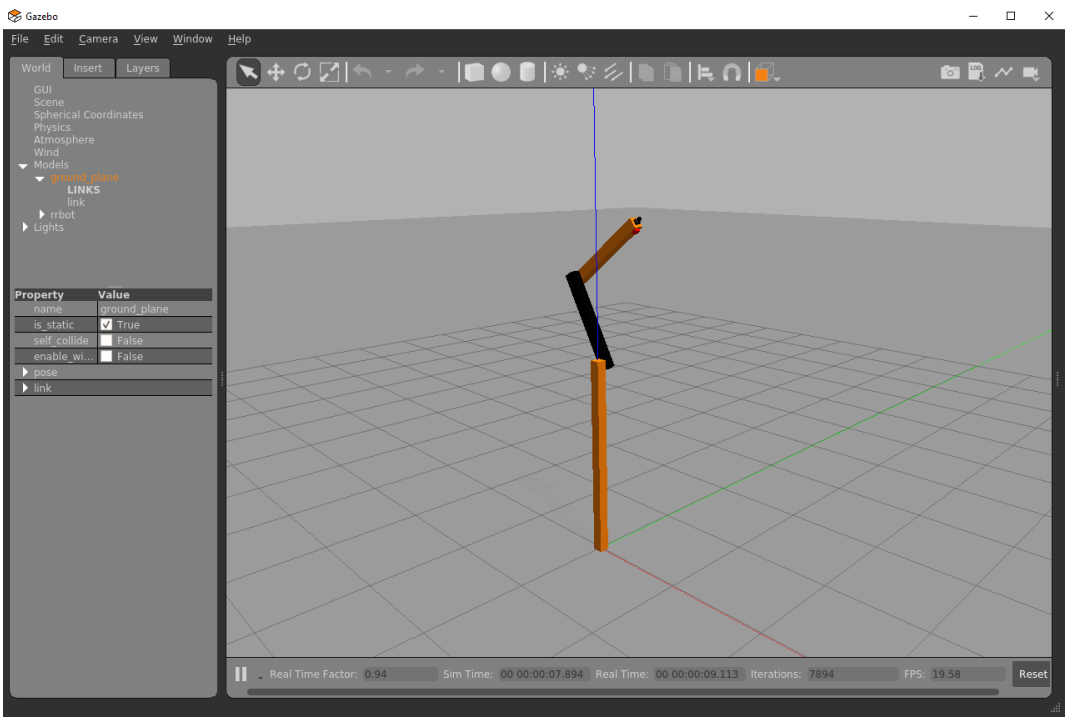
```
cd ~/rrbot_ws && catkin_make
```

Part VI: Get to Know RR-Bot

The working environment for this exercise is Gazebo. Start by launching the old-new RR-Bot:

First, close all your previous terminals and open a new one. Launch Gazebo by executing

```
source ~/rrbot_ws/devel/setup.bash
roslaunch rrbot_gazebo rrbot_world.launch
```



You should already be familiar with this simple revolute-revolute manipulator. Refresh your memory of operating Gazebo before you move on. Let the robot joint variables be ϑ_1, ϑ_2.

Kill the above Gazebo job and launch the below:

```
roslaunch hw3 rrbot_world_controlled.launch
```

This time, in addition to the Gazebo world, a PID controller is launched as well. Therefore, the arm remains in its initial "stretched" position, despite the gravity.

You may play with the controller parameters by using the rqt UI:

```
roslaunch rrbot_control rrbot_rqt.launch
```

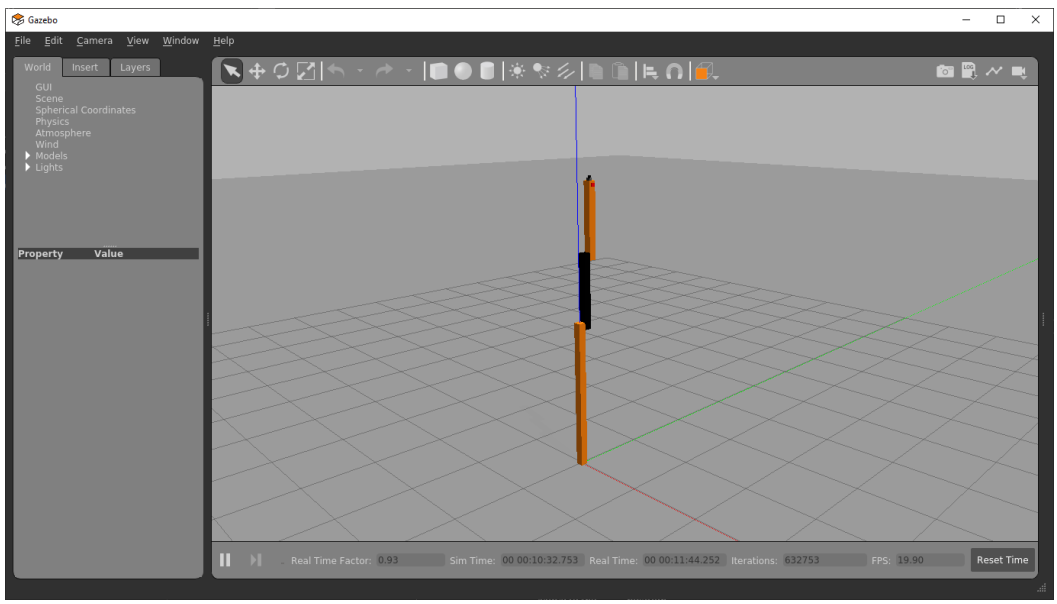
Make sure to kill the UI process before you move on.

Part VII: Trajectory Planning

Wee return now to the rrbot from the previous assignment.

The motion requirements of the arm are as follows:

- Start at the initial stretched position:



- Move the arm to $\vartheta_1 = \frac{\pi}{2}, \vartheta_2 = \frac{\pi}{2}$ at $t = 2[sec]$.
- Move the arm to $\vartheta_1 = -\frac{\pi}{2}, \vartheta_2 = -\frac{\pi}{2}$ at $t = 4[sec]$.
- Remain motionless in $\vartheta_1 = -\frac{\pi}{2}, \vartheta_2 = -\frac{\pi}{2}$ afterwards.

The motion should be smooth.

Use single spline polynomial trajectories for each of the joints. **Write down the two polynomials you selected. Explain your considerations and add your explicit calculations to the written report.**