Lab 5: Dead Reckoning

Learning Outcomes

- Understand Physics of robot movement
- Learn how ROS represents robot position/orientation
- Implement Dead Reckoning with encoder data
- Perform basic navigation to goal points

Section 1: Requirements and Design

In this lab, you will use Jet's encoder data to estimate the velocity, position, and orientation of Jet. Then you will use this information to navigate the robot towards a goal.

Section 2: Installing the Lab

To perform this lab, you will need to get the lab5_dead_reckoning template into your catkin workspace. First ensure that your Jet has internet access by connecting it using WiFi or ethernet. Next ssh into Jet and enter the following command:

```
wget http://instructor-url/lab5_dead_reckoning/lab5_dead_reckoning-code.zip
```

Where the url should be replaced by the URL provided by your instructor. Now unzip the lab:

```
unzip lab8_line_follower-code.zip -d ~/catkin_ws/src/jetlabs/lab5_dead_reckoning
```

Delete the zip file:

```
rm lab5_dead_reckoning-code.zip
```

To build the code, use the following command when you are in `~/catkin_ws/`:

```
catkin_make --pkg lab5_dead_reckoning && source devel/setup.sh
```

To run the system, execute the following

```
roslaunch lab5_dead_reckoning lab5.launch
```

Section 3: Odometry in ROS

Odometry is the relative position, orientation, and velocity of the robot with respect to an origin. In our case, the origin is wherever the robot is when you initialize ROS with `roslaunch`, but we could adapt the origin by specifying an offset at startup. In this lab, you will write code that produces the odometry of the robot based on the robot's sensors.

In ROS, there is an Odometry message that contains the following fields:

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

The final component should be familiar to you: `Twist` is the same type of message as the velocity topic that we use to publish to the motors. `TwistWithCovariance` includes a covariance matrix that specifies the uncertainty of the velocity measurements. The `PoseWithCovariance` type has a covariance matrix to identify the uncertainty, and a `Pose` message. The `Pose` type contains the `position` (in x, y, and z coordinates) as well as the `orientation` (in x, y, z, w).

You can read the official documentation for the Odometry message here: http://docs.ros.org/kinetic/api/nav_msgs/html/msg/C

Section 4: Calculating Odometry

Open the lab5 source code folder (`catkin_ws/jetlabs/lab5_dead_reckoning/src`). The first objective is to publish a topic called `myodom`. The calculation and publishing of the odometry will occur in the `publishOdom` method in the `DeadReckoning` class. In order to calculate the odometry, you will need to maintain the state of the robot in several class fields. You can add these variable declarations to the class definition and initialize them in the constructor.

What state should be maintained? The current value of the encoders is already maintained for you in `left_count` and `right_count`. You will need to know the difference in time between updates, so you should store the previous time in a class field. Additionally, you need to maintain the current x_pos, y_pos, and angle of the robot.

The code at the end of the `publishOdom` method provides a hint as to what variables you should use. Feel free to modify the odometry message publishing if you devise a different way of tracking the state of the robot. Below are the equations that you can use to calculate x_pos, y_pos, and angle.

```
x_pos = (v_r + v_l) * (r / 2) * cos ( angle ) + x_pos_0
y_pos = (v_r + v_l) * (r / 2) * sin ( angle ) + y_pos_0
angle = r * (v_r + v_l) / l + angle_0
```

- v_r is number of encoder ticks since last reading for right wheel
- v_l is number of encoder ticks since last reading for left wheel
- r is the radius of the wheel
- l is the separation of the wheels

Once you have implemented your solution, compare your reading against the `odom` topic. Use `ROS_INFO` to debug your code. You may find that after refinements you can more accurately measure the position of the robot than the built-in `odom` topic! Use a tape measure to help you calibrate your solution; add multipliers to the velocities or position updates to achieve the best results possible.

Section 5: Moving to Goals

Now that you can relatively accurately determine where the robot is located, you can use this data to navigate to goal positions. Let's implement this functionality in the `move` function. When a message is sent to the `/dead_reckoning/goal` topic, the variable `navigating` is set to true and the variable `goal` is set to the goal `Pose`. In this lab we will only worry about getting to the x and y goal location and not the orientation.

Within the `move` method, first write the code necessary to compute what direction the goal Pose is from the current `Pose` of the robot. This direction should be in radians and between -PI and +PI. Use `ROS_INFO` to display the direction. You can send goal messages to the node with the following command:

```
rostopic pub /dead_reckoning/goal geometry_msgs/Pose '{position: {x: 5.0, y: 5.0, z: 0.0}, orientation:
```

After you calculate the angle between the current pose and goal pose, you should use `vel_pub` to publish velocity messages until the robot is pointing towards the goal pose. After you have turned the robot towards the goal, publish velocity messages that make the robot drive forward until it reaches the destination. Since your measurements are noisy, you should specify a threshold value that determine when you stop moving once you are within the threshold of the destination.

Thoroughly test your design using many different goals.

Section 6: Challenge

With the current system, only the encoders are used to determine the location of the robot. Try incorporating the gyroscope or accelerometer to refine the position and orientation estimates. Hint: the gyroscope can be used to accurately estimate changes in orientation, so you could use encoders for estimating linear velocity and the gyroscope for angular velocity.

Can Jet adapt to changing goals? Publish a goal for Jet and then quickly send a different goal before Jet is able to reach the first destination. Modify your code if necessary in order to make sure Jet can smoothly switch between these goals.