Foundations of Robotics, Fall 2018

Coding Project 3: Manipulator Kinematics
(CS 4750: 95 points, CS 5750: 100 points)

Due Friday, Oct. 19 at 3:00 PM

Objective:

Demonstrate understanding of the mathematical formalisms underlying serial chain manipulator kinematics.

Collaboration Policy:

This assignment can be discussed as a group of arbitrary size. You may work with up to one partner on this assignment, but each student is responsible for writing and submitting a separate solution, and **no student should see another student's solution**. Include in a comment at the top of each file your name and the names of all classmates you discussed any part of the project with.

Using the Simulator Tool:

To run code for the assignment, use the command `./simulator-tool run p3 problemX`, where `problemX` is either `problem1`, `problem2`, `problem3a`, or `problem3b`. Note that you must complete the launch file for each problem in order for your code to be executed.

If you encounter difficulty using the simulator tool, please contact the course staff. Note that you can also run `./simulator-tool --help` to get a listing of commands and `./simulator-tool <command> --help` to get more information on the usage of a specific command.

Once you run a problem with the simulator tool, the V-REP simulator will open up. **To start the simulation, you must hit the play button.**

Debugging Tips:

If you add `output="screen"` to your launch file after you specify `type="something.py"`, the print statements from that Python file will print to your terminal. We recommend using `rostopic echo` to see what messages are being published. Refer to the ROS documentation for more information.

We will have a FAQ and clarifications post for this assignment pinned on Piazza, so please check that post regularly for updates and information.

Assignment:

Implement code to solve the following problems, using the provided simulator framework. When you're finished, compress your complete ROS package with the command `./simulator-tool package p3`. Then upload the compressed package (named `p3_solutions.tar.gz`) to CMS. Note that you can upload to CMS as many times as you'd like up until the deadline; the most recent submission will override all the previous ones. Keep this in mind and don't wait until the last minute to submit! We recommend uploading at least a half hour before the deadline to ensure that your solutions are submitted on time.

Remember to run `./simulator-tool check p3` prior to submitting, which verifies that your submission is runnable and can be graded correctly. If this command returns errors or warnings, you will lose points on the assignment. Bear in mind that a successful return from the `check` command does not guarantee full points on this assignment.

Note about NumPy:

It is highly recommended that you use NumPy for *all* matrix multiplications. Appendix B provides information about common NumPy functions, examples, and links to more information.

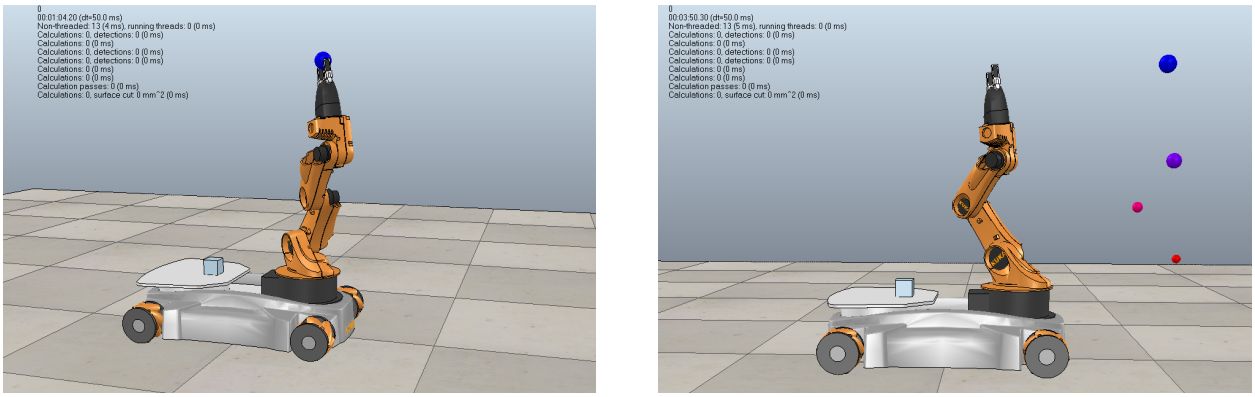
1 Forward Kinematics (30 points)

This problem is designed to assess your ability to compute the forward kinematics (FK) of a serial chain manipulator. Specifically, you will be computing the forward kinematics of the manipulator arm of a KUKA youBot. Appendix A provides a design specification for the youBot arm. Please note that the design specification gives lengths in millimeters, but the V-REP simulator uses meters.

