RPAL ROS & Programming Skills Assessment

1 Objective:

Demonstrate understanding of basic ROS usage and Python (or C++) programming, as well as some basic robotics concepts. This project will be used to assess your readiness to begin working in RPAL as an undergraduate researcher.

2 Collaboration Policy:

This assignment **must be completed alone**. You are permitted to use any internet resources (short of asking for help on a forum, etc.), books, tools, what have you, but you may **not** collaborate with anyone and you may **not** ask any grad students or anyone else for assistance. This is an assessment of you and your current skills; if you violate this policy we cannot make an accurate assessment and will not be able to work with you.

Note that you can come to us if you are truly stuck. If the issue is a bug or simple misunderstanding with the provided code, we can help you out and you can get back to the assessment. **However**, if you come to us and the provided code is working correctly (i.e. the error is with your code), that will constitute termination of the project and we will assess your work in whatever state it is in.

3 Running the simulator framework

We have provided a simulation framework for you to develop and test your code. In order to use the framework, you'll need to carefully follow these instructions.

3.1 Setup

Do these steps *once*, before starting your work. We also recommend using Git to store your partial progress. Note that successfully completing the setup process is part of the assessment.

3.1.1 ROS

Follow the instructions here: `https://wiki.ros.org/melodic/Installation` to install ROS Melodic.

3.1.2 V-REP

The simulation framework is based around V-REP, which you can download at that link. Get the "Pro EDU" version for Linux and extract the files to some directory (`~/V-REP` is a good choice). Finally, copy the file `libv_repExtRosInterface.so` from this directory to wherever you extracted V-REP.

3.1.3 Catkin Workspace

Before starting the project, you need to set up a catkin workspace. To do this, follow the instructions here: `https://wiki.ros.org/catkin/Tutorials/create_a_workspace`. Afterwards, copy the `src` subdirectory from this package into `<YOUR CATKIN ROOT>/src/rpal_ros_test` and run `catkin build` to build the initial workspace.

3.2 Using V-REP

The second part of this assignment will require you to run V-REP. To this end, we have provided a V-REP scene file, `assessment.ttt`, in this directory. Use the following steps to run the simulator:

1. Start `roscore`.
2. (assuming you extracted V-REP to `~/V-REP`), run `~/V-REP/vrep.sh assessment.ttt` in this directory.
3. Press the "Play" button in the upper toolbar.
4. Run the commands you need for the particular problem.
5. Run the commands to execute your code.

To start the simulation, you must hit the play button.

4 Assignment:

RPAL primarily uses ROS and Python for research coding. Thus, you will need to write Python code using rospy to be an effective researcher in the lab. If you need a refresher for Python, we suggest this quick reference[1]. You should also use the numpy library for matrix calculations and linear algebra, which is common in robotics research.

If you need a reference for ROS, we suggest the ROS "Getting Started" guide, the rospy documentation, and Jason O'Kane's "A Gentle Introduction to ROS"[2].

Implement code to solve the following problems, using the provided simulator framework. Please also document any difficulties you encountered and how you overcame them in a file named `problems.txt`, which you should include with the final submission. When you're done, bundle up the *whole* `rpal_ros_test` directory into a `.tar.gz` and send it to us for assessment.

[1] Note that we use Python 3 in most research.

[2] Note that AGITR uses C++ instead of Python. The concepts will be the same, but the language used is different.

5 Part 1: Basic ROS

This section is intended to assess your knowledge of and ability to use basic ROS concepts.

5.1 Publishing and Subscribing

The goal of this problem is to have you learn how to move a turtle using Twist messages and calculate the distance between turtle and a point.

The work for this problem should be done in file `pubsub.py`.

(a) In `__init__`:

i. First, have your turtle subscribe to `turtle1/pose` with the callback function `self.update_pose`.

ii. Have your turtle publish to `turtle1/cmd_vel`.

iii. Specify a publishing rate of 10.

