CMPUT 412: Exercise 2 - Ros Development and Kinematics

Justin Valentine (jvalenti), Sergey Khlynovskiy (khlynovs)

Abstract:

This lab report presents an overview of ROS development and kinematics as applied to a Duckiebot. The report covers the basic concepts of ROS, including the computation graph, nodes, topics, services, and bags. We discuss the various trials and tribulations we faced using ROS for the first time. We conclude with a brief error analysis of the Duckiebot and its wheel encoder odometry.

Understanding ROS:

ROS at the Filesystem Level:

- In ROS, packages serve as the primary method of organizing software. These packages may consist of ROS nodes, libraries dependent on ROS, datasets, configuration files, or any other items that are logically grouped together.
- Repositories, on the other hand, are a group of packages that share a common version control system (VCS). These packages, which use the same VCS, can be released as a whole using the catkin release automation tool "bloom." Often, these repositories are converted from ROS build Stacks. However, it is also possible for a repository to only include one package.

ROS Computation Graph Level:

The Computation Graph is the Peer-to-peer network of ROS processes. Processes work together to process data in ROS.

- **Nodes**: Processes that perform computation in ROS. Nodes are modular in nature, and make up a robot control system
- **Master:** Provides name registration and lookup services to the rest of the computation graph. Essential for nodes to communicate with each other
- **Messages:** Used for communication between nodes. Data structures with typed fields
- Topics: Publish-subscribe mechanism for routing messages. Nodes publish messages to a topic, and other nodes subscribe to the topic to receive the message. Nodes decouple the production of information from its consumption
- **Services:** Used for request-reply interactions in a distributed system. Not appropriate for publish-subscribe communication
- **Bags**: Format for storing and playing back ROS message data.

Using ROS To Interact With The Duckiebot:

Using ROS to communicate with the camera:

The first task we had to solve using ROS on the Duckiebot was creating a subscriber to the camera, that would retrieve info about the camera as well as a compressed image. This can be done accessing the topics CameraInfo, and CompressedImage produced by the node camera_node. We had some troubles

dealing with the compressed image, but in the end we were able to figure it out. The lack of proper documentation can at times make simple tasks quite difficult.

Using ROS to move the Duckiebot:

- **Method 1 - WheelsCmdStamped:** Using the node wheels_driver_node we can publish to the topic wheels_cmd the velocities of each wheel. Using this approach is nice in theory because most of the kinematics equations are written in terms of the respective wheel velocities. Because there are so many sources of error, we found that using the kinematic equations to calculate the robot commands was not very useful by itself (expanded upon more in the analysis section). We found the WheelsCmdStamped from the duckiebot tutorial and then traced back to the topic which uses that message. The following code shows how we first create an object pub_wheel_commands, and then how we use it to publish the wheel commands.

```
self.pub_wheel_commands = rospy.Publisher(
    f'/{self.veh_name}/wheels_driver_node/wheels_cmd',
    WheelsCmdStamped,
    queue_size=1
)
```

```
header = Header()
self.pub_wheel_commands.publish(
    WheelsCmdStamped(
        header=header,
        vel_left=vel_left,
        vel_right=vel_right
    ))
```

- **Method 2 - Twist2DStamped:** Using the node car_cmd_switch_node we can publish both the linear and angular velocity to the topic cmd. This allows us to more easily incorporate rudimentary feedback loops to help correct for errors due to external factors. We found this method because others have mentioned it in class. In the end we end up using this method to send wheel commands. We used 0.4 move forwards, 0.3 and omega= -2.5 to do the circle and omega = 9 or -9 to do the turns. The following code shows how we first create an object pub_twist, and then how we use it to publish the velocity commands.

