# eInfochips – An Arrow Company

## *Internship Documentation*

***By***

*Akshay Mahesh Laddha, Engineering Intern (Sensor Fusion)*

**Manager:**
*Trevor Bingham*

**Team:**
*Gurpreet Singh*
*Naitik Nakrani*
*Akash Parikh*
*Amit Gupta*

**Internship Completion Date:**
*August 11th, 2022*

# Table of Content

# Introduction

Observing an autonomous vehicle that stops at the sign of red lights shows the rapid response of the vehicle and such a response is generated by the fusion the output from multiple sensors. AV and mobile robots are the latest players in this ecosystem of sensor fusion aiming at basically combining sensors that assists in tracking both stationary and moving objects in order to simulate human intelligence. In the context of sensor fusion, predict equation is the one that predicts the state of the car, and update equation is the one that continuously updates that prediction. The predict equation uses the previous prediction of the state (the range of possible state values calculated from the last round of predict-update equations) along with the motion model to predict the current state. This prediction is then updated (via the update equation) by combining the sensory input with the measurement model. Based on the above explaination, It can be deduced that performing sensor fusion provides us with much more accurate results as opposed to using an individual sensor for simulation. Therefore, the topic of focus will be performing Sensor fusion using ROS based packages on turtlebot3. For the given task, we will be using IMU and odometry sensor and aim to introduce certain type of elements that would present us with a more realistic simulation environment including noise addition to the sensors in any form including weather and environmental noise to surface friction and more.

## State-Space Models

A state-space model is basically a mathematical equation that demonstrates how a robotic system might move from one timestep to the next. It also demonstrates adjustments to the robot's control inputs, such as velocity in meters/second (typically represented by the variable v) that can affect the robot's current position (e.g X,Y coordinate) and orientation (yaw heading) with respect to its environment.

Consider a robot moving around in an x-y coordinate plane, the position and orientation of the robot make up for state vector of the robot –

$$StateVector = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \gamma_{t-1} \end{bmatrix}$$

Here, **X** and **Y** as well as the **yaw** degree are represented in global frame

The yaw angle describes the rotation around z-axis in the counterclockwise direction and is represented in radians

The linear velocity in x and y-direction can be represented as -
$Vx = Vcos(yaw)$
$Vy = Vsin(yaw)$

In order to modify the equation to calculate the state forward for one timestep **dt**, the formula of

$$Distance = velocity * time$$

Which results in the following equation -

$$\begin{bmatrix} x_t \\ y_t \\ \gamma_t \end{bmatrix} = \begin{bmatrix} x_{t-1} + v_{t-1} \cos \gamma_{t-1} * dt \\ y_{t-1} + v_{t-1} \sin \gamma_{t-1} * dt \\ \gamma_{t-1} + \omega_{t-1} * dt \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

The matrix equation above is in non-linear form and to use it for applications like Kalman Filter or Particle Filter, it is converted to linear state-space model form –

$$\mathbf{x_t} = A_{t-1}\mathbf{x_{t-1}} + B_{t-1}\mathbf{u_{t-1}}$$

$\mathbf{X^t}$: Current state vector at timestep **t**
$\mathbf{X^{t-1}}$ : State of the robot at previous time-step **t-1**
$\mathbf{U^{t-1}}$ : represents the control input vector at previous timestep [forward velocity, angular velocity]
**A matrix**: describes how the system's state (x, y, and yaw angle) changes from time t-1 to time t when no control instruction is executed, or when we don't instruct the robot to move at a certain pace (or velocity)
B matrix:

# Sensor Fusion and Algorithms

Combining sensors that track both stationary and moving objects in order to simulate human intelligence. Sensor fusion suggest fusing together signals of multiple sensors to determine the position, trajectory and the speed of an object which help in reducing the uncertainty in machine perception Sensor fusion relies on data from several of the same types of sensors known as competitive configuration Combining different types of sensors (fusing proximity sensor with speedometer data) usually yields a more comprehensive understanding of object and this kind of setup is known as complementary configuration Motion model deals with the dynamics of the object by predicting the current state of the car by drawing from a range of values that depend on its state during the last time step. Measurement model works on the dynamics of the car's sensors.

The susceptibility to interference is one feature shared by all sensors. Sunlight can hide or blind a camera, and a radar can be jammed. These situations may provide sensory data that is distorted, spotty, or just incorrect. As a result, the majority of sensor data collected in the actual world consists of two components: a signal (the part we're interested in) and noise (the part we'd want to avoid). Sensor fusion analyzes multiple data sources at once in an effort to extract the noise from the facts and provide with a more filtered output

On the basis of abstraction level, Sensor fusion can be divided into –

**Low-Level**

Low-level sensor fusion takes raw data as input. We're referring here to the sensor's point data measurements. This approach makes sure we don't add any noise to the data upon post-processing it. The downside to this method is that it requires the processing of an immense amount of data.