(b) In `move`:

i. Create a Twist message with values that change both the angular and linear velocities such that the turtle moves both angularly and linearly.

ii. Publish the Twist message to the velocity publisher.

(c) In `update_pose`:

i. Set the instance variable `self.pose` to the information provided by the subscriber.

(d) In `calculate_distance`:

i. Using `numpy`, set `vector` to be the difference between the coordinates of the turtle's current pose and the point (x, y). (Hint: Use a `numpy` array for your vector)

ii. Using `numpy` and `vector`, set `distance` to the distance between the turtle and the point (x, y).

iii. Set the ROS parameter to `distance`. (Note: the value of distance must be cast to a float).

(e) In `pubsub.launch`:

i. Review the format of this file and understand what it does (see ROS launch file documentation for more information).

5.2 Single Flower

This problem is designed to have you create a node and generate specific behavior for your turtle using `numpy` commands. Your turtle's objective is to create a single flower pattern. See Figure 5 for exactly what the flower should look like.

The work for this problem should be done in file `singleflower.py`. Look for the comments throughout the code that correspond to the instructions in the writeup.

(a) In `__init__`:

i. Start a new node with the name 'flower'.

ii. Create a publisher that publishes to `/turtle1/cmd_vel` with type `Twist` and a queue size of 10.

iii. Set your rate to 5.

(b) In `draw_flower`:

i. To calculate constants, use `numpy` to perform the following operations on the following matrices:

$$M_1 = \begin{bmatrix} 1.5 & -1.5 & 0 \\ -1.5 & 0 & 2 \\ 2.5 & 0 & 0 \end{bmatrix} \; M_2 = \begin{bmatrix} 3 & 5 & 9 \\ -1 & 2 & 6 \\ -3 & 2 & 0 \end{bmatrix}$$

$$a = (M_1 * M_2)_{(0,1)} + (M_1 * M_2)_{(1,1)}$$

$$b = -(\langle M_1, M_2 \rangle_{(0,2)} + \langle M_1, M_2 \rangle_{(1,2)}) - 1$$

$$A = \langle M_1, M_2 \rangle_{(0,1)} - \langle M_1, M_2 \rangle_{(0,0)} + 0.5$$

$$B = -((M_2 * M_1)_{(1,2)} - (M_2 * M_1)_{(2,1)})$$

$$\texttt{max\_times} = (\texttt{det}(M_2)/-9) - 4$$

Note: Angle brackets indicate taking an inner product, subscripts indicate the indices of the term that should be taken from a matrix

ii. Each of the above computed constants should be set as rosparams `a`, `b`, `A`, `B`, and `max_times` with the type `float`.

iii. In `while not rospy.is_shutdown()` section: Use `numpy` to set angular z to $B * \cos(b * count)$ and linear x to $A * \sin(a * count)$. Be sure to publish your velocity message and sleep your turtle for your preset rate.

iv. Also in `while not rospy.is_shutdown()` section: If value *count* is greater than $2 * (\texttt{times}) * \pi$: publish a velocity message with angular $z = 5$ and linear $x = 0$, then increment `times` by 1 and if `times` is greater than `max_times` stop drawing the flower.

(c) In `if __name__ == '__main__'` section:

Create your turtle and have the turtle use your written function to draw a flower. Your final product should look something like Figure 5.

(d) In `singleflower.launch`:

i. Launch the turtlesim node.

ii. Launch the `flower_turtle` node from `singleflower.py`.

5.3 Service Clients

This problem is designed to introduce the client-service structure. For this question you will be using our implemented service and writing your own client to create a slightly different single flower then before. See Figure 2 for what the flower should look like.

The work for this problem should be done in file `client.py`.