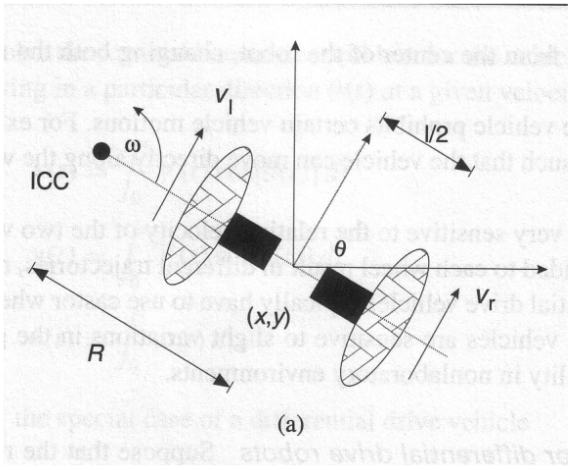
```
self.pub_twist = rospy.Publisher(
    f'/{self.veh_name}/car_cmd_switch_node/cmd',
    Twist2DStamped,
    queue_size=1
)
```

```
twist = Twist2DStamped()
twist.v = .4
twist.omega = 0
self.pub_twist.publish(twist)
```

LEDs & ROS Services:

For controlling the LEDs on the Duckiebot we use two services, the SetCustomLEDPattern service and the ChangePattern service. In the led_controller_node we have a proxy that updates the LEDs on the Duckiebot according to the service SetCustomLEDPattern, and then in our main drive node twist we have a proxy of the ChangePattern serves where we can send simple commands like “red”that will signal a message to be sent using the SetCustomLEDPattern service proxy from the led_controller_node. Because we are using SetCustomLEDPattern we have to run the LED demo before running our code so that it can create the service for us. We struggled for a while trying to wrap our heads around services and how they work, and it was through this code where we figured out how to get them to work.

Differential Drive Kinematics:



The Duickiebot has wheels of radius 0.0318m and a l distance of 0.1m. In the above diagram ω is the angular velocity of the robot, v_l, v_r are the respective velocities of the wheels, IIC is the instantaneous center of curvature and R is the radius of the circle that the robot would trace with the current wheel velocities.

By varying the velocities of the two wheels, we can vary the trajectories that the robot takes. Because the rate of rotation ω about the ICC must be the same for both wheels, we can write the following equations:

$$\begin{aligned} \omega\,(R + l/2) &= V_r \\ \omega\,(R - l/2) &= V_l \end{aligned}$$

At any instance in time we can solve for R and ω using the following equations

$$R = \frac{l}{2}\frac{V_l + V_r}{V_r - V_l};\ \omega = \frac{V_r - V_l}{l};$$

There are three main cases to consider when working with a differential drive robot:

1. $v_r = v_l$ In this case the robot will travel in a straight line
2. $v_r = -v_l$ In this case the robot will rotate on the spot
3. $v_r - v_l = c \neq 0$ In this case the robot will travel along a circular path

The Tick Rabbit Hole:

We use the wheel encoders to determine the distance that each wheel has traveled. Using the following equation we can get the displacement of each wheel:

$$\Delta X = 2\pi R N_{ticks} / N_{total}$$

- ΔX is the distance travelled by each wheel;
- N_{ticks} is the number of ticks measured from each wheel;
- N_{total} is the number of ticks in one full revolution (in our case that's 135).

We use the ticks to calculate both the displacement and the orientation of the robot using the following equations:

$dA = \frac{dl+dr}{2}$

$x_{t+1} = x_t + dA * cos(\theta_t)$

$y_{t+1} = y_t + dA * sin(\theta_t)$

$\theta_{t+1} = \theta_t + \frac{dr-dl}{2*L}$ Where L is the length from the middle of the Duckiebot to center of wheel, r is the radius of the wheels, dl is the linear velocity for the left wheel and dr is for the right. We also immediately converted this information into global coordinates when sampling ticks.

The ticks are a useful measurement because the wheels do not turn consistently, and so using ticks we can be more confident that the robot has done what we have asked it to. Of course ticks are not perfect, we have to worry about things like wheel slippage which ticks will not be able to detect. Also ticks are only sampled at 10Hz so depending on the speed of the robot we could miscount the number of ticks. Also there are only 135 ticks per revolution which is a relatively low resolution with each tick measuring 2.67 degrees of rotation.

Initial frame to global frame transformation:

Using the following homogeneous transform the coordinates of the robot w.r.t the initial frame can be transformed into the coordinates w.r.t the global frame:

$$\begin{pmatrix} cos\left(-\frac{pi}{2}\right) & sin\left(-\frac{pi}{2}\right) & 0 \\ -sin\left(-\frac{pi}{2}\right) & cos\left(-\frac{pi}{2}\right) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -0.32 \\ 0 & 1 & -0.32 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

After this is done we also need to update the orientation of the robot so that it is w.r.t the global frame this can be done by adding $\pi/2$ to it.