**Mid-Level**

At the intermediate level, data fusion operates on object hypotheses. It uses data that has been interpreted either within the sensor itself or by a different processing unit. For example, when a camera thinks an object is straight ahead, the Lidar might sense it slightly to the right. With mid-level sensor fusion, these two interpretations are weighted to arrive at a single projection.

**High-Level**

Tracks are hypotheses about an object's movement in space. In high-level sensor fusion, we again see the merger of two hypotheses in a weighted manner. This time, however, the hypotheses aren't just about an object's position, but also about its trajectory, thus incorporating its past and future states

Given that different kinds of sensors each have advantages and disadvantages, a powerful algorithm will also favor some data points over others. For instance, you should assign more weight to the speed sensor because it is likely to be more accurate than the parking sensors. Perhaps the speed sensor is not particularly accurate, in which case you should place more emphasis on the proximity sensors. The possibilities are almost limitless and vary according to the use case.

Some useful sensor fusion algorithms widely used for Noise filtering include –

# Kalman Filter

A Kalman filter is an algorithm that takes data inputs from multiple sources and estimates unknown values, despite a potentially high level of signal noise. Often used in navigation and control technology, Kalman filters have the advantage of being able to predict unknown values more accurately than individual predictions using single methods of measurement.

The Kalman Filter is further divided into two categories on the basis of their application –

***Extended Kalman Filter***:

ROS Package: robot_localization, robot_pose_ekf

The Extended Kalman filter is a state estimation and localization algorithm that makes an estimate the position of the robot with an estimated noise, then uses the information from the sensor as a measurement and update it further to get a more realistic value and continue the same steps in recursion in order to get a more accurate position of the robot.

The main difference between Kalman Filter and Extended Kalman Filter is that Kalman Filter works only when the state space model is linear and the function that governs their transition from one state to another.

On a high level, EKF algorithm has two stages, a predict phase which predicts the state estimate of the robot using the state space model and control input applied at previous time step and update phase that calculates the difference between actual sensor data and the values predicted by the measurement model and accordingly updates the covariance matrix and the Kalman gain at time t.

EKF help to predict the nonlinear quantity of the robot for instance, the pose of the robot which includes the orientation. With reference to the ROS packages, robot_localization package provides 3D tacking based on Extended Kalman filter with mulitple sensors (odom, IMU, GPS, etc) and more functionality while robot_pose_ekf can take only two odometry data and one imu data

***Unscented Kalman Filter***:

ROS Package: robot_localization, OpenCV/Eigen-based packages

The Unscented Kalman Filter belongs to a bigger class of filters called Sigma-Point Kalman Filters or Linear Regression Kalman Filters. Similar to Extended Kalman Filter, Unscented Kalman Filter is used to linearize a nonlinear function of a random variable through a linear regression between n points drawn from the prior distribution of the random variable. The UKF has a concept of sigma points. For instance, some points on source Gaussian are taken and these are then mapped on target Gaussian after passing points through some non linear function and in order to calculate the new mean and variance of transformed Gaussian. In certain cases, It can be very difficult to transform whole state distribution through a non linear function but it is very easy to transform some individual points of the state distribution, these individual points are sigma points.

The basic difference between EKF and UKF is that in Extended Kalman Filter, only one point is selected i.e mean and approximate, although in UKF, a bunch of points called sigma points are taken and approximate with a fact that more the number of points will give a more precise approximation.

In more depth, The number of sigma points in UKF depends on the dimensionality of the system and the general formula to follow is 2N+1 where N denotes the dimensions.

***Particle Filter***:

ROS Package: AMCL (Adaptive Monte Carlo), bfl

In case of particle filter, This problem assumes that the robot has a map of its environment, however, the robot either does not know or is unsure of its position and orientation within that environment. The particle filter localization is similar to how people used to find their way around unfamiliar places using physical maps. The particle filter localization makes numerous assumptions (particles) about the possible locations of the robot throughout the world. Then, it compares what it is seeing (as determined by its sensors) with each particle's prediction of what it would see. It's more likely that assumptions that reflect what the robot sees are accurate estimates of its location. Which hypotheses are the ones that are most likely to match the actual robot's location should become more and more obvious as the robot travels around in its environment. the particle filter localization first initializes a set of particles in random locations and orientations within the map and then iterates over the following steps until the particles have converged to (hopefully) the position of the robot:

- Capture the movement of the robot from the robot's odometry
- Update the position and orientation of each of the particles based on the robot's movement
- Compare the laser scan of the robot with the hypothetical laser scan of each particle, assigning each particle a weight that corresponds to how similar the particle's hypothetical laser scan is to the robot's laser scan
- Resample with replacement a new set of particles probabilistically according to the particle weights
- Update your estimate of the robot's location