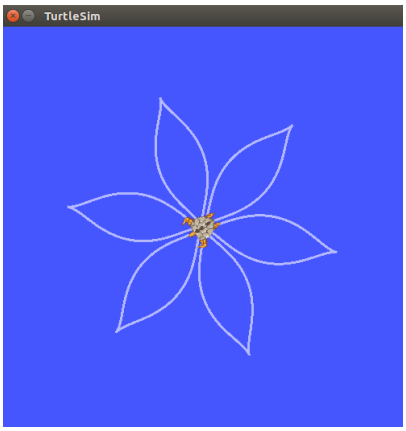


Figure 1: For Problem 5.2. Single Flower

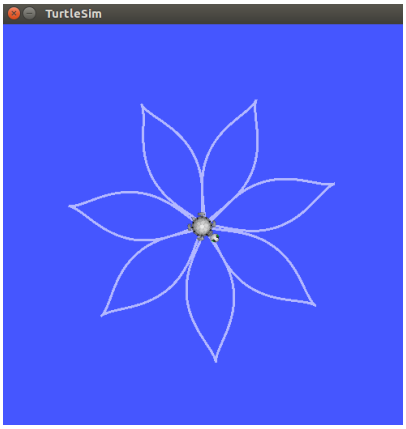


Figure 2: For Problem 5.3. Service Clients

(a) In `draw_flower`:

   i. Initialize your node with the name `drawing_turtle`.

   ii. Wait for the service `draw`.

(b) In `try` section: Fill in the service handle for the draw service with `draw` and `Draw`. Check `srv/Draw.srv` and `rospy` documentation for more information.

(c) After the `try` section: Fill in with a publisher that publishes to `/turtle1/cmd_vel` with type `Twist` and queue size of 10 then set your rate to 5.

(d) In `while not rospy.is_shutdown()` section:

   i. Use the draw service to get the velocity message to publish. Look at `srv/Draw.srv` to determine the parameters and appropriate return name.

   ii. Sleep for the previously stated rate.

(e) In `if __name__ == '__main__'`: Call your function to draw the flower!

(f) In `client.launch`:

i. Launch the turtlesim node.

ii. Launch the `draw_service` node.

iii. Launch the `draw_flower` node.

5.4 Services

This problem is meant to have you write a service for the client-service structure.

5.4.1 Creating a Service

The work for this problem should be done in the file `draw_server.py`.

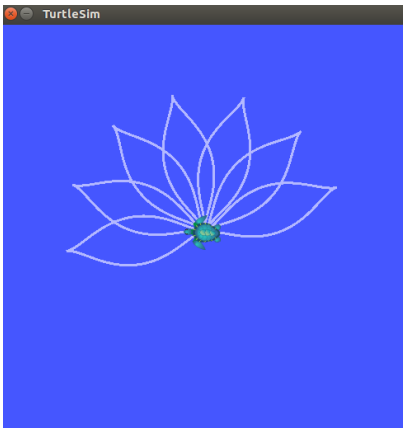


Figure 3: For Problem 5.4.1

(a) In `draw`:

i. Initialize `vel_msg`.

ii. Set k to be the count parameter of `req`.

iii. To calculate constants, use `numpy` to perform the following operations on the following matrices:

$$M_3 = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 5 & 8 \\ 13 & 21 & 34 \end{bmatrix} \quad M_4 = \begin{bmatrix} 1 & 8 & 7 \\ 2 & 5 & 3 \\ 9 & 2 & 6 \end{bmatrix}$$

$$a = det(M_3) - 1$$

$$b = -((M_3 * M_4)_{(2,0)} - (M_3 * M_4)_{(2,2)}) + 1$$

$$A = (\langle M_3, M_4 \rangle^T_{(0,0)} - \langle M_3, M_4 \rangle^T_{(1,0)})/10$$

$$B = ((M_3 * M_4)^T_{(2,1)} - \langle M_3, M_4 \rangle^T_{(2,1)})/2.0$$

iv. Each of the above computed constants should be set as rosparams `a`, `b`, `A`, and `B` with the type `float`.

(b) In `if req.rotate` section:

i. If `req.rotate`, set angular z to -3 and linear x to 0.

ii. Otherwise use `numpy` to set angular z to $B * cos(b * count)$ and linear x to $A * sin(a * count)$.