ROS Bags:

In ROS, a bag file is a format used for storing ROS message data. These files are named as such due to their ".bag" file extension and play a crucial role in ROS. Bags are usually generated by tools like rosbag, which subscribe to one or multiple ROS topics and record the message sequence as it is received, saving it in a file.

We use rosbags to store the odometry data of the Duckiebot everytime it moves. We found a nice way of creating timestaped ROS bags using a method from Akemi.

Results:

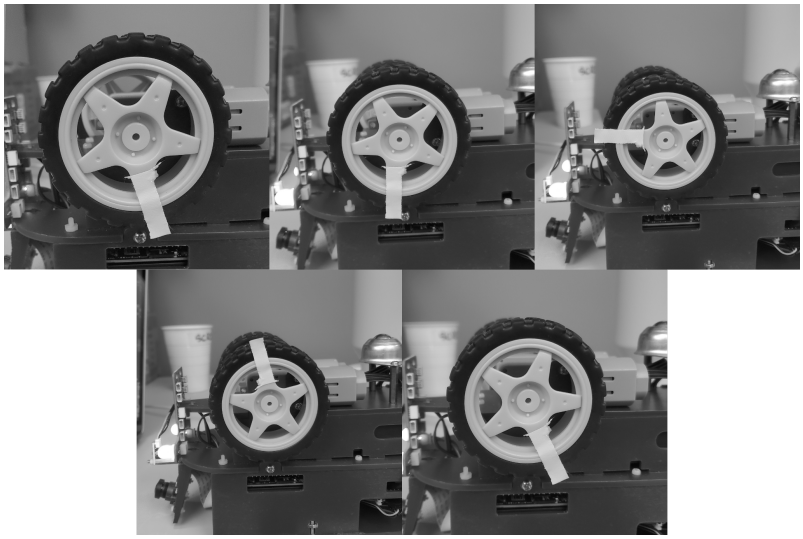
- https://drive.google.com/file/d/1eE6SPFM9ZZUe67sRWITAzKP7JzXBwZTI/view?usp=sharing
- We synchronized the tick collection of each wheel using ApproximateTimeSynchronizer with a queue size of 10 to allow all ticks updates to be sent, and updated the robot and global position when we did so. This is also when we write to the ros bag and update the angles incrementally. At 30 hz we ran a loop with a plethora of if else statements to determine which stage the robot is currently in. First the led is changed using a service then a twist 2d velocity command is sent to move the robot with a queue size of 1 because we often only want the latest command to be considered. We calculated the distance/angle the robot needed to move before the execution of each stage and updated how much we already moved/rotated to know when we need to stop if we fall within a certain tolerance. We have 2 nodes, the led service (led_controller_node.py) and our movement node (movement.py). In our launch file, these nodes are wrapped in a name space group with an argument for vehicle name to be set. Our final robot location was x=1.3930425874587296 y=-0.37083570074357836 theta= 2.464684156496318 which is somewhat close to the actual location but it is still very off.

