Lab 4: Odometry

In this lab we will design and implement an Extended Kalman Filter (EKF) to track the robot odometry. As we learned in class, odometry is used to track robot position by using internal sensors only, without relying on external references or fiducials.

Odometry is simple to implement, it relies on cheap and readily available sensors, and has low computational requirements. The downside is that the pose estimate based on odometry drifts over time. Nevertheless, the odometry is very useful for pose prediction, interpolation, and extrapolation if the time interval of interest is bounded.

The human analogy of odometry is walking with eyes closed and still being able to track our position for a limited time, based only on our inner feeling of steps covered and internal feeling of our body posture.

Principle of Operation

To implement the odometry, we will use the gyroscope which can tell us robot's angular velocity, wheel encoders, which can tell us wheel angular velocity, which in turn are a function of robot angular and linear velocity, and the motion command that the robot receives, which can be transformed into the body motion if we have the model of robot's dynamics.

We will use a simplified first-order model for robot dynamics and we will only rely on the z-axis of the gyroscope. Using additional sensors and a more sophisticated model will result in better odometry, but the one we will implement is already quite robust and usable in real systems, and it also exposes us to all steps needed to design and implement the filter.

The general principle of operation is the following: The robot receives the motion command which contains the requested linear and angular velocity. This command is used by the motion controller inside the robot as an input to the loop that controls the motors. We will also use it in the prediction step of the Extended Kalman Filter. In other words, our odometry system will look at the received commands, apply the dynamics model, and predict the future robot velocity and position. The measurement (correction) step of our EKF is based on reading the angular velocity of the robot body as measured by the gyroscope and angular velocity of each wheel. The prediction model will be based on a simplified dynamic model, whereas the measurement model will be a direct application of robot kinematics.

To design and implement your filter, you will have to formulate the model and do some measurements to identify the model parameters. From there, it is the straightforward application of the EKF algorithm as we learned in class.

Ground Truth Extraction

To characterize the robot during the model design stage and to evaluate the filter during the testing stage, we need some way of knowing the actual location and velocity of the robot. This is commonly referred to as the “ground truth”. In a physical world (real robot), ground truth is never available, and the only way to characterize and test the robot is to use measurements from highly accurate and precise sensors that are only available in laboratory conditions.

In simulation, ground truth is readily available because the simulator constructs the world, so extracting the ground truth is the matter of accessing the variables of interest in the simulation state. For our purpose, we need the robot position and robot body velocity. In Gazebo, we can use `/gazebo/link_states` topic. This topic publishes the pose and velocity of every link that is subject to motion. Start the empty world simulation and examine this topic using `rostopic` command:

```
ros2 launch prob_rob_labs turtlebot3_empty_launch.py
ros2 topic echo /gazebo/link_states
```

You should be able to easily figure out the fields in this topic. The link of interest (robot body) is called `waffle_pi::base_footprint` in Gazebo.

Assignment 1: Write a ROS node that subscribes to `/gazebo/link_states` topic, extracts the pose and velocity of the `base_footprint` frame and publishes the extracted information to `/tb3/ground_truth/pose` and `/tb3/ground_truth/twist` topics. The types of the two topics should be `PoseStamped` and `TwistStamped` (both available in `geometry_msgs` package). Make the frame of reference a command-line parameter to the launch file. Submit the code.

Assignment 2: Make the robot move by publishing to /`cmd_vel` topic and show that it is achieving the requested velocity. Use `rqt_plot` tool to visualize the velocities. Use the `rviz2` tool to visualize the ground-truth pose and submit a short video capture of your screen that proves that your publisher is working. In this lab, we will use `odom` frame as the frame of reference when publishing the pose, so launch the ground-truth publisher with the appropriate command line parameter.