To complete this problem, design a ROS service named `fk` that computes the forward kinematics for a given arm configuration. Take a look at `srv/Problem1.srv` for the proper input and output types of your service. You may find it helpful to first derive the Denavit-Hartenberg parameters of the youBot arm.

Note that the return value of `fk` has type `PointArray`, which we define for you. Points in this array, which have type `geometry_msgs/Point`, are interpreted as the positions of the origins of the arm link frames. Since there are five links in the youBot arm (not including the base), you should put the five origins in the array. For each of the five points in the array, a ball of a different size and color will be displayed in the simulator so that you can see where your `fk` service believes that each link is located. The final point in the array should be the position of the end-effector.

All of your code for this problem should go in the provided `src/problem1.py` file. Additionally, `problem1.launch` is completed for you.



(a) Correct FK calculation

(b) Running in debug mode

Figure 1: Visualization of forward kinematics calculations. Image (a) shows a correct FK calculation with debug mode turned off; note that the blue ball is between the fingers of the end-effector. Image (b) shows an example of running the problem in debug mode.

When you run this problem with the simulator-tool, you will see that the arm moves quickly to a series of random configurations. If your implementation of the `fk` service is correct, then you will see a blue ball between the youBot fingers.

If your implementation has a bug, then it may be difficult to determine from the simulation what is wrong because the arm moves rapidly. We have therefore provided you with a debug mode. In the file `problem1.launch`, there is a ROS parameter named `fk_debug` that is set to `False` by default. When it is set to `True`, the arm will move slowly through the entire range of each joint. Furthermore, each ball showing your FK solution will be offset 0.4 meters in front of the robot. If you provide the positions of all five frame origins in the return value of `fk`, then using debug mode will help you identify mistakes in your implementation.

2 Velocity Kinematics (30 points)

This problem is designed to assess your understanding of velocity kinematics and your ability to implement simple velocity kinematics computations.

To complete this problem, design a ROS service named `vk` that numerically computes the Jacobian matrix of the youBot arm at a given configuration. It is sufficient to compute only the linear rates and neglect the angular rates for this problem (because you only need to give a position in the next problem, not an orientation too). Look at `srv/Problem2.srv` for the proper input and output types of your service.

All of your code for this problem should go in the provided `src/problem2.py` file. Additionally, most of `problem2.launch` is completed for you. You will need to add the line to run `problem2.py`. You will also need to add the line to run your previous service, which your `vk` service should use.

We have provided a tool to help you verify that your service is working correctly. The tool, which is launched when you run the problem with the simulator-tool, provides a visualization of the Jacobian matrix returned by your service. If your calculations are correct, the entire matrix should be bright green. Incorrect results will show with varying shades of yellow and red, indicating how far off the calculations are. See Figure 2 for some example visualizations.

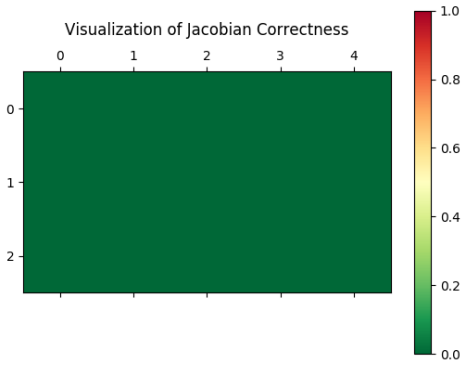
3 Inverse Kinematics (40 points)

This problem is designed to assess your understanding of inverse kinematics and your ability to implement simple inverse kinematics computations.

