Borja Rojo
CSC232
Howard
Laboratory 1
University of Rochester

Overview

Software Version Control

Here, a student is taught the steps used to create a subversion repository and how to use it. First, a student tunnels through to the University server using the 'ssh' command. Then, they would create a new repository in their secure directory using the 'svn admin' command. This repository can then be used to develop a project from many places by keeping track of changes and ensuring all copies are kept in sync.

To sync a project, one would use the 'svn up' command. To save changes made to files of the project, one would use the 'svn commit' command. To add new files to the project, one would use the 'svn add' command, followed by the name of the file the editor wished to add.

Turtlebot Robot

Here, a student is taught how to use the ROS interface in order to know how to communicate with the Turtlebot.

First, start up commands are issued to begin communication with the Turtlebot, such as 'source /opt/ros/indigo/setup.sh' to source the setup file, 'roscore' to initialize ROS, and 'roslaunch turtlebot_bringup minimal.launch' to run the Turtlebot bringup script.

Then, many commands are taught that allow a student to interface with the Turtlebot. These commands can display current data, such as position and orientation, as well metadata about the communication lines from the Turtlebot. The different communication lines are called 'topics'.

To view a list of all the topics available, the command 'rostopic list' is used. If a student wants further information on a topic, the command 'rostopic info' followed by the topic name is used. For any publishing topic, a student can access the information being communicated with the computer with the command 'rostopic echo' followed by the topic name.

Using C++ code interacts with these topics, which is done by using the appropriate header files and objects depending on the message type. To find out what message type a certain topic is publishing or subscribing to, the command 'rostopic info' is used. To learn more about a message type, 'rosmsg show' followed by the message type is used.

Robot Operating System

This section of the laboratory dealt with creating the files that will be used to build and code the programs that run the Turtlebot. The first step introduced was creating the CMake file that links and builds the executable. The language used here has an incredible amount on nuance and customizability. The specifics are available online. The two types of files this CMake file links are the gengetopt files and the C++ files.

The gengetopt files are used to create customized command line argument commands that is difference from the default C++ format. The C++ files are where the code is programmed that commands the Turtlebot to perform different tasks.

Two different programs are made to communicate with the Turtlebot. There is the listener and the speaker, or more specifically the publisher and subscriber. The publisher is used to tell the Turtlebot what to do. This is done by properly adding the correct header files that allow the Publisher object to access the corresponding fields. For example, the topic used for sound is '/mobile_base/commands/sound'. Using the 'rostopic info' command, it can be found that the message type of this topic is 'kobuki_msgs/Sound'. This then has a corresponding header file in the form 'kobuki_msgs/Sound.h', so the line '#include kobuki_msgs/Sound.h' must be used at the beginning of the publisher program sound commands were to be sent to the Turtlebot successfully. The Publisher object then uses the message type and topic to communicate with the Turtlebot. This is instantiated using the NodeHandle object's method, returning a Publisher object using this function: <Nodehandle>.advertise("/mobile_base/commands/sound", 1). This Publisher uses the publish() function, which takes in a message type as a parameter, to communicate commands to the robot. A message can be created from the message object found earlier and can have multiple aspects to it, each of which can be set and then used to control the Turtlebot.

The publisher file also uses some ROS functions, such as spinOnce() and the Rate object to use the sleep() function. spinOnce() tells the Turtlebot to go through one update iteration, while the sleep() function tells the Turtlebot to hold off on updating for the remaining duration of a previously set time. This was set to the Rate object when it was instantiated.

The subscriber is very similar to the publisher. All required header files are needed to interact with the corresponding publishing topics. The Subscriber object is slightly different. It has less parameters, but also has a callback function parameter. This the API of the callback function has it's own message parameter that then manipulates and allows access to the information taken from the Turtlebot. The subscriber also uses the ROS spin() function, used to querry the Turtlebot as quickly as it can be, acquiring and handling data in the callback function correspondingly.