- This is not the rosbag from the previous run but it is an example of rosbag information

```
serg@serg: ~/Downloads/lab2ducks
Rosbag data: robot_frameTheta data: 4.699543357090012 1676258425426520824
Rosbag data: global_frameX data: 0.323893945018301 1676258425426683187
Rosbag data: global_frameY data: -0.3171546846548059 1676258425426830768
Rosbag data: global_frameTheta data: 0.8492971648666158 1676258425426960468
Rosbag data: robot_frameX data: 0.001867733155939965 1676258425459525108
Rosbag data: robot_frameY data: -0.005005182667787418 1676258425459800004
Rosbag data: robot_frameTheta data: 4.551539436520893 1676258425459950685
Rosbag data: global_frameX data: 0.32500518266778744 1676258425460095643
Rosbag data: global_frameY data: -0.31813226684406004 1676258425460233688
Rosbag data: global_frameTheta data: 0.38432913530418483 1676258425460374116
Rosbag data: robot_frameX data: 0.0004956625535717026 1676258425493142604
Rosbag data: robot_frameY data: -0.005560104565444923 1676258425493414163
Rosbag data: robot_frameTheta data: 4.403535515951774 1676258425493585109
Rosbag data: global_frameX data: 0.32556010456544493 1676258425493753671
Rosbag data: global_frameY data: -0.3195043374464283 1676258425493929624
Rosbag data: global_frameTheta data: 6.202546412921341 1676258425494096994
Rosbag data: robot_frameX data: -0.000979567184542115 1676258425530381679
Rosbag data: robot_frameY data: -0.005440885145182739 4 1676258425530631780
Rosbag data: robot_frameTheta data: 4.255531595382656 1676258425530792713
Rosbag data: global_frameX data: 0.32544088514518277 1676258425530953407
Rosbag data: global_frameY data: -0.3209795671845421 1676258425531099557
Rosbag data: global_frameTheta data: 5.737578383358909 1676258425531239986
Rosbag data: robot_frameX data: -0.002244723187720122 1676258425559763669
Rosbag data: robot_frameY data: -0.004672838051570854 1676258425560019016
Rosbag data: robot_frameTheta data: 4.107527674813537 1676258425560159206
Rosbag data: global_frameX data: 0.3246728380515709 1676258425560296535
Rosbag data: global_frameY data: -0.32224472318772013 1676258425560434579
Rosbag data: global_frameTheta data: 5.272610353796479 1676258425560569286
Rosbag data: robot_frameX data: -0.0034244041501737964 1676258425592589139
Rosbag data: robot_frameY data: -0.002792142983202206 1676258425592842578
Rosbag data: robot_frameTheta data: 3.9743241463013295 1676258425592997789
Rosbag data: global_frameX data: 0.3227921429832022 1676258425593146324
```

Error Analysis:

- In the below set of images we gathered some data on how the wheel movement differs from theory. In each of the images the tape started in an upright position and then the robot wheel was told to move one revolution. As can be seen in the images the robot overshoots the commands dramatically everytime. This is to be expected as there is no active braking and so the wheels keep moving until the resistance of the motor stops their momentum. However perhaps what is more concerning is the amount of variability between trials. This can be seen in our Exercise 2 performance as well, where each run varies dramatically even with encoder feedback.



- The inconsistency between trials led to us spending long nights in the lab hopelessly twerking parameters. This in part is due to the strange Duckiebot phenomenon where the parameters that worked once completely shift and become completely different. We have experienced this many times and have concluded it is due to the low quality of the motors and the low refresh rate of the encoders (10Hz).
- Wheels often stop moving when rotating or slow down causing calculations to be affected significantly.

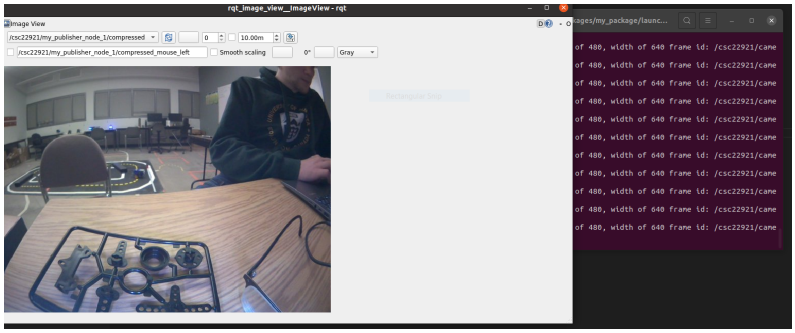
Trials and Tribulations:

- Wheel stopped working: On the csc22921 robot the wheel stopped working randomly which made development much harder. We also had major problems with having our duckiebot getting stuck mid rotation or slowing down. We tried increasing and decreasing the speed but with faster speeds comes more error and slower speeds did not give the motors enough power to turn.
- Long nights at the lab: A lot of time has been spent tweaking parameters for our bots to get them to follow the demo just right, although this is still random and changes with each run. We went on a lot of tangents to find better methods to update robot position such as pose2d and the dead reckoning demo, but we could not get pose2d to give us values and the output from the dead reckoning demo was not as accurate as we would have hoped. Or maybe this was due to a lack of understanding because there were very few resources online to guide us in using these tools.
- Wifi not working: For the first week and a half of the assignment a lot of time was wasted trying to get duckienet to work and until it was actually fixed we could not develop and test our duckiebot properly. csc22921 also disconnected from the wifi frequently and the fix was to just restart the duckiebot.
- Ungodly amounts of error: There is quite a significant amount of delay between when a command is sent to the bot and when the bot actually reacts. Especially for wheel commands this delay can be significantly detrimental to performance for fast and short tasks such as wheel rotation. To account for this we add a 0.3 radian tolerance to when the robot should actually stop when rotating. Then a correction is sent to the bot for a duration of hz =np.clip([int(1/np.abs(ang))], 2, 10), so we scale by the amount we over or undershot by. The farther we are from the goal, the longer the duration we send a correction speed.
- One of the Duckiebots likes to use 95%+ of its ram at all times which prevents you from being able to build on it.
- We used the on_shutdown method of the class to stop when a control+c command is sent which shuts down the nodes properly. When it is, the rosbag is closed, a few 0 velocity commands to stop the bot are sent. We send it multiple times because sometimes these commands are lost and ignored. The only problem is that lights are not turned off on control+c but we could not figure out how to get this to work.

