046212 Intro to Robotics – Wet Homework 2

May 1, 2021

Submission deadline: 25/5/2021, 23:59

In this exercise you will implement forward and inverse kinematics methods for a robotic manipulator. Our workflow is organized as follows:

- We will start by setting up a new ROS workspace for this exercise.
- 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 you saw in class!

1 Submission instructions

Please submit a zip file containing the code and a written report. The code should include the `launch`, `scripts`, and `URDF` directories under your `hw2` package. The written report should include figures as required in the questions.

2 Workspace setup

For this exercise create a new workspace named `robotics2`. Use following command to create a new package called `hw2`:

```
git clone https://github.com/tomjur/robotics2.git
```

Repeat the instructions from wet homework1 to create a workspace with `catkin`. We recommend that you revisit the first wet homework for instructions on how to source files and work with ROS in the terminal.

3 Visualizing and moving the robot

Our robotic arm (Figure 1) consists of a RRR Wrist on top of an Anthropomorphic Arm [1].

1. Run Rviz by launching `display.launch` in the `hw2` package. Note that you can change the robot's joint configuration with the sliders. Play with the `Randomize` and `Center` buttons under the `joints_state_publisher` UI. Attach a screen capture of the robot in a some random joints configuration.

[1] The RRR wrist and anthropomorphic arm were covered in the lectures and tutorials (weeks 3 and 4).

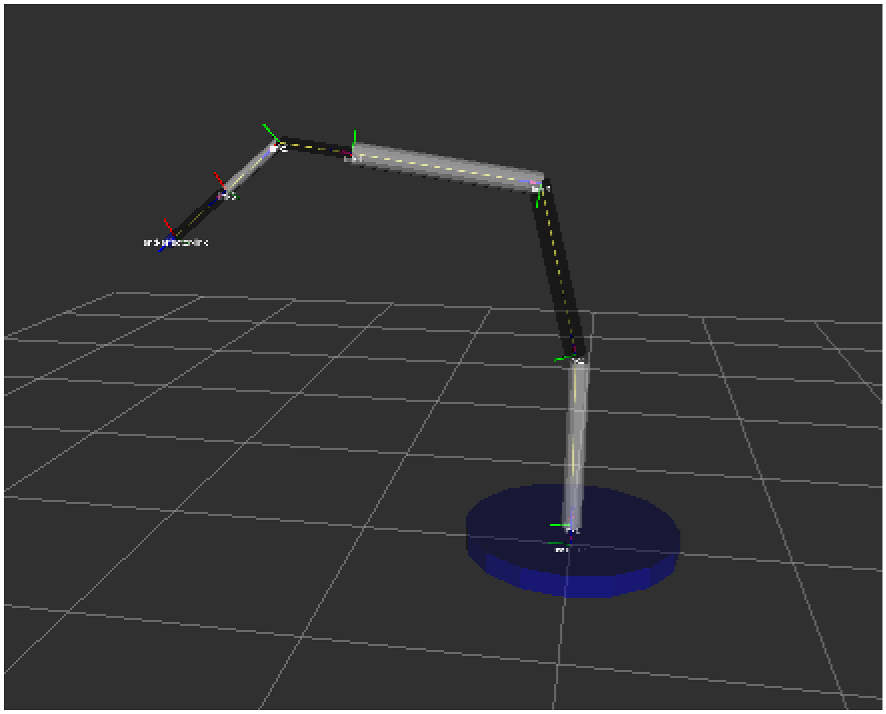

Figure 1: RViz robot visualization

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 X,Y, and Z axes, and their respective colors are red, green, and blue. We will denote as the `center` configuration the robot configuration where all joint angles are zero. For the initial configuration of `center`, what is the rotation matrix between ”base_link” and ”end-effector-link” (hint, looking in RViz is enough, code not required)?

4. We would like to implement a Python method for setting the robot joints. For this, we first need to understand how the different components in ROS interact. What are the ROS nodes and ROS topics after launching `display.launch`?

5. The launch file you used in the previous questions opens the `joints_state_publisher` node. This node draws the corresponding UI, and also publishes `JointState` messages to the topic `joint_states`. The `JointState` message contains the names and values of the robot joints (for more information see http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html). The node `robot_state_publisher`, reads these joint messages and translates the joints configuration to the link poses by applying the transformations defined by the URDF of the robot. Finally, the `rviz` node reads the link poses and creates the interactive UI for visualizing the robot. The above can be visualized with the `rqt_graph` as you saw in wet homework 1.

   Run the `rqt_graph` command and attach a screen capture of it. What is the topic that `robot_state_publisher` publishes to? Which node listens to? these messages?

6. We will next write a python method for setting the robot's joint angles. We will write a node that replaces `joints_state_publisher`, and publishes fixed joint messages. We will call this node `fixed_joints_publisher`.

(a) Create a new directory under `hw2` called `scripts`. Copy the `fixed_joints_publisher.py` we provided to that directory (don't forget to make the script executable).

(b) Create a new launch file called `display_fixed.launch` in the launch folder, by copying `display.launch`. Replace the invocation of `joints_state_publisher` by a call to `fixed_joints_publisher` (notice the package).

