16-662 Assignment 1: Kinematics & Control

Instructor: Oliver Kroemer
TAs: Timothy Lee and Mohit Sharma
Due Date: Wednesday, February 13, 2018 at 11:59pm

Instructions

1. **Integrity and collaboration:** If you work as a group, include the names of your collaborators in your write up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is strongly prohibited and may lead to failure of this course.
2. **Start early!**
3. **Canvas:** If you have any questions, please look at Canvas first. Please go through the previous questions before submitting a new question and adding its tag in the sticky.
4. **Write-up:** Please note that we **DO NOT** accept handwritten scans for your write-up in this assignment. Please type your answers to theory questions and discussions for experiments electronically.
5. **Code:** Please stick to the function prototypes mentioned in the handout. This makes verifying code easier for the TAs.
6. **Submission:** Please include all results in your writeup pdf submitted on Canvas. For code submission, create a zip file, `<andrew-id>.zip`, composed your Python or C++ files. Please make sure to remove any temporary files you've generated.

1 Kinematics

In the first part of this homework we will look more closely at the kinematics of our robot model. More specifically we will look at some of the basic building blocks of robot kinematics such as, *forward kinematics* and *inverse kinematics.* As noted in the class, forward kinematics use the kinematic equations of a robot to compute the position of the wrist given specific values for the joint angles. Mathematically, the forward-kinematic equations can be seen as equations that map between the space of joint position and cartesian position and orientations. Similarly, the mathematical function that maps the velocity relationship, *i.e.*, which maps the space of joint velocities to cartesian velocities, is referred to as the *Jacobian.* In the first part of this assignment we will compute both, the **forward kinematics** and the **Jacboian Matrix** for a given arm configuration.

1.1 Problem Setup

For this problem statement we will be using the LowCost Robot (LocoBot) platform. The *assignment* folder contains the *locobot_description_v3* folder which contains the URDF file for above platform. We will be using this URDF file to build the robot model to calculate both the forward kinematics and the Jacobian matrix.

The *locobot_description_v3* is a ROS package, you can download it and install it in any of your ROS workspaces. First we will look around this ROS package and see how we can visualize it. If you're not famililar with ROS and workspaces, this is a good time to go through the ROS fundamentals here `http://wiki.ros.org/ROS/Tutorials`. You only need very basic ROS knowledge to complete this part of the assignment. Once you've created a ROS workspace and added the above package to your workspace, you can build it using *catkin.* The *locobot_description_v3* contains the URDF file, meshes and some configuration files to build the Robot model. To see what the robot and its kinematic chain look like we can use *rviz.* To start visualizing you need to launch the *display.launch* file by using the following command *roslaunch locobot_description_v3 display.launch.* **NOTE: Please read basic ROS tutorials before trying these things out.** Now you should be able to visualize the robot model in rviz and also see the kinematic chain on the left hand side.

1.2 URDF model

The base link for our Robot model will be *arm_base_link* link declared in the URDF file and the target link will be *gripper_link.* We will be using this kinematic chain for forward kinematics. Notice that in the URDF file *joint_1* connects *arm_base_link* to *shoulder_link* and so on. Please make sure you're using this as the kinematic chain. Now we will look into how to calculate the forward kinematics for this kinematic chain. To calculate the forward kinematics we need the position and orientation of all the links in the above chain. This requires us to parse the URDF file. There are a few ways to achieve this for the assignment. First, you can read the position and orientation for the relevant links and joints directly from the urdf file. You can then convert these into matrices that represent the joint frames and link frames and simply apply transformations from the base link to the wrist link.

Another option to parse the above URDF file is to use any of the available ROS packages such as **urdf_parser_py** (if you're using Python) else **urdf parser** (`http://wiki.ros.org/urdf`) for C++. These packages will abstract away reading the XML directly from the URDF file and will give you a more amenable representation for the links and joints, which you can then use to implement the forward kinematics. **NOTE: Even if you use an existing ROS package to read the URDF file you still need to implement YOUR OWN forward kinematics code.**

1.3 Forward Kinematics

Once you have the URDF file parsed into links and joints we want to find the wrist position and orientation with respect to the given joint angles. The wrist pose and orientation need to be returned as a homogeneous transformation matrix $T \in SE(3)$, which is a to-scale (last element is 1) 4x4 matrix. You need to return these 16 values in either a list (Python) or a vector (C++). **NOTE:** Please use column-major format to convert your rotation matrix into 16 values.