Robot Dynamics

When the robot receives the motion command it takes finite time to accelerate (or decelerate) to the target velocity. To accurately model this behavior, we would need to know about the robot dynamics, such as the mass, moment of inertia, motor models, tire-road interaction model, etc. For the purpose of this exercise, we will use a simple first-order model which ramps up and down following the damped exponential law. We will assume that the commanded and achieved velocity are governed by the following difference equation:

$$v[n+1] = av[n] + G \cdot (1-a)u[n],$$

where a is the forgetting factor of the first-order filter and G is the input gain. The former models the slope at which the output velocity follows the commanded velocity, whereas the latter models the difference between the achieved and commanded terminal velocity. The step response of the above system is

$$v[n] = G \cdot (1 - a^n)$$

To determine the model parameters, we can send the velocity command and observe the value at which the output settles. That is the input gain. Next, we can find the sample at which the output settles at 90% the terminal velocity. We have

$$0.9 = (1 - a^n)$$

$$a = 0.1^{\frac{1}{n}}$$

Instead of counting samples, we can measure the 90% ramp-up time τ and express the parameter a in terms of the period between samples Δt.

$$a = 0.1^{\frac{\Delta t}{\tau}}$$

The above representation is useful because in a practical system the time between samples Δt will have slight variations, so we can adjust the parameter on each iteration of the filter.

Assignment 3: Step the linear velocity by publishing to `cmd_vel` topic and observe the actual velocity ramp-up by plotting the `/tb3/ground_truth/twist` topic in `rqt_plot` tool. Capture the screen, annotate relevant points on the plot and determine the parameters as a function of Δt. Submit the plot and your calculations. Repeat the process for angular velocity.

Note that, because Turtlebot is a ground robot with differential drivetrain, only the linear velocity along x-axis and angular velocity around z-axis are of interest..

Robot Motion

The dynamic model above relates the system input (robot command) with robot angular and linear velocity (two variables that will be part of the system state). The odometry must also track robot position, so we need to add two more variables to the system state: x, y, and θ, which

collectively describe the 2D pose in the odometry frame of reference. Straightforward application of robot kinematics will yield three equations that relate angular and linear velocity with the pose.

Starting from the matrix-multiplication form of 2D motion in the local frame of reference that we saw in class

$$\begin{bmatrix} R(\theta) & p(x,y) \\ 0 & 1 \end{bmatrix}_n = \begin{bmatrix} R(\theta) & p(x,y) \\ 0 & 1 \end{bmatrix}_{n-1} \cdot \begin{bmatrix} R(\omega\Delta t) & v(x,y)\Delta t \\ 0 & 1 \end{bmatrix}$$

we can expand it into three equations that express the future pose as a function of present pose and velocity. Note that for the skid-steer drivetrain we are using here, the lateral velocity is always zero

$$x[n] = x[n-1] + v_x \Delta t \cos\theta[n-1]$$
$$y[n] = y[n-1] + v_x \Delta t \sin\theta[n-1]$$
$$\theta[n] = \theta[n-1] + \omega\Delta t$$

Assignment 4: Let $[\theta,\ x,\ y,\ v,\ \omega]^T$ be the state vector and let $\begin{bmatrix} u_v & u_\omega \end{bmatrix}$ be the robot command. Combine the above model with the result from Assignment 3 to formulate the state-update law for the Extended Kalman filter. Hint: the state transformation will contain multiplications with trigonometric functions, so it will be non-linear (A-matrix does not exist, but instead we will have a nonlinear function of state variables), but the input transformation will be linear, so B-matrix will exist. Also, derive the Jacobian for state transformation.

Measurement Model

Robot-dynamics model along with the motion model gave us the state update model for the Kalman filter. We now focus on the measurement model. For differential drivetrain the angular velocities of the wheels are related to body velocity as

$$v = \frac{r_w}{2}(\omega_r + \omega_l)$$

$$\omega = \frac{r_w}{2R}(\omega_r - \omega_l)$$

where r_w is the wheel radius and R is the robot radius (half of the wheel-separation). The sensors available to us are the gyroscope which will measure the body-angular velocity ω

directly, and two encoders, one for each wheel. Encoders associated with the wheel provide measurements of ω_r and ω_l.

Assignment 5: Let $z = \begin{bmatrix} \omega_r & \omega_l & \omega_g \end{bmatrix}$ be the measurement vector consisting of encoder measurements for right and left wheel, as well as the gyroscope measurement. Use the solution of the above kinematics equations to formulate the measurement model. Hint: the model will be linear, so you should be able to write out the C-matrix and express the model as matrix multiplication. Express the model in terms of robot radius and wheel radius. For Turtlebot 3, the wheel radius is 33mm and wheel separation is 143.5mm.

