W23 CMPUT 412/503 Exercise 3

Computer Vision for Robotics

Primary TA's: Justin and Rizwan
Written report due: March 5th
Oral demonstration: March 8th/10th

Starting Template - https://github.com/wagonhelm/cmput412_exercise3

Part One - Computer Vision

1.1 Augmented Reality

Your first task is to become more familiar with camera projections and using the apriltag library. To begin, start by reading and implementing Part A - Perception Fundamentals. You will create an augmented_reality package and an augmented_reality_apriltag package.

There are no deliverables required for this section, these exercises are designed to help you understand lane following and apriltag detections.

1.2 Apriltag Node

For later localization you will need to write a node that:

1. Subscribes to the camera image topic
2. Use your camera calibration file to un-distort the image.
3. Converts the image to black and white
4. Efficiently detects apriltags at a reasonable rate and publishes a new image topic with labeled apriltags.
    a. i.e. draw a bounding box / print tag number in the center of detection / etc…
    b. Note: we use the tag36h11 apriltag family
5. Changes the leds based on apriltag detection
    a. **Red** : Stop Sign
    b. **Blue** : T-Intersection
    c. **Green** : UofA Tag
    d. White : No Detections
6. (Optional) Ignores apriltag detection from far distances

This Node will not be needed until **3.6**

Deliverable 1: Record a video of your apriltag detector image topic detecting an apriltag and labeling the apriltag.

Questions

- What does the april tag library return to you for determining its position?
- Which directions do the X, Y, Z values of your detection increase / decrease?
- What frame orientation does the april tag use?
- Why are detections from far away prone to error?
- Why may you want to limit the rate of detections?

Useful Resources

- https://pyimagesearch.com/2020/11/02/apriltag-with-python/

Part Two - Lane Following

1. Create a node which can lane follow, use any method of your choosing but using OpenCV is highly recommended. You will need some sort of error / target which you can minimize.
2. Start with a basic proportional controller, how well does it work? What happens when your control loop has a very large error?
3. Try adding the derivative term to your proportional controller and implement a PD control logic, does this help with dealing with large errors?
4. (Optional) Try adding the integral term to your PD controller and implement a PID control logic, does this help with your control?

Exercises

After implementing your PID controller to drive on the right lane do exercise 3.3 - English Driver.

Deliverable 2: Record a video of your bot lane following english driver style

Questions

- What is the error for your PID controller?
- If your proportional controller did not work well alone, what could have caused this?
- Does the D term help your controller logic? Why or why not?
- (Optional) Why or why not was the I term useful for your robot?

Useful Resources

- PID Controller
- Color Masking

- Contours

Part Three - Localization using Sensor Fusion

This part of the exercise is similar to the Unit - B4: Exercises - state estimation and sensor fusion but (*hopefully*) simplified so you don't have to spend as much time doing complicated transforms, there is still a lot of great material in the Duckietown documentation so we ***highly*** recommend reading it.

In the previous lab the robots initial position is considered the initial/fixed frame and we needed us to translate the robots movements into the world frame. The goal now is to have the world frame be the fixed frame with everything relative to this frame. We also want to take prior known information and sensor readings into consideration when making our localization estimates.

