

# Lab 6: EKF Localization – Filter

In this lab we will use the results from the previous lab to fully implement the landmark-based localization based on the Extended Kalman Filter. As we have seen in the previous lab, there is a significant amount of processing that the system needs to do before it even gets to run the Kalman filter. Everything we built in the previous lab will be used here to implement a complete localization system. The Kalman filter alone is only a small part of a much larger system that must be implemented to reliably localize the robot.

## Measurements

In the previous lab, we focused on one landmark and implemented the code that estimates the distance and bearing between the camera frame and the landmark. There are five landmarks in our environment and the measurement branch of the Kalman filter must be able to handle all of them and distinguish them by the color. Each landmark has known x, and y coordinates. A file that lists all the landmarks and associates their color with the coordinates will be our map.

**Assignment 1:** Design the map file that stores the necessary information about the landmark. Common file formats like JSON, YAML, CSV, or XML can be used, but you can pick any format that you see fit. Write the initial ROS node for EKF localization that initializes the node and loads the map. The path to the map file should be a parameter that can be passed to the launch command at startup time. Keep in mind that your code will need to retrieve the landmark coordinates and dimensions given the spotted landmark color, so pick the data structure that makes it easy to perform the retrieval.

Your node will have to subscribe to all cornerpoint topics. When the message comes in, the callback must know which landmark color the message is associated with and retrieve the appropriate landmark information. This step is what the textbook refers to as the correspondence mapping. After processing the corner points using the algorithms developed in the previous lab, your node will calculate the distance and bearing and estimate the associated variances. At this point it will have everything it needs to fuse the measurement with prediction.

**Assignment 2:** Extend the vision processing code from the previous lab to handle all landmarks specified in the map. Avoid having repeated code in your implementation (i.e. writing five callbacks with identical code is bad programming practice and should be avoided). Instead design the callback to receive the landmark color information when it is entered along with the list of corner points.

Hint: The best practice is to stem everything from the map data structure. The map contains the information about landmark colors so you can construct the topic names from the map and instantiate the subscribers. Subscribers can be color-specific object instantiations which are hanging off the main filter object which you pass the color and landmark position to the

constructor and have the callback return or update the common state. Alternatively, you can implement one common class and use lambda operator to capture the color name and landmark coordinates at subscriber-creation time.

The measurement branch of our filter is now ready and the next step is to implement the full filter which uses robot velocity (available on odometry topic) for prediction and fuses the measurements each time the landmark is spotted. The core of your code is a transcription of the algorithm shown in Table 7.2 of your textbook, but there are a few implementation concerns worth highlighting.

First, velocity is reported on odometry topic and it is not synchronized with corner-point topics which are the source of information for measurements. Synchronizing the topics using the `ApproximateTimeSynchronizer` class is not viable, because there may be prolonged time periods when the robot does not have any landmark in sight and the position must be tracked based on odometry only. If we were to use the time synchronizer, there would be no updates during these “blind” periods. When the measurement arrives, the system will only see the last recorded velocity (unless it kept track of all observed velocities since the last measurement), which would result in inaccurate predictions (unless the system integrated all observed velocities since the last measurement).

Second, measurements are arriving on separate topics and each measurement arrival invokes the callback. If multiple measurements are observed at the same time, each would be processed in a separate callback. Hence, there is no explicit for-loop shown in lines 9 through 17. Instead, each measurement arrival is one iteration of the loop that calculates the new posterior as measurements arrive.

Third, the algorithm of Table 7.2 does not explicitly handle the case when angular velocity is zero (applying the motion model verbatim would result in division by zero). In implementation, you must handle this case by applying linear motion to the prior position. For numerical stability, you should consider the motion linear whenever the absolute value of the angular velocity is below some small threshold  $\epsilon$ .

Finally, the system state is the pose of the robot body frame, which is in ROS known as the `base_link`. However, measurement is in the camera-lens mounting frame, known as the `camera_rgb_frame`, which has the same origin as the camera optical frame we used in the previous lab, but it is oriented following the robot positioning convention (x-axis shoots out of the camera, and z-axis points up). We used the optical frame orientation to derive the 3d-space point-to-pixel projection, but for localization we use the camera-lens mounting frame. To correctly fuse the measurement with the predicted state, you will have to transform the predicted state to represent the pose of the camera-lens mounting frame, run the innovation branch of the filter, and then transform back to the robot-body pose.

Arrival of the measurement does not reflect the actual time when the measurement was taken. The same is true for odometry. Your code must, therefore, look at the timestamp to calculate

the elapsed time since the last state update. Putting it all together the implementation should look like this:

- Keep track of the time associated with the system state.
- Initialize the state using the timestamp associated with the first measurement. This message is “sacrificed” only to establish the time reference.
- When the odometry sample arrives, run the prediction branch of the filter (and prediction only) where  $\Delta t$  is the difference between the time associated with the message and the system time. Update the system time with the value of the timestamp.
- When the measurement sample arrives, calculate the  $\Delta t$ . Run the prediction step for the calculated elapsed time (or skip it if the elapsed time is zero), followed by running the innovation step. Use the last known velocity for prediction. Update the system time with the value of the timestamp.
- If you get negative elapsed time, you are dealing with the late-arriving sample (due to out-of-order delivery on independent topics) that cannot be used. Throw it away and leave the system time unchanged.