Open Loop Motion Control

This section of the laboratory dealt with experimentally commanding the Turtlebot and recording the readings of the Turtlebot. Linear velocity, v_x, angular velocity, w_z, and time, t, are all published to and the /odom topic is subscribed to. From the /odom, the position (x and y), the yaw, the linear velocity, and the angular velocity of the Turtlebot are all graphed below for the corresponding open- loop motion control sequences.

Questions

Q1)

1.

$$x(t) = .25t$$
$$y(t) = 0t$$
$$\psi(t) = 0t$$
$$v_x(t) = .25$$
$$\omega_z(t) = 0$$

2.

$$x(t) = \sin(t)$$
$$y(t) = 1 - \cos(t)$$
$$\psi(t) = t$$
$$v_x(t) = .25$$
$$\omega_z(t) = .25$$

3.

$$x(t) = \sin(t)$$
$$y(t) = \cos(t) - 1$$
$$\psi(t) = -t$$
$$v_x(t) = .25$$
$$\omega_z(t) = -.25$$

4.

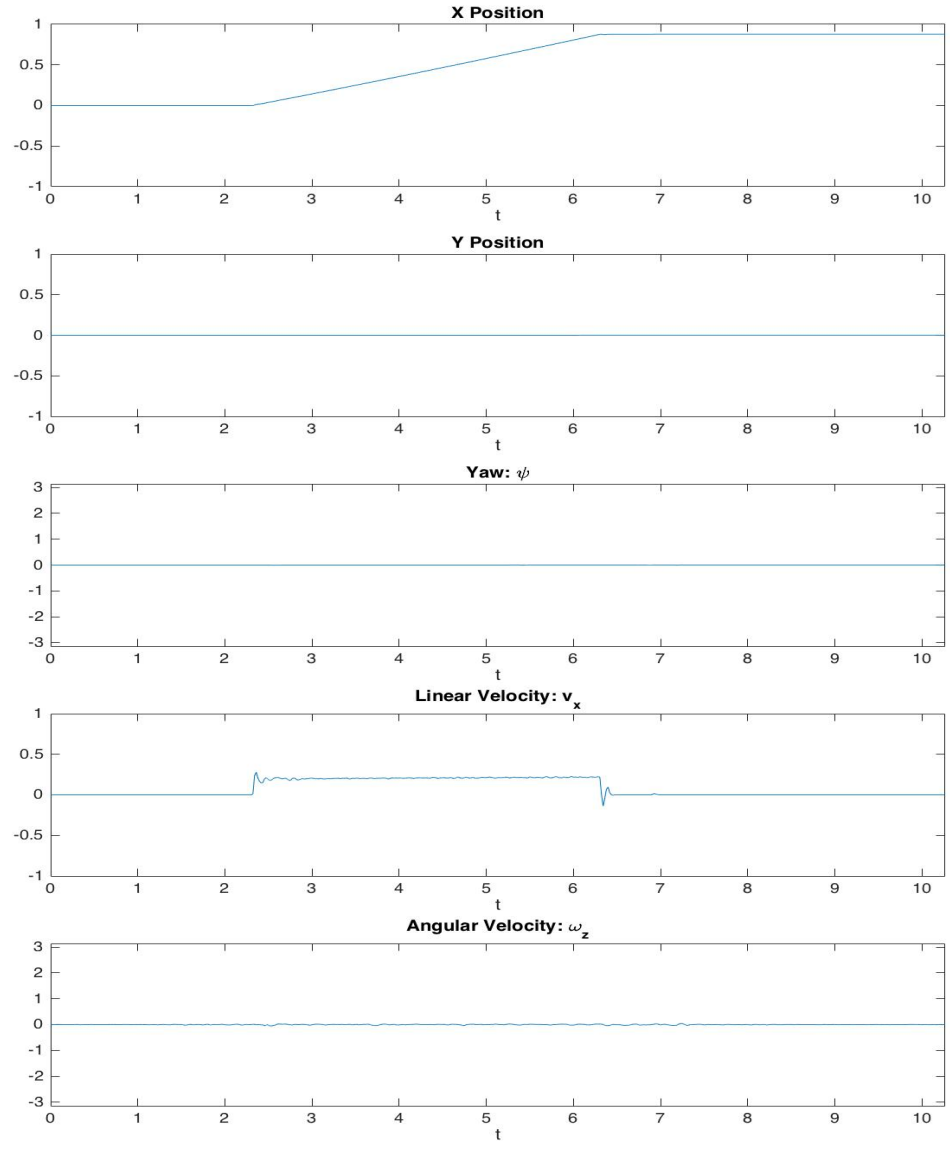
$$x(t) = .25\sin(t)$$
$$y(t) = 0$$
$$\psi(t) = 0$$
$$v_x(t) = .25\sin(t)$$
$$\omega_z(t) = 0t$$