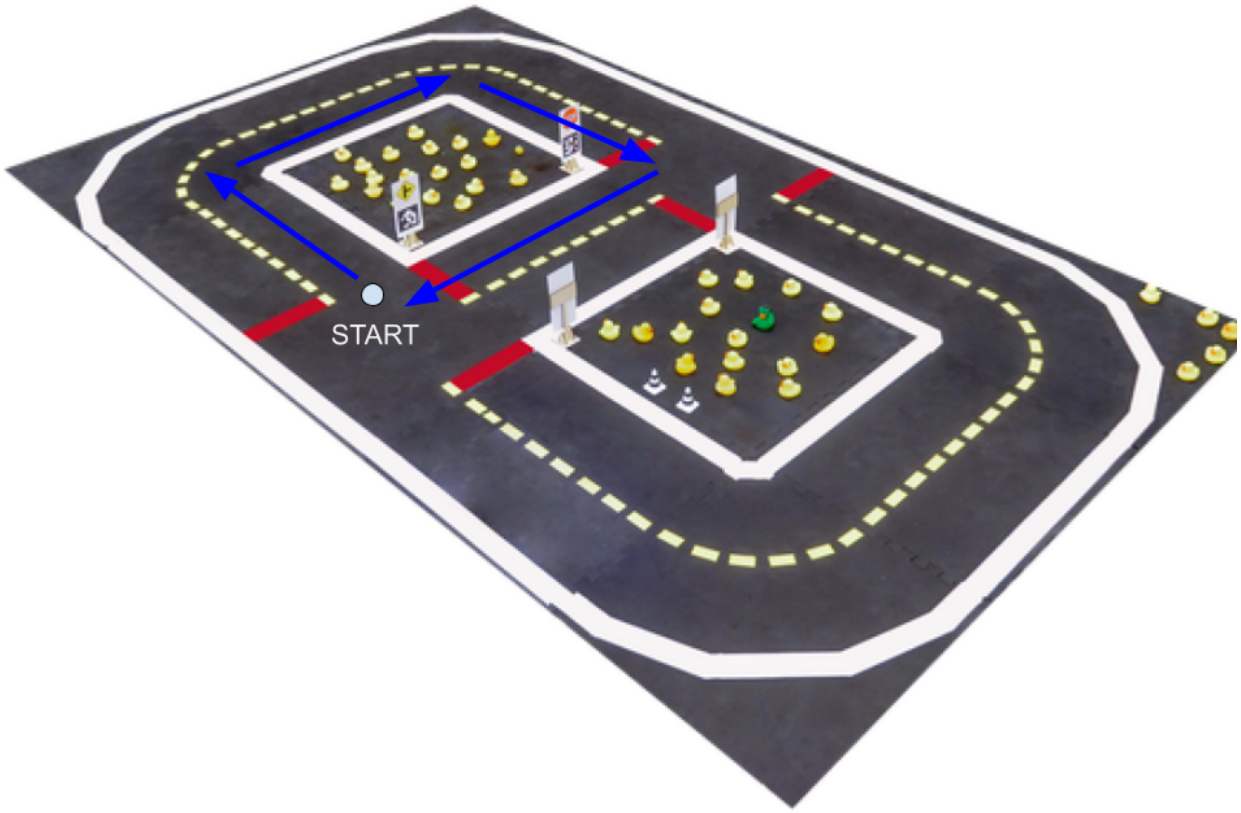
A large part of this section will utilize the TF, Transformations and TF2 (the TF2 online API is currently broken so the source code docstrings is the best documentation) libraries, you are welcome to use either TF or TF2 as you wish, but some of these instructions are based on using TF2.

3.1 Odometry Frame

Read about transformations.
Read the Writing a tf2 broadcaster (Python) tutorial; you do not need to implement this.

Our first goal is to see the robot frame move around in RViz. In order to visualize things in RViz you almost always need a frame/marker.

1. Do Option 1 or Option 2:
    a. (Option 1) If using the exercise template, modify your dead reckoning package so the odometry's parent frame is world and the child frame is odometry.
    b. (Option 2) If using your code from exercise 2 modify your odometry node to use a TF broadcaster to broadcast your odometry frame. Name the parent frame "world" and child frame "odometry"
2. Launch Rviz and set the Fixed Frame
3. Add an Image visualization to RViz that uses your camera feed.
4. Add a TF visualization
5. Use the robots start point as the world frame origin and either manually control or use lane following to do a circle and stop at the origin. You should see the odometry frame move exactly as your robot moves.



Make certain the robot is moving and rotating correctly and note how off the location is from the origin world frame. For an extra experiment circle multiple times.

3.2 Creating Static Landmark Frames

Read #5 From the Writing a TF2 Static Broadcaster tutorial.

We need to start creating the basis / ground truth of where we know things are in the real world. We have placed apriltags in various spots of duckietown, these can act as landmarks that we know the true world frame location of.

1. Measure the location of each apriltag, make note of the origin and the direction that rotations increase and decrease. Follow the right hand rule and make sure you follow the X and Y markers on duckietown as the (0,0) origin.
2. Create a static frame for each apriltag and name them at_<#>_static. Ensure the rotation and height is correct. You do not need to detect the Apriltags, we are simply using them as landmarks.

3. Visualize the frames in RViz using the TF visualization.
4. Update your previous odometry / dead reckoning code to use a topic/service to update its X, Y, Z and Theta. Think of this as teleporting the robot to any location in the world frame.
5. Repeat the previous experiment driving in a circle but this time before you begin, update your robot's starting location in the world frame using the topic/service written previously.
6. (Optional) - Trace your robots path using a marker possibly using this?

Deliverable 3: Record a video on your computer showing RViz: displaying your camera feed, odometry frame and static apriltag frames as it completes one circle. You can drive using manual control or lane following.

Questions

- Where did your odometry seem to drift the most? Why would that be?
- Did adding the landmarks make it easier to understand where and when the odometry drifted?