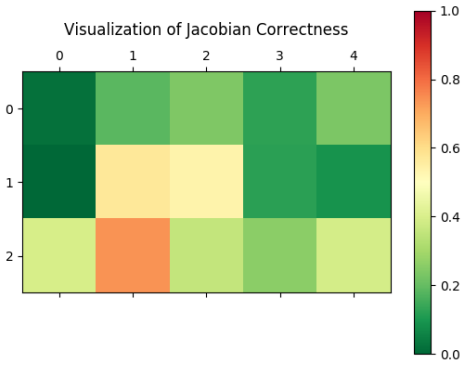
3a (20 points)

Design a ROS service named `ik` that computes the inverse kinematics of the youBot arm for a given end-effector position by employing the iterative algorithm given in Section 3.9.3 of the reading, using the Jacobian transpose to approximate the inverse. Take a look at `srv/Problem3a.srv` for the proper input and output types of your service.

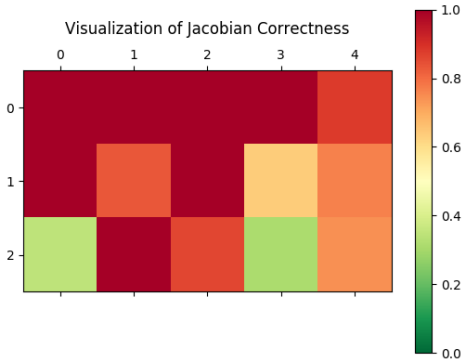
All of your code for this problem should go in the provided `src/problem3a.py` file. Additionally, most of `problem3a.launch` is completed for you. You will need to add the line to run `problem3a.py`. You will also need to add the lines to run your previous two services.



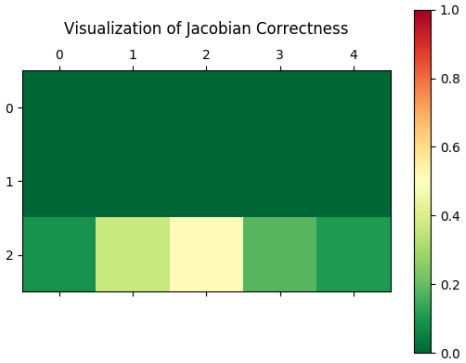
(a) Correct Jacobian calculation



(b) Partially correct Jacobian calculation



(c) Incorrect Jacobian calculation



(d) One row of Jacobian incorrect

Figure 2: Visualization of Jacobian matrix calculations. Note that the matrix has three rows (one for each linear degree of freedom) and five columns (one for each joint angle). Graph (a) depicts a correct result, graph (b) depicts a partially correct result, graph and (c) depicts an incorrect result. Graph (d) shows a result in which only the third row is incorrect, which indicates that something is wrong with the z-component of the linear velocity calculations.

To test your service, you can use the end-effector position of type `geometry_msgs/Point` from the topic `p3/effector_position` as input to your service and publish the resulting output to the five joint angle topics `/vrep/youbot/arm/jointX/angle` (where `X` $\in [1, 5]$ is the joint number) with type `std_msgs/Float64`. Similar to the first problem, publishing to these five topics will move the simulated arm such that the blue ball is between the fingers of the end-effector, but this time you are controlling the position of the arm, not the ball. Figure 3 depicts the correct behavior.

Note: Your implementation should not move the base of the robot.

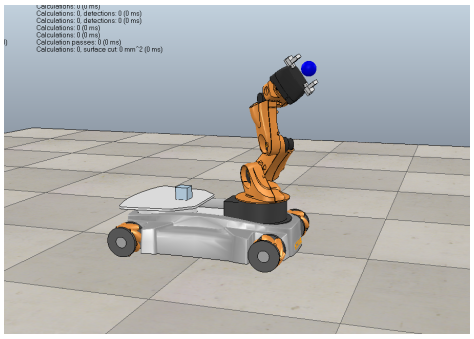


Figure 3: The blue ball should be located between the fingers of the end-effector.