5.

$$x(t) = 0$$
$$y(t) = 0$$
$$\psi(t) = -.25\cos(t)$$
$$v_x(t) = 0t$$
$$\omega_z(t) = .25\sin(t)$$

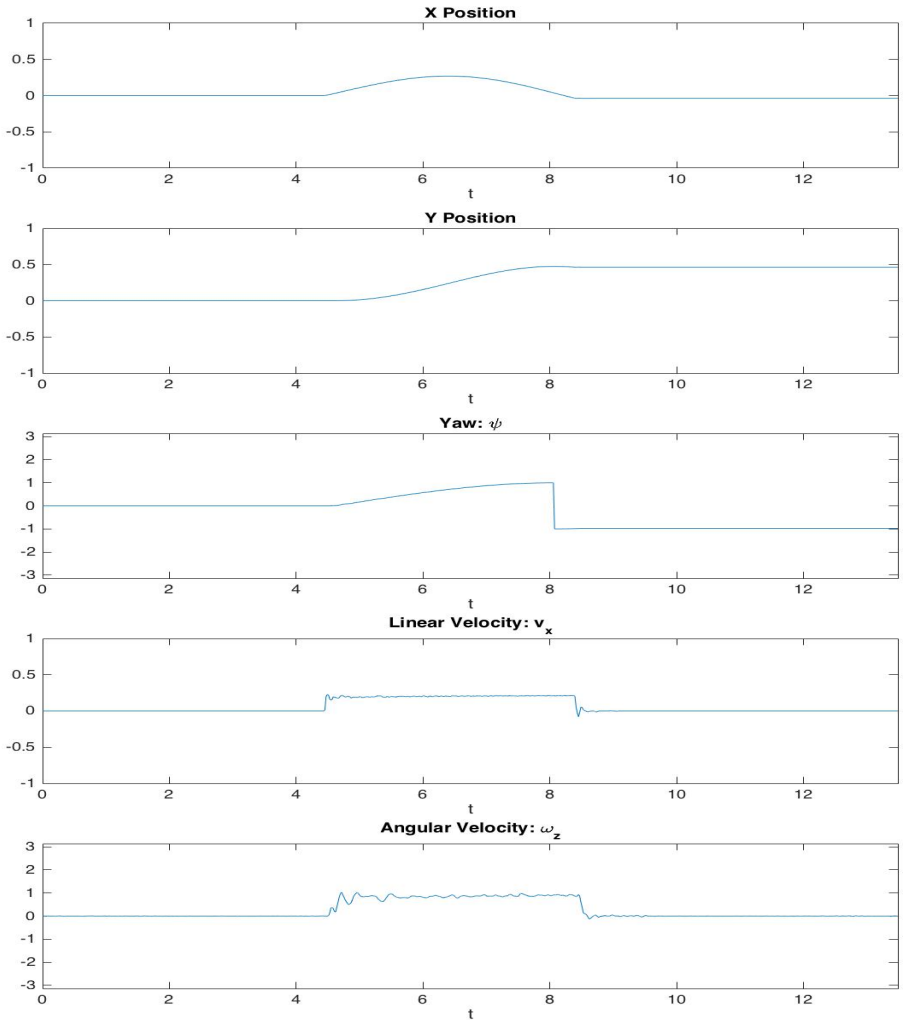
Q2)