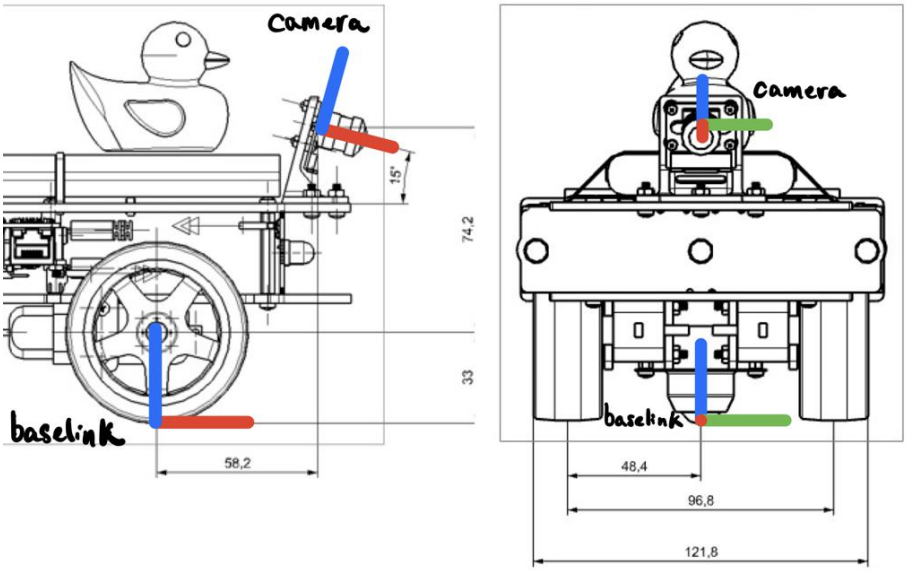
3.3 Creating a Transform Tree Graph

None of this section requires you to write code.

Read #4.1 from Introduction to TF2

One goal for this lab is to get the vector/transform from an apriltag detection to our baselink.

There are at least two transforms in this path, from the apriltag to the camera and the camera to the baselink.



The nice folks at Duckietown provide numerous robot frames for you so you don't have to calculate them yourself (though you certainly could if you wanted to!). All of these frames are created using a URDF (Unified Robotic Description File). Learning more about URDFs is extremely important if you're interested in robot manipulators as it is the framework for calculating inverse kinematics.

Our first goal is to figure out the root parent transform of our robot/environment. We can determine this by generating a transform tree from the transforms that are automatically broadcasted by the Duckiebot. Unfortunately Duckietown did not install the tf2_tools package on our Duckiebots (and the tf1 version has a known bug) so you have three options.

Option A: (if you have ROS natively installed on your computer)

1. Install the `ros-noetic-tf2-tools package`
2. Set your ROS_MASTER_URI (set the variable using export) to your bot's ROS_MASTER_URI
3. Generate the transform graph as in the above TF2 tutorial

Option B:

1. Use Portainer in the dashboard to connect to the console for the `ros` container
2. Install the `ros-noetic-tf2-tools package`
3. Generate the graph and copy it to /data directory so it can be viewed in the file manager

Option C:

1. Find the root frame by reading the Duckiebot URDF here

You can also echo /tf and /tf_static to view all the frames but a graph is a much easier way to visualize them.

Deliverable 4: Attach the generated transform tree graph, what is the root/parent frame?

3.4 Visualizing the Robot Transforms

None of this section requires you to write code.

Trying to visualize a URDF is difficult, we can visualize it best using RViz.

1. Start RViz using the gui tools and set your fixed frame to the root/parent frame from your transform graph.
2. Add a TF display, you will need to modify the scale to make it easier to see.
3. Add a Robot Model display so you can see a render of your duckiebot, lowering the alpha makes it easier to see the frames.

Make note as to where you think the robot's parent frame is physically located. Move the wheels and make note of which joint is moving, what type of joint is this? You can also find this in the URDF.

3.5 Connecting our Odometry Frame to our Robot Frame

You may notice that the wheel frames rotate when you rotate the wheels, but the frames never move from the origin? Even if you launch your odometry node the duckiebot's frames do not move. Why is that?

1. Using a static transform attach your odometry child frame to the parent frame from the URDF.

- What should the translation and rotation be from the odometry child to robot parent frame? In what situation would you have to use something different?
- After creating this link generate a new transform tree graph. What is the new root/parent frame for your environment?
- Can a frame have two parents? What is your reasoning for this?
- Can an environment have more than one parent/root frame?

Deliverable 5: Attach your newly generated transform tree graph, what is the new root/parent frame?

3.6 Visualize Apriltag Detections in RViz

1. Modify your Apriltag detection code so that it broadcasts the transform from the camera to the apriltags estimated location.