Accessing Measurements

Now that we have the model, the last question is where we will get the measurements from. Turtlebot simulation models the Inertial Measurement Unit (IMU) which consists of a gyroscope, magnetometer, and accelerometer and publishes the measurements along with the covariance on `/imu` topic. The simulation does not model the encoders explicitly, but you can access the state of joints that model the wheel axle on `/joint_states` topic. Unfortunately the covariance is not available, so you should make one up. PIck a number that is a reasonable guess. In a more advanced system you would do a set of experiments to characterize covariance but for the purpose of this lab, just setting a reasonable constant value is fine. If typical measurements are in the tens of rad/sec what would be a reasonable guess for variance?

Implementation

We now have all the information needed to design the odometry tracking system based on the Extended Kalman Filter. There are a couple of practical considerations to be aware of.

1) For successful sensor fusion, measurements must be aligned in time. ROS provides the facility to do that based on measurement timestamps. Before implementing the filter, please read about the `ApproximateTimeSynchronizer` class in the `message_filters` module. This class allows you to create multiple subscribers and get a single callback containing measurements that are aligned in time based on the timestamp. The timestamp in all messages passed to the callback will be within the tolerance specified by the `slop` parameter. Take advantage of the synchronizer class.
2) Turtlebot `/cmd_vel` topic is “sticky”, that is, the last received message on this topic will be the velocity at which the robot will continue to move until it receives the next message. So the easiest way to implement the odometry is to simply record the present velocity in /cmd_vel subscriber and use it in the synchronized callback that subscribes to sensor topics.

3) In topics that represent sensor readings, the timestamp reflects the time when the sensors were sampled. Likewise, topics that represent processed information (e.g. output of your Kalman filter) the timestamp should reflect the source of information used to generate the output, not the time when the output was calculated (the latter would include communication time and processing time which has no mathematical meaning from signal-processing perspective).
4) The IMU and encoder topics are sampled at different rates and they operate asynchronously of each other, so you should pick the value of the `slop` parameter accordingly. If you pick the value that is too small, you might end up with a message from one sensor that has no message from another sensor to pair with. If you pick the value too large, you may end up with unnecessary misalignment of messages.
5) In `Odometry` topic, the orientation is represented in quaternion form, but your odometry tracker calculates the 2D bearing angle (also known as the yaw angle), which represents the orientation around the z-axis in 3D space. The other two orientations are zero. You must construct the quaternion accordingly.
6) The covariance associated with the `Odometry` topic is a 36-element array that represents a 6x6 matrix in row-major order. There is a separate covariance for twist and pose. The covariance associated with your system state is a 5x5 matrix representing the uncertainty of your state variables. Think carefully about which elements of your state covariance map to which elements of the two published covariance matrices. Clearly, some elements of published covariance matrices will be zero.

Assignment 6: Implement the odometry-tracking node using the model you derived and following the above guidelines. Publish the odometry messages to `/ekf_odom` topic. Submit the code.

Hint: Do not attempt to implement all at once. Implement functions incrementally, test, make sense out of the result, and gradually add new functionality. Use plenty of logging to track what your code is doing. If you try to implement the whole system at once before testing, you will end up too much to debug without the idea where to start from. Instead, start small. Implement the subscribers and message synchronizer. Make sure you are getting the messages from topics as expected. Make sense of timestamps and timestamp deltas. Then implement the core math functions. Do not bundle everything in one function, break the functionality up in smaller functions that are called from a higher-level function. If you are revision-controlling your code with Git, commit in small meaningful increments.

Evaluating the Results

Once you get your filter working it is time to evaluate how well it works. We can compare the filter output with the ground truth.

Assignment 7: Add your odometry pose and ground-truth pose to rviz and make the robot move. Are they moving together with reasonably small error? Record the video of the desktop that shows that your tracker is working. Submit the video.

Hint: ground truth pose is relative to the origin of the fixed frame of reference, where odometry pose is relative to whichever pose the robot had at the time when the odometry tracker node was started. To avoid the biased measurement, make sure you start the odometry tracker node before you move the robot for the first time.

Visualization in RVIZ is good for subjective evaluation, but to be more objective, we should establish some metric.

Assignment 8: Implement a ROS node that subscribes to the ground truth and your odometry topic and calculates the euclidean distance between estimated robot positions and angular difference in robot orientation (be careful about wraps around +/-180 degrees). These metrics quantify the error of your odometry. Use the `rqt_plot` tool to visualize the error and move the robot around for about a minute. Submit the plot.

Assignment 9: Look at the covariance matrix of your output and answer the following. Which elements are stable and which are growing out of bounds? Explain why?