(c) Also be sure to return your velocity message before the end of the `draw` function.

(d) In `draw_server`: Create a service with the name `'draw'`, service_class `Draw` and handler `draw`. Please see rospy documentation for further details.

(e) In `__name__ == '__main__'`: Call your function!

(f) In `draw_server.launch`:

i. Launch the turtlesim node.

ii. Launch `draw_service` node from `draw_server.py`.

iii. Launch `draw_flower` node from `client.py`.

Your resulting flower should resemble Figure 3.

5.4.2 Integrating the pieces

The work for this problem should be done in file `spawn_flower.py`.

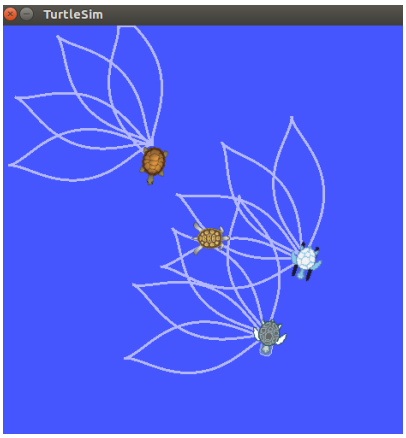


Figure 4: For Problem 5.4.2

(a) In `draw_flower`:

i. Initialize your `'draw_turtle'` node .

ii. Block while waiting for the `'spawn'` service to become available. See rospy documentation for more information.

iii. In first `try`: Create a handle for calling the spawn service.

iv. In second `try`: Uncomment the provided lines of code

v. Block while waiting for `'/draw'` service, and then set up the service handle for the draw service (in the same way as problem 3).

vi. Set up velocity publisher for `'/cmd_vel'.format(turtle_name)` and set the rate to 5.

(b) In `while not rospy.is_shutdown()` section of `draw_flower`: Publish the velocity messages returned from draw. Then sleep for predetermined rate.

(c) In `__name__ == '__main__'`: Call your function!

(d) In `spawn_flower.launch`:

i. Launch the turtlesim node.

ii. Launch `draw_service` node from `draw_server.py`.

iii. Launch `launcher` node from `launcher.py`.

Your resulting product should resemble Figure 4. There should be between 2 and 5 turtles spawned, in addition to the original turtle. The starting points of the turtles is random.

6 Part 2: Forward Kinematics

This section, which is intended to be more difficult, will assess your ability to use ROS in a more realistic robotics context as well as your knowledge of/ability to learn some fundamental robotics concepts.

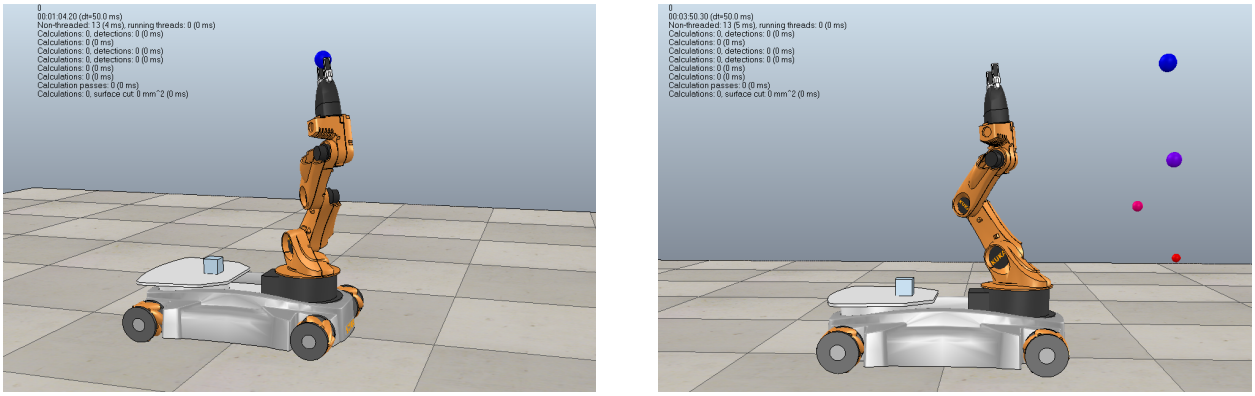
This problem is designed to assess your ability to compute the forward kinematics (FK) of a serial chain manipulator. Specifically, you will be computing the forward kinematics of the manipulator arm of a KUKA youBot. Appendix A provides a design specification for the youBot arm. Please note that the 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/Fk.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. If you need a refresher on DH parameters or forward kinematics, these notes may be helpful.