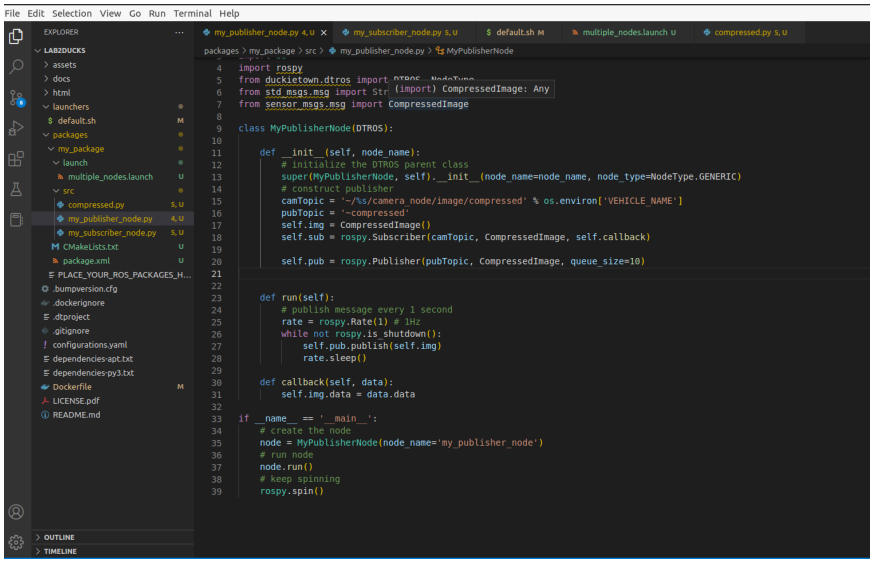
Deliverables:

Your answers to the questions included in the procedures above

- A screenshot of your robot's camera image view in your own customized topic



- A screenshot of your code that shows how you acquire and send images using ros topics



- Rosbag:
https://drive.google.com/file/d/1V331y_0BreoVV2PDyepFMW0szpxRNIZe/view?usp=sharing
- A video that captures the print/plot of your odometry information stored in the bag file

References:

- Robot kinematics: https://www.cs.columbia.edu/~allen/F17/NOTES/icckinematics.pdf
- LED node: https://github.com/anna-ssi/mobile-robotics/blob/50d0b24eab13eb32d92fa83273a05564ca4dd8ef/assignment2/src/led_node.py
- Odometry: https://docs.duckietown.org/daffy/duckietown-robotics-development/out/odometry_modeling.html
- Bag file naming: https://codeberg.org/akemi/duckietown/src/commit/70507322806ae0ff4e39fcbfa4bada3a7328a179/lab2/heartbeat-ros/packages/odometry_node/src/odometry_publisher_node.py
- ROS/Concepts: http://wiki.ros.org/ROS/Concepts
- Template: https://github.com/duckietown/template-ros
- Development in the Duckietown Infrastructure: https://docs.duckietown.org/daffy/duckietown-robotics-development/out/dt_infrastructure.html
- Odometry with Wheel Encoders: https://docs.duckietown.org/daffy/duckietown-robotics-development/out/odometry_modeling.html

- Bag; https://docs.duckietown.org/daffy/duckietown-robotics-development/out/ros_logs.html
- Writing a Simple Service and Client: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29
- Synchronization: http://wiki.ros.org/message_filters