The function signature you need to implement is

```
T = getWristPose(joint_angle_list) (In Python)
```

```
T = getWristPose(vector<double, 5> joint_angles) (In C++)
```

Here is a sample result for a given joint configuration

```
joint_angle_list = [0.727, 1.073, 0.694, -0.244, 1.302]
```

$$T(matrix) = \begin{bmatrix} 0.0355 & -0.895 & -0.443 & 0.121 \\ 0.0316 & -0.442 & 0.896 & 0.108 \\ -0.998 & -0.0458 & 0.0126 & -0.0945 \\ 0. & 0. & 0. & 1 \end{bmatrix}$$

1.4 Jacobian

In the second part of the assignment we will be calculating the Jacobian matrix for the above URDF file. Recall that the Jacobian maps the joint space velocities to the cartesian space velocities, thus it is a $\mathbb{R}^{6\times N}$ matrix for any N link arm.

The function signature that you need to implement for this part is

```
J = getWristJacobian(joint_angle_list) (In Python)
```

```
J = getWristJacobian(vector<double, 5> joint_angles) (In C++)
```

Here is a the Jacobian for the above given joint configuration

$$J(matrix) = \begin{bmatrix} -0.108 & -0.1551 & -0.1935 & -0.0469 & 0 \\ 0.1219 & -0.1382 & -0.172 & -0.0418 & 0 \\ 0 & -0.1633 & 0.0362 & -0.0029 & 0 \\ 0 & -0.6652 & -0.6652 & -0.6652 & -0.355 \\ 0 & 0.7465 & 0.7465 & 0.7465 & -0.0314 \\ 1 & 0 & 0 & 0 & 0.9986 \end{bmatrix}$$

1.5 Deliverables

1. A small, concise paragraph (2-4) sentences describing your implementation and understanding for the forward kinematics.
2. A small, concise paragraph (2-4) sentences describing your implementation and understanding for the Jacobian.
3. Report the wrist pose using your forward kinematics code and the Jacobian for the following joint configurations. Add it to your report. **Only report up to 3 decimal values for each element.**
   - $[-0.9501, 0.8786, 0.5130, -1.4157, -0.1997]$
   - $[-1.5684, 0.08128, -0.4975, 0.3102, -1.2215]$
   - $[-0.9307, -0.7170, -0.6461, 0.3086, -0.9533]$
4. Your updated *forward_kinematics.py* file and any other code you implemented. If you're using some ROS package for URDF parsing add it as a dependency of your ROS package and just export your ROS package code.

2 PID Control

In the second part of this homework, we will implement PID control in the joint space of a LocoBot simulation. This problem will also provide an introduction to V-REP, a robot simulator that we will be using in this course. Please use Python 2.7 for this assignment (with your choice of software to plot your results).

Installing V-REP. Download V-REP directly from the Coppelia Robotics website: `http://www.coppeliarobotics.com/downloads.html`. Select the "V-REP PRO EDU V3.5.0 rev4" version. V-REP supports Windows, Mac, and Ubuntu.

- For Ubuntu, navigate into the V-REP directory and run the launch script: `./vrep.sh`
- For Mac, run the executable in the downloaded zip file, and a GUI should open.

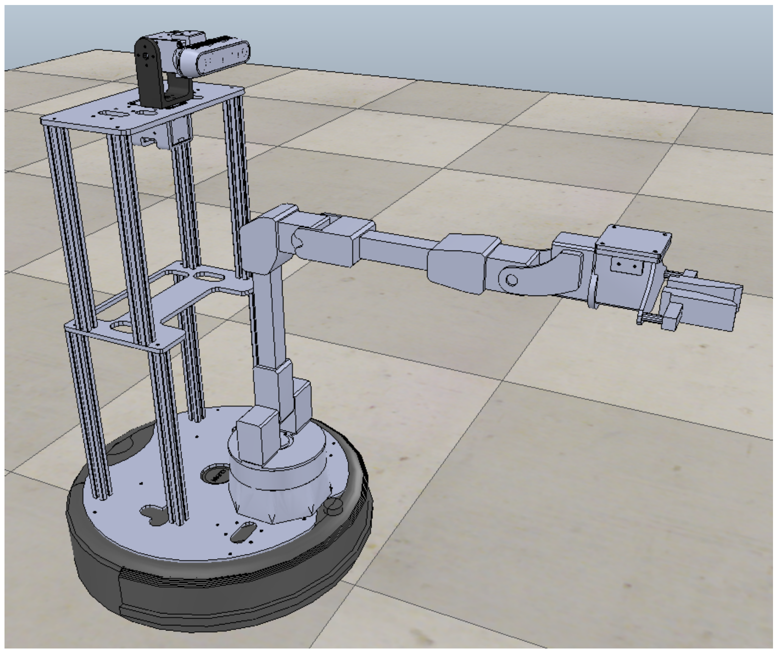

Figure 1: The LocoBot model in V-REP, as imported from the URDF file.