Sequence 1



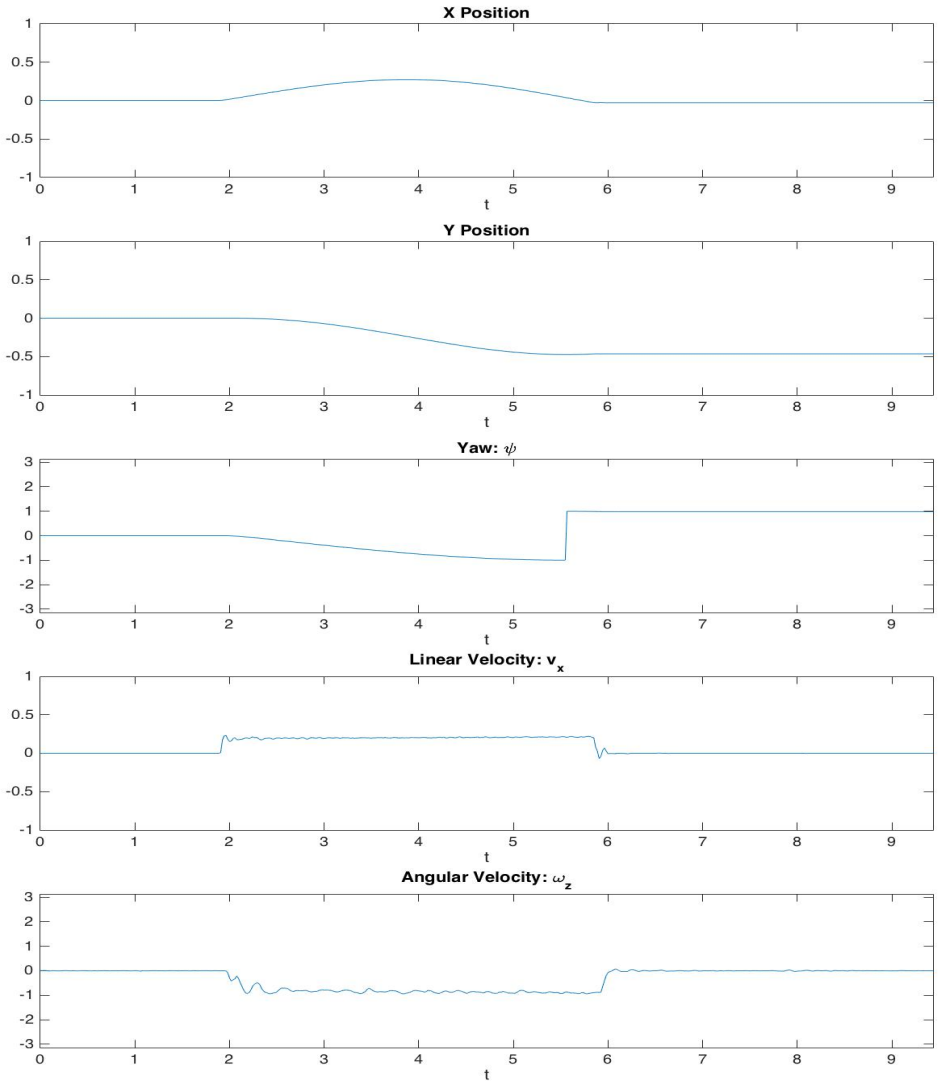
This plot is very representative of the predicted outcome. There is a constant increase in the x position, no increase in the y position, no change in the yaw, a constant linear velocity, and no angular velocity. This is predicted due to the nature of having only a linear velocity published, causing only the x position and linear velocity to change at all, and the x position to be the integral of the linear velocity, demonstrating basic kinematics.