The primary focus of particle filters are to solve non-gaussian oise problems although they are generally more computationally expensive then Kalman Filter, the reason is because particle filter uses simulation methods instead of analytical equations in order to solve localization and estimation tasks. The similiarity between Kalman Filter and Particle filter is that both these make use of an iterative process in order to produce its estimations. The **AMCL (Adaptive Monte Carlo Localization)** and **bfl (bayes filtering library)** package provides support for particle filters**.**

***Bayesian Models***:

ROS Package:, bfl

Bayes' rule, which deals with probability, is the basis of the update equation described earlier that combines the motion and measurement models. Bayesian networks, also based on Bayes' rule, predict the likelihood that any one of several hypotheses is the contributing factor in a given event.

Some well-known Bayesian algorithms include –

- K2
- Hill climbing
- Iterative hill climbing
- Simulated annealing

The Bayesian Filtering Library (BFL) provides an application independent framework for inference in Dynamic Bayesian Networks, i.e., recursive information processing and estimation algorithms based on Bayes' rule. Above mentioned bayesian algorithms are heuristic search algorithms used for mathematical optimization problems in the field of Artificial Intelligence and purposes to select the best route out of all possible routes.

# Why Extended Kalman Filter?

Our use-case mainly involves a turtlebot3 performing SLAM by navigating the environment autonomously. For this use-case, we will be resorting to data output from wheel odometry sensor and inertial measurement unit sensor. It is predictable that since the data rom the aforementioned sensors will contain non-linear output (for e.g orientation) thereby indicating that using Linear Kalman Filter may not be a good choice since it is mainly used for linear applications.

One of the widely used algorithms is the Extended Kalman Filter. The EKF is heuristic for nonlinear filtering problems and provides stable functionality based on a considerable tuning.
The Extended Kalman Filter is a mathematical tool if –

- Have a state space model of how the system behaves,
- Have a stream of sensor observations about the system,
- Can represent uncertainty in the system (inaccuracies and noise in the state space model and in the sensor data)
- You can merge actual sensor observations with predictions to create a good estimate of the state of a robotic system

The main benefit of Extended Kalman Filter is that the gain and covariance equations converge to constant values on a much bigger set of trajectories as compared to equilibrium points resulting in a better convergence of estimation.

Further down the line, the Extended Kalman Filter can be modified into more variants consisting of –

**Iterated extended Kalman Filter**

The iterated extended Kalman filter improves the linearization of the extended Kalman filter by recursively modifying the centre point of the Taylor expansion. This reduces the linearization error at the cost of increased computational requirements

**Robust extended Kalman Filter**

The extended Kalman filter arises by linearizing the signal model about the current state estimate and using the linear Kalman filter to predict the next estimate. This attempts to produce a locally optimal filter, however, it is not necessarily stable because the solutions of the underlying Riccati equation are not guaranteed to be positive definite. One way of improving performance is the faux algebraic Riccati technique which trades off optimality for stability. The familiar structure of the extended Kalman filter is retained but stability is achieved by selecting a positive definite solution to a faux algebraic Riccati equation for the gain design**.**

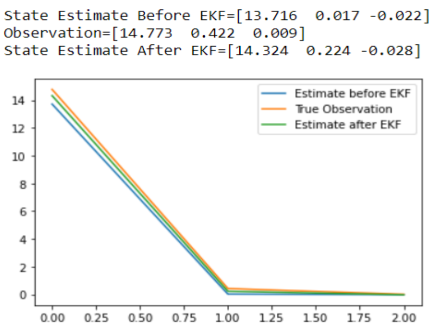
**Invariant extended Kalman Filter**

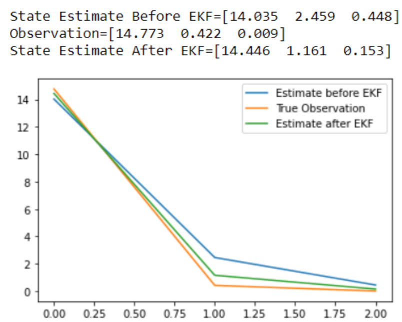
The invariant extended Kalman filter (IEKF) is a modified version of the EKF for nonlinear systems possessing symmetries (or *invariances*). It combines the advantages of both the EKF and the recently introduced symmetry-preserving filters. Instead of using a linear correction term based on a linear output error, the IEKF uses a geometrically adapted correction term based on an invariant output error

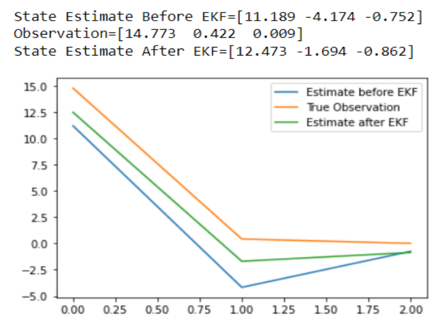
## Implementing Extended Kalman Filter (Multi-D)