(c) We will now start editing the `fixed_joints_publisher.py` file. In `TODO1` you are required to save to a class members the joint names and positions the publisher is going to publish.

(d) Next, in `TODO2`, initialize the node with the name `fixed_joints_publisher`.

(e) Create a publisher that publishes to the topic `joint_states` in `TODO3`.

(f) Next comes the tricky part: set the joint names and positions in the message (`TODO4`). To identify the field names, you can search online, or use the topic and message investigation tools from homework 1.

(g) In `TODO5`, publish the message with the publisher.

(h) Finally, in `TODO6`, complete the loop by calling the message publishing function you created.

Denote by `j1` the joints configuration where each joint is set to 90 degrees. Provide a screen capture of Rviz where the robot is at `j1` (notice this is the default position provided in the skeleton code).

4 D-H Convention

Now that we can set the robot to fixed joints state, it is time to implement the kinematics algorithms you saw in class.

1. **Understanding the URDF file format:** Read through the URDF file, identify how links and joints are defined. We encourage you to play around with (a copy of) the file to investigate the parameters. Add a sphere with radius 1, connected to the end effector and 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.

   NOTE: revert back to the original URDF for the rest of this exercise.

2. **DH diagram:** Draw 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 the `center` position, and recall that at this position all joint angles are zero (q_i=0 for all i). Make sure that your solution's z axes keep the rotation in the same direction as the URDF (according to the right hand rule) – verify that positive rotation according to the diagram matches positive rotation in Rviz (which you can check using the sliders in Question 3.1).

3. Redraw the diagram for the robot position `j1`.

4. Create 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 URDF.

5. What are T_0^B and T_{EE}^n?

6. **TF intro:** TF is a ROS package that reads the robot URDF file and handles the computation of links poses as well as transformations between different frames (as defined by the URDF links). The tf package in ROS wiki: http://wiki.ros.org/tf, some tutorials http://wiki.ros.org/tf/Tutorials.

   (a) Set the robot in 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*[2].

[2]notice that in class, we defined quaternions to have a positive value in the first component: $q_0 \geq 0$.

(b) Repeat the previous question when the robot is set to `j1`.

Now when you implement your own forward kinematic solver, you can compare your results to the results from TF.

5 Forward Kinematics

You will now implement an FK solver using the provided skeleton in `hw2_services.py`.

1. First, observe the node initiation in the main function (similarly to what we did in wet homework 1). Keep the line `# solve_ik(gs)` commented as we will only need this in the next Section. Use the member `current_joints` of `GeometricServices` and save the joint states by subscribing to the `joint_states` topic (as you did in wet homework 1). Implement `TODO1` in the code (2 occurrences).

2. Create a service called `get_tf_ee` that when called, reads the value of `current_joints` and by using the `TF` package returns the pose of the end effector. Use `tf.TransformListener` as described in http://wiki.ros.org/tf/Tutorials/Writing%20a%20tf%20listener%20%28Python%29. Use `TODO2` in the code (3 occurrences). What are the values for `j1`?

3. We will now create an FK service called `get_ee_pose`, and verify its result against the result of the `get_tf_ee` service. We will use the `numpy` python package for numerical computations (https://numpy.org/).

   (a) Start by populating the DH parameters in `TODO3`.

   (b) Implement the `_generate_homogeneous_transformation` static method that returns a 2D `numpy` representing a single joint transform (`TODO4`).

   (c) Since `TF` outputs quanternions as orientation representation we need to output quanternions as well. Implement the static method `_rotation _to_quaternion` that outputs a quanternion representation based on a rotation matrix `r` (`TODO5`).

   (d) The service `get_ee_pose` is already initialized in the code. You are required to implement its core logic in method `_get_ee_pose` in `TODO6`. This method gets the joints configuration, and should use `_generate_homogeneous_transformation` and `_rotation _to_quaternion` methods to return the transformation matrix, translation, and quanternion representation of the end-effector for these joints.

   You have completed the FK solver! Verify that your result for joints configuration `j1` matches the result from `TF` from the previous question. Call your ROS service and report the resulting message in the written report.

6 Inverse Kinematics

Now that we have computed the transformation matrices, we are ready to implement an inverse kinematic solver. Our inverse kinematics representation will use ZYZ Euler angles as orientation representation for poses. Please uncomment the third line of the main function `# solve_ik(gs)` to enable this part. Our goal will be to find joint configurations that set the end effector to position $p = (-0.770, 1.562, 1.050)$ and orientation (expressed as a quaternion) as $q = (0.392, 0.830, 0.337, -0.207)$.

1. First, we need to express the required end-effector orientation in ZYZ Eular angles. Implement function `convert_quanternion_to_zyz` by completing `TODO7`. Report the ZYZ Eular angles for $q = (0.392, 0.830, 0.337, -0.207)$.

2. we will now implement the IK solver in three steps:

   (a) Implement the geometric Jacobian in method `get_geometric_jacobian` (`TODO8`). You may find it helpful to reuse `_generate_homogeneous_transformation` from previous questions.

   (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 (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 joints limits (as defined in the URDF).

   You are free to implement any one 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-axis 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.