Sequence 2



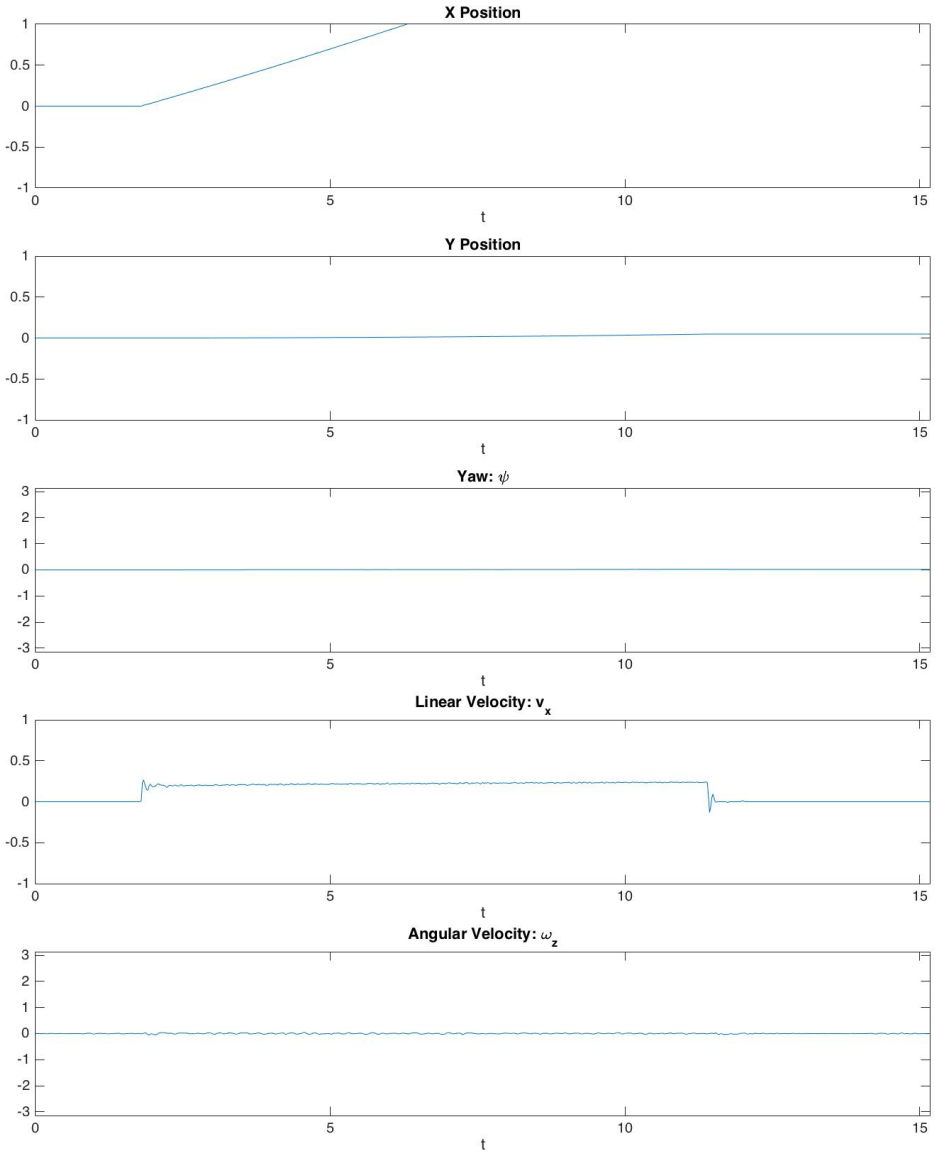
This sequence also matches dick predicted results. The movement of the Turtlebot is in a circle, with a constant a linear and angular velocity. This means that the x and y component will have a sinusoidal curve, where x is represented by sin(t) and y is represented by cos(t). The yaw is continuously changing because it is spinning in a circle. The bounds on the yaw component were not accounted for in dick model, so the quick spike clearly represents having moved a full half turn. The linear and angular velocities match dick results, and easily so as they are set by the experimentor.