3b (20 points)

In this problem, you will use your `problem3a` service to make the youBot arm perform a back-and-forth waving motion. Note that you are not creating a new service for this problem.

The topic `p3/wave` will provide messages of type `geometry_msgs/Point` that correspond to the waving motion. Use your `problem3a` service to calculate the corresponding joint angles and then publish the angles to the five joint angle topics `/vrep/youbot/arm/jointX/angle` (where `X` $\in [1, 5]$ is the joint number) with type `std_msgs/Float64`.

All of your code for this problem should go in the provided `src/problem3b.py` file. Additionally, most of `problem3b.launch` is completed for you. You will need to add the line to run `problem3b.py`. You will also need to add the lines to run all of your previous services.

If done correctly, the arm of the youBot should swivel left to right, as shown in Figure 4 below.

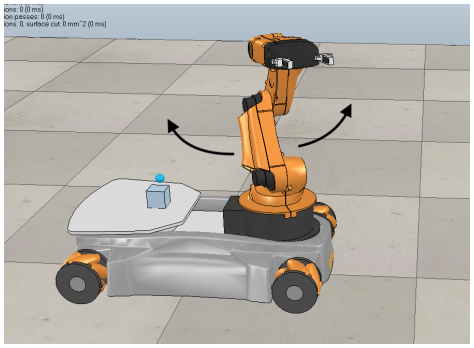


Figure 4: The youBot arm performing a "wave".