Deliverable 6: Record a short video of your robot moving around the world frame with all the robot frames / URDF attached to your moving odometry frame. Show the apriltag detections topic in your camera feed and visualize the apriltag detections frames in rviz.

Questions

- How far off are your detections from the static ground truth.
- What are two factors that could cause this error?

3.7 Calculating Transform from Apriltag to Robot Base & Applying to Static Frame

1. Get the transform from the apriltag location to your wheelbase in world frame
2. Apply this transformation to the known location of the apriltag, this is your robot location.
3. Teleport your robot to the position and rotation calculated by the transform in step 2

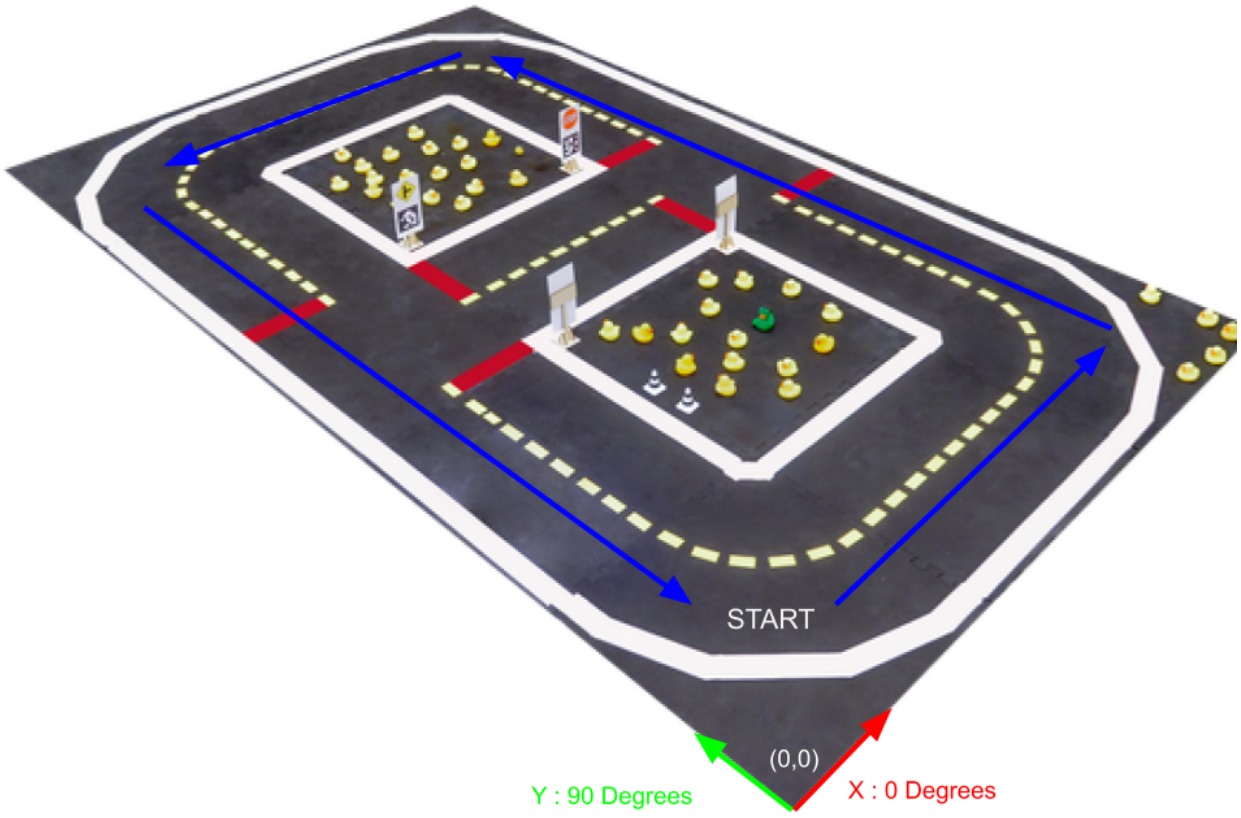
Hints:

- The TF2 library has a function that will calculate the full transform of frame A (apriltag) to Frame B (wheel base) in Frame C (world frame).
- You can also try to use the homography matrix obtained during extrinsic calibration, but this has not been tested.
- Using homogeneous transforms is another method that could work.

Deliverable 7: Show a short video of your robot moving around the entire world (see image below) using lane following and have your sensor fusion node teleport the robot if an apriltag is found and use odometry if no apriltag is detected. Try to finish as close to your starting point as possible.

Questions

- Is this a perfect system?
- What are the causes for some of the errors?
- What other approaches could you use to improve localization?


START
(0,0)
Y : 90 Degrees
X : 0 Degrees