Sequence 3



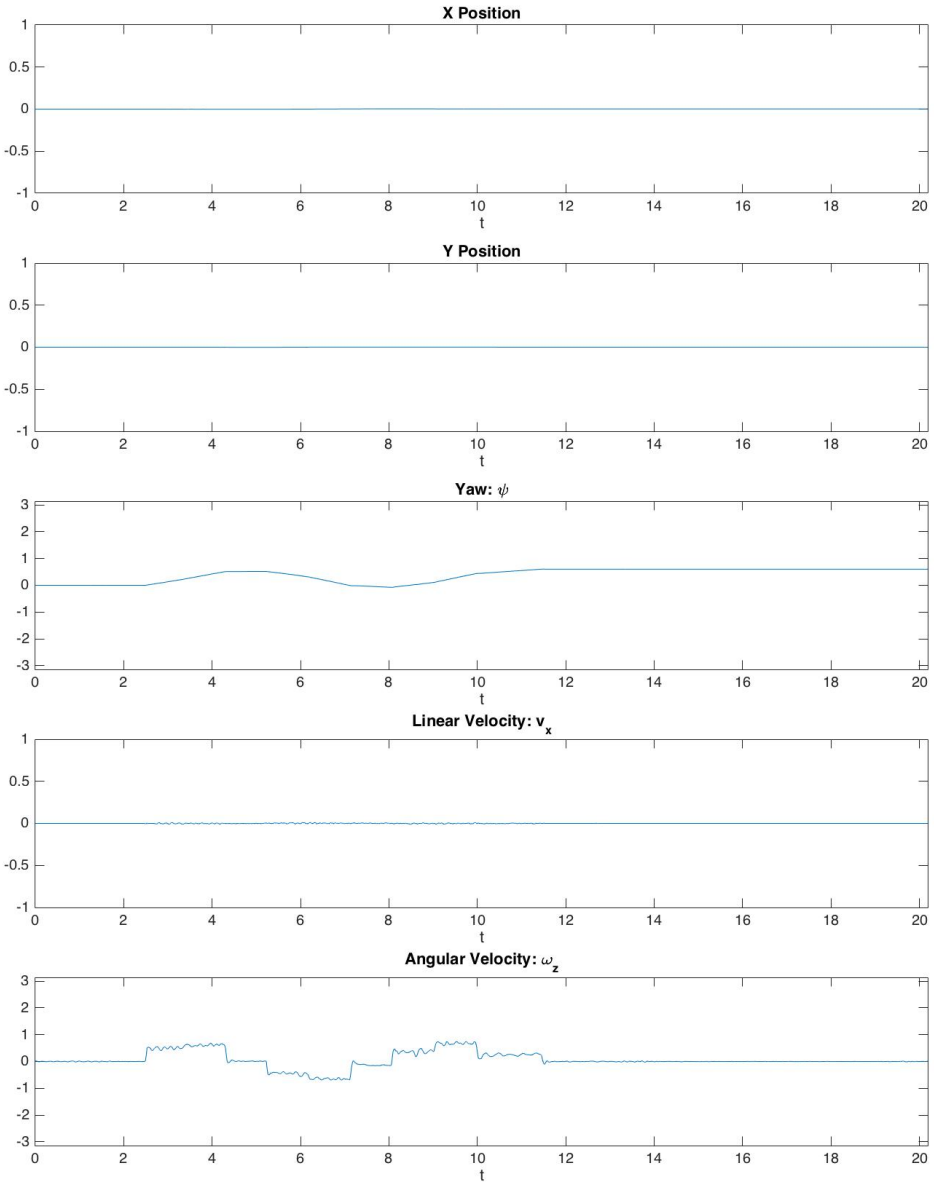
These experimental results also match the predicted results. Due to the negative value of the angular velocity, all corresponding motion was accounted for. The Turtlebot should have equal bot opposite angular components for velocity and yaw. Also, the Turtlebot turns the opposite way then the last sequence, so the y component of position is also equal, but opposite.

Sequence 4



These results do not match the predicted results. This is due to an error in the coding. The sinusoidal component was not added to the linear velocity, so therefore there was no sinusoidal data in the experimental linear velocity and the x position. There was no angular velocity published, so the y, yaw, and angular velocity components did not change, which did match my results.

Sequence 5



These results did not match the predicted results. This is due to an error code. In the sinusoidal component, a multiplicative factor was introduced in the form of integer division. This causes the internal sine argument to step, as opposed to update smoothly. The rest of the predictions match.

Q3)
Topid: mobile_base/events/bumper
Message type: kobuki_msgs::BumperEvent