The implementation above is driven by data arrivals. Notice that between measurements, the system state still progresses and reflects the prediction since the last posterior update. The covariance will inflate during this period because the robot is essentially dead-reckoning. Once the measurement arrives, the posterior is calculated. Simultaneous measurements will be processed in sequence, but the timestamp will still reflect the time of last sensing.

**Assignment 3:** Implement the EKF localization node following the guidelines above and make it publish the estimated pose and covariance to `/ekf_pose` topic.

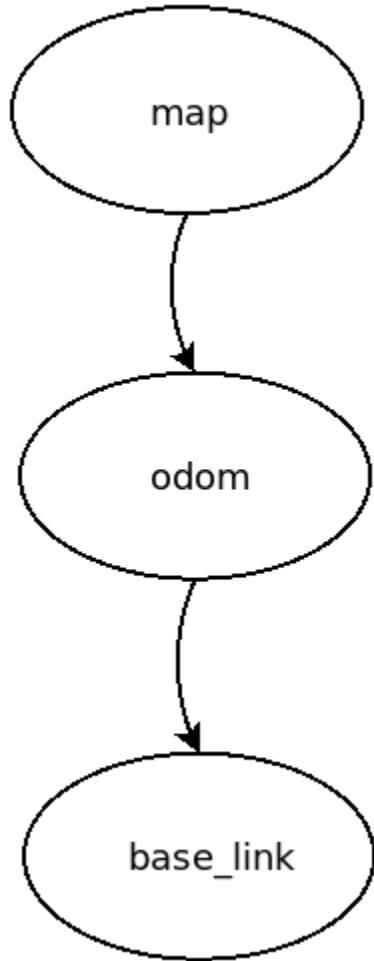
**Assignment 4:** Write the node that subscribes to the ground truth topic (using the ground-truth publisher that you developed in previous labs) and to the output of the EKF localization node, and calculates the position and orientation error. Publish the error to ROS topics. Drive the robot around and plot the errors using `rqt_plot` tool and submit the screenshot of the plot. Visualize the pose in RVIZ and submit the video screencast of the robot moving around in the environment. Make sure you drive the robot away from the landmarks and show the error from odometry accumulate and then turn it towards the landmark and show the error pull back in. This proves that your filter is working properly.

## Final Touches (Transform Tree)

When the robot is localized, it essentially learns the transformation between the global frame (map) and the robot body frame (`base_link`). This transformation is not part of the transform tree (`/tf` and `/tf_static` topics) until the node responsible for the localization publishes it.

Having a complete transform tree is essential for full visualization of all sensors in the global frame and also for reconciling the sensor measurements among different frames of reference. One subtle detail about the transform tree in ROS is that the localization node does not publish the map/`base_link` transform directly. Instead it starts from the localization result which

establishes this transformation, but it does not publish it yet, and reads in the transformation between the odometry (odom) frame and the robot body (odom/base\_link transform). It then works backwards to find the map/odom transform and publishes it. This transform determines the pose of the odometry frame in the map frame. The frame hierarchy looks as shown below.



The map frame is on the top of the hierarchy, and the odom frame hangs off it. The base\_link frame is the robot body frame that hangs off the odometry frame, and all links mounted on the robot hang off the base link frame. Typically “base link” represents the center of gravity of the robot. In some implementations there may be another link called “base footprint” which is a 2D projection of the “base link”. For ground robots, there is little practical difference between the two frames.

Imagine that we perfectly localized the robot at time  $t_0$  and that the odometry was at the origin at that time. If the odometry didn't drift we would not need to localize the robot any more. The accumulated odometry in the “odom” frame on the top that first localization would always yield the perfect knowledge of the robot's pose.

The problem is that the odometry drifts over time so we have to repeat the localization. What do we do when the new localization is calculated and it differs from composition of the old localization and the accumulated odometry. We have two choices. The first is to stop the odometry and the second is to adjust the origin of the odometry frame. Because we would like the odometry frame to be differentiable, we do the former.

**Assignment 5:** Write the ROS node that subscribes to the output of the EKF Localization node and publishes the map-odometry transform on `/tf` topic. Each time the localization node produces a new pose, the node must adjust the published transform. The transform should be published at a 30Hz rate regardless of the frequency at which the localization node calculates the pose and the map/odom transform should change only when the EKF Localization node produces the pose incorporating the new measurement (not if the localization is only doing prediction without measurements). Submit the code, the transform-tree screenshot from `rqt_tf_tree` tool and a video of RVIZ that visualizes all frames of reference moving through the space.