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/fk.py` file. Additionally, `fk.launch` is completed for you.

When you run this problem, 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.



(a) Correct FK calculation

(b) Running in debug mode

Figure 5: 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.

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 `fk.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.

A KUKA youBot Design Specification

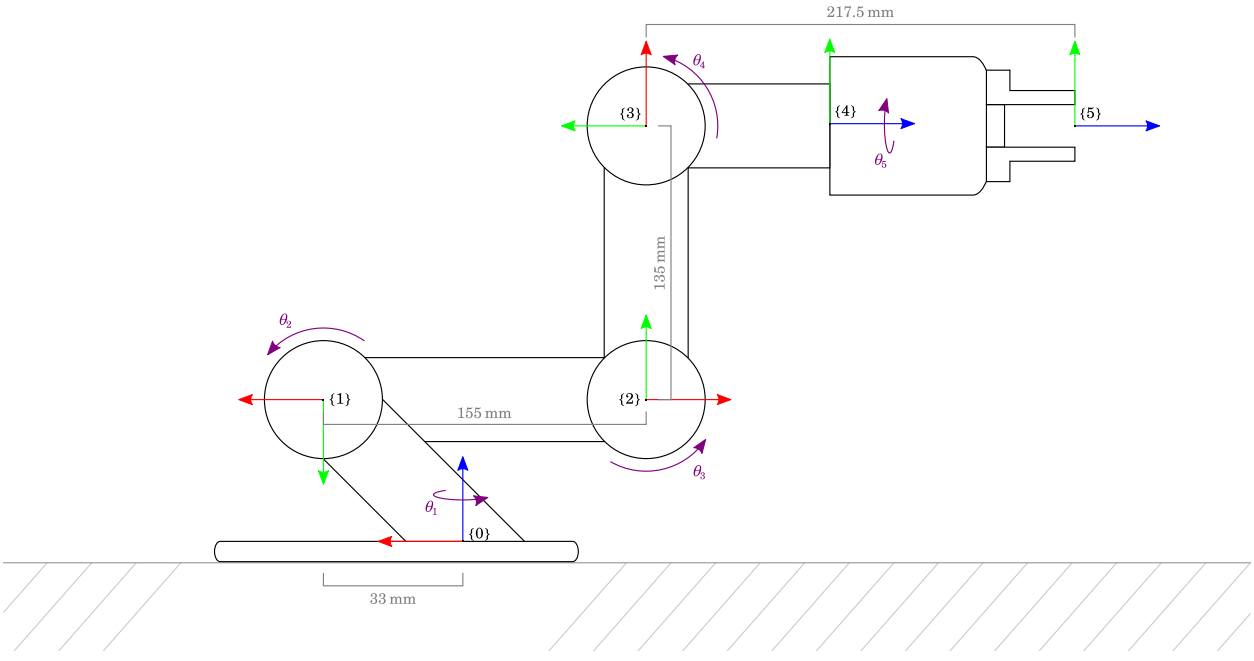


Figure 6: Informal design specification for the KUKA youBot manipulator arm, which has five revolute joints and a two-finger gripper. The angle of joint i is given by θ_i, and the direction in which the angle is measured is shown with a purple arrow. Each link in the arm has a frame rigidly attached 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.