Importing the LocoBot URDF. First, we need to update the paths included in the URDF file because we are not using ROS for this portion of the homework. Make a copy of the URDF file included in the homework and replace all instances of `package://` such that the `mesh filename` tags reflect the absolute path to these files on your computer. Once you've done that, open V-REP and import this modified URDF file by selecting `Plugins` → `URDF import...`. Keep the default import options. Do not be alarmed if many errors and warnings are shown; if you see the same model as in Fig. 1, it has imported correctly. (The model does import the gripper fingers backwards, but we will account for this in the assignment.) At this point, you may wish to save the simulation scene through File → `Save scene` to avoid needing to import the model again. **Do not change any other simulation parameters, such as physics engine, timestep, joint properties, joint names, etc, after importing the model.**

On the left side of V-REP, you will see the Scene Hierarchy tree. Scroll down and find the joints named `joint_1`, `joint_2`, ... `joint_7`. We will be working with these arm joints for this problem (see Table 1). We will not be using other parts of the robot for now, such as the mobile base and the camera.

Once the model is loaded, start the simulation (in the menu, `Simulation` → `Start simulation`, or press the Play button on the toolbar). You should observe that the arm starts to pivot forward. Why does this happen? The weight of the arm applies a torque on the base. This causes an uncorrected rotation at the base joint because no control inputs are being applied to counter this torque. Indeed, the role of the control system is to introduce system inputs to drive the system into a desired state. In our case, we want a PID controller to work in

Table 1: Description of LocoBot arm joints.

Joint name	Type	Parent link	Child link
`joint_1`	Revolute	`arm_base_link`	`shoulder_link`
`joint_2`	Revolute	`shoulder_link`	`elbow_link`
`joint_3`	Revolute	`elbow_link`	`forearm_link`
`joint_4`	Revolute	`forearm_link`	`wrist_link`
`joint_5`	Revolute	`wrist_link`	`gripper_link`
`joint_6`	Prismatic	`gripper_link`	`finger_r`
`joint_7`	Prismatic	`gripper_link`	`finger_l`

the joint space of the robot to control the system.

2.1 Coding Assignment.

Implement a PID controller in `locobot_joint_ctrl.py` that consumes the joint positions of the LocoBot arm and calculates the necessary joint control inputs to achieve a sequence of three consecutive target joint positions (defined in `joint_targets`). The joint vector $j \in \mathbb{R}^7$ is the collection of the arm revolute joints and end effector prismatic joints (in the same order as Table 1). For actuation, we will directly actuate joint velocity through the input $u \in \mathbb{R}^7$ that is calculated from the difference between the target and feedback joint positions (also called the joint error). The PID controller acts to minimize this joint error, driving the system to the target joint configuration. In reality, we do not directly control joint velocity, but usually a mechanism of actuation such as motor voltage, current, and/or drive signal (such as PWM [pulse width modulation]). For this simulation assignment, we control joints directly by their velocity, as this results in a more stable physics simulation and pleasant homework experience. There are other minor discrepancies between this model and the actual robot you will use, such as the use of the two prismatic joints in the simulation to control the end effector.

Three target joint positions have been defined in `target_joints` in `locobot_joint_ctrl.py`. The controller **must stably converge** to each target configuration before proceeding to the next target. This demonstrates that your controller has been implemented correctly through achieving small steady state error at each target over a period of time (e.g., 1 second). However, it is left to your creativity to best determine the convergence criteria. However, if your steady state error is too large (over several degrees), you will not receive full credit. (The integral and derivative terms in your controller may help with this.)

It is fine if your controller initially overshoots the target joint positions, as long as it eventually converges before moving on to the next target.

You have been provided with the `ArmController` class, wherein you will implement your methods. Specifically, you will:

- Implement the `__init__` constructor for `ArmController`;
- Implement the necessary control logic in `calculate_commands_from_feedback`;
- Implement your condition for determining whether the controller has reached steady state convergence to the target joint position in `has_stably_converged_to_target`;
- Implement any plotting or post-simulation code as needed after the simulation terminates. You may include this directly within `locobot_joint_ctrl.py`, or you may use a separate script or software (e.g., MATLAB). The existing class variable that aggregates the time history may be useful for this.

Please do not modify outside of the designated areas in `locobot_joint_ctrl.py`; it will otherwise delay grading your submission.

Deliverables:

1. A small, concise paragraph (2-4 sentences) describing your controller. Include your controller gains.
2. A concise description (1-2 sentences) of your steady state convergence criterion.
3. A time history plot for each of the seven joints. Include the target joint position on each plot. **Your plots must include labels and units on each axis.** [This may seem strict, but following such best practices for data visualization greatly improves how effectively your results are conveyed.]
4. Your completed `locobot_joint_ctrl.py` file and any other scripts you implemented, such as any plotting scripts (if you did not implement this in `locobot_joint_ctrl.py`).