A KUKA youBot Design Specification

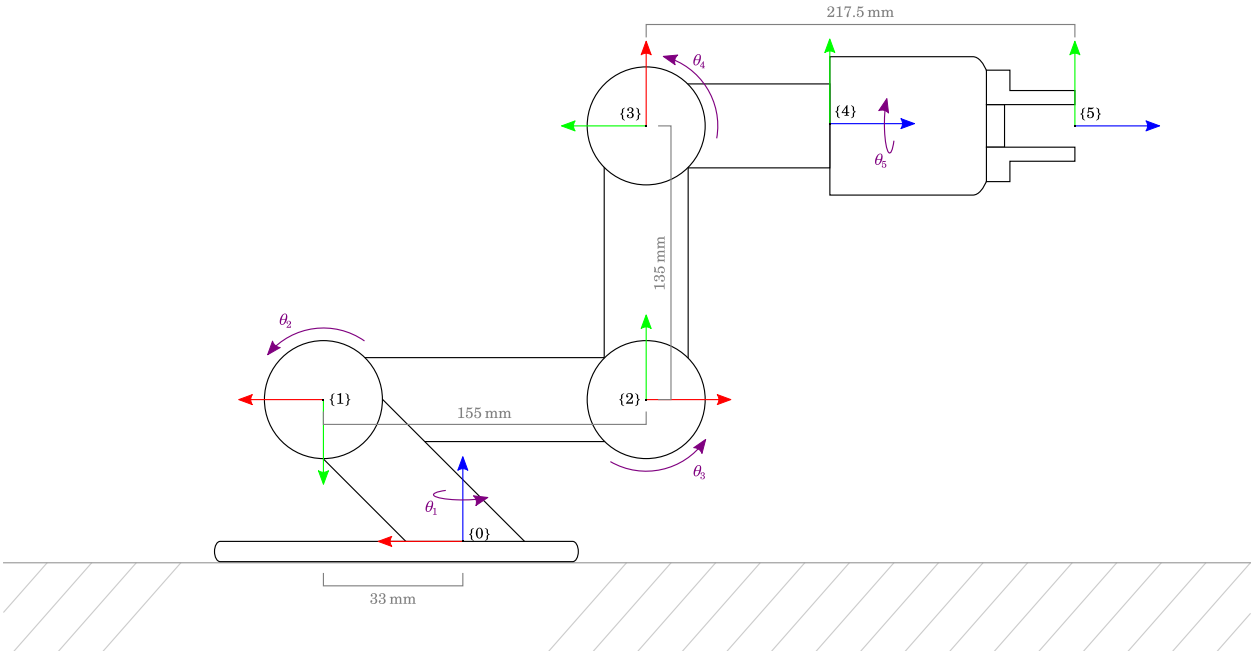


Figure 5: Informal design specification for the KUKA youBot manipulator arm, which has five revolute joints and a two-finger gripper. The angle of joint i is given by θ_i, and the direction in which the angle is measured is shown with a purple arrow. Each link in the arm has a frame rigidly attached to it, and the x-, y-, and z-axes of the frame are shown in red, green, and blue, respectively (the one unshown axis for each frame points either out of or into the page). Note that z_i is the axis of rotation for joint $i+1$, **not** joint i. The frames are attached in this manner so that when joint i is actuated, link i and its attached frame $\{i\}$ move. Finally, note that since axes z_3 and z_4 intersect, the origin of frame $\{4\}$ can be placed at any point along z_4. We have depicted it at the wrist for the sake of the visual, but in practice we would place it at the intersection of z_3 and z_4 to simplify the math.

B NumPy Overview

Importing NumPy First, make sure to import NumPy: `import numpy as np`

Constructing an array A NumPy array can be any number of dimensions. To create a n-dimensional array, you can use `np.array()`. For example, `a_1D = np.array([1.,2.,3.])` creates a 1D array of floats, and `a_2D = np.array([[2,2],[0,1]])` creates a 2D array of ints where the first row is [2, 2] and the second row is [0, 1].

Special arrays NumPy includes some useful functions to create special arrays. One such function is `np.zeros(shape)`, which returns a n-dimensional array of the specified shape (a tuple specifying each of the n dimensions). For instance, `np.zeros((3,2))` returns a 3×2 array of 0s. Another useful function is `np.identity(n)`, which returns a $n \times n$ identity matrix.

Constructing a matrix A NumPy matrix is just a special 2D NumPy array. Use `np.matrix()` to create a matrix. For example, `m = np.matrix([[2,2],[0,1]])` creates a matrix with the same values as `a_2D` from above. However, while they may be very similar and you can do the same things with both, the matrix and array types are not exactly the same. Besides the fact that a matrix can only be 2D, an important difference is in matrix multiplication, as described next.

Matrix multiplication To perform matrix multiplication with the matrix type, you can simply use the * operator. For example, we can multiply the matrix `m` we defined above by doing `m*m`. If you're working with 2D matrices, the matrix type is convenient since matrix multiplication is very intuitive. **Be careful:** In this case, the * operator performs matrix multiplication like you might expect. However, if you use n-dimensional arrays instead of matrices, the * operator will multiply the arrays element-wise. You can do normal matrix multiplication with arrays by using NumPy's `dot()` function. For example, `a_2D.dot(a_2D)`, or alternatively `np.dot(a_2D,a_2D)`, will return the same matrix product as `m*m`, and these are NOT equal to `a_2D*a_2D`.

Inverse and transpose Getting the transpose of a matrix/array with NumPy is easy. You just need to use `.T` (e.g. `m.T` or `a_2D.T`). You can get the inverse of a square matrix/2D array using `np.linalg.inv()`. For example: `np.linalg.inv(m)`.

Indexing You can access specific parts of an array/matrix by indexing particular cells like you would expect (e.g. `m[1,0]` would return 0). You can also use slicing to get subsets of a matrix/array. For example, `m[:,1]` returns the second column of `m`, where the colon selects all of the rows and the 1 specifies the column we want.

For a more in-depth NumPy guide, you can read the quickstart tutorial here.