The following steps show the implementation of Extended Kalman Filter using Python3 and MATLAB. For python3, the implementation was carried out using numpy package to perform

### Python3 Implementation

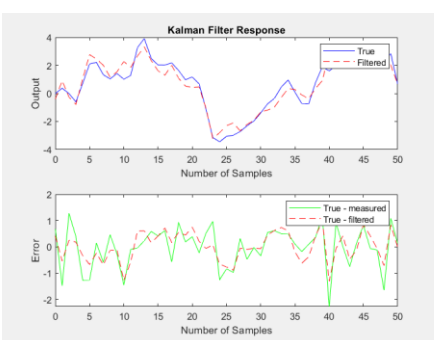


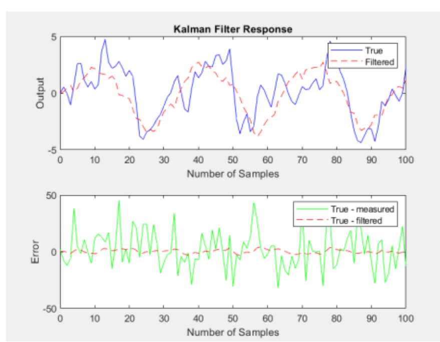



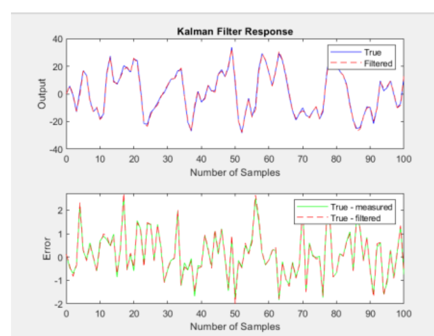




Extended Kalman Filter was implemented using python3 by initially deriving the matrix equation for previous timestep and current timestep and then converting those matrix equations to state space models by incorporating A and B matrix which represent the transition from one state to another. The above results indicate that EKF provides a more smooth output of the true observations. The graphs include the plots when some noise was added to the EKF node, and it visible that there was some deviation but the output was close to the actual value

### MATLAB (R2022a) Implementation




In case of MATLAB, Extended Kalman Filter was implemented as a filter that uses the nonlinear state update model and measurement equations and performs state transition using Jacobians matrix. The measurement jacobian replaces the measurement matrix. The Automated driving toolbox of MATLAB provides support and package to implement Extended Kalman Filter and includes some predefined EKF functions for –

- *Constant Velocity*: starting with one-dimensional to three-dimensional state representation
- *Constant Acceleration*: three-dimensional state representation and acceleration values in x,y,z direction
- *Constant turn rate*: three-dimensional state representaion with an additional **omega** term (as turn-rate)

# ROS Packages for Extended Kalman Filter

Kalman Filter implementation can be performed using python3 or any other programming language and the values can be tweaked in accordance with how we want the output to be shown. Although, creating a ROS node of Kalman Filter code can be a huge task considering all the state space equation needs to be formulated before.

For this, there are several pre-defined ROS packages that assist us in implementing EKF and UKF on an actual robot for simulation and testing purposes. Few of the widely used packages include –

- **robot_localization**

  robot_localization package of ROS is a collection of state estimation nodes, each of which is an implementation of a nonlinear state estimator for robots moving in 3D space. It contains two state estimation nodes, ekf_localization_node and ukf_localization_node. In addition, robot_localization provides navsat_transform_node, which aids in the integration of GPS data. The package provides nonlinear state estimation through sensor fusion of an abritrary number of sensors.

  Take a look at a few features of the robot_localization package that makes it superior then the other filter packages –

  - Fusion of an arbitrary number of sensors. The nodes do not restrict the number of input sources. If, for example, your robot has multiple IMUs or multiple sources of odometry information, the state estimation nodes within `robot_localization` can support all of them.
  - Support for multiple ROS message types. All state estimation nodes in `robot_localization` can take in nav_msgs/Odometry, sensor_msgs/Imu, geometry_msgs/PoseWithCovarianceStamped, or geometry_msgs/TwistWithCovarianceStamped messages.
  - Per-sensor input customization. If a given sensor message contains data that you don't want to include in your state estimate, the state estimation nodes in `robot_localization` allow you to exclude that data on a per-sensor basis.
  - Continuous estimation. Each state estimation node in `robot_localization` begins estimating the vehicle's state as soon as it receives a single measurement. If there is a holiday in the sensor data (i.e., a long period in which no data is received), the filter will continue to estimate the robot's state via an internal motion model.

  As mentioned above, robot_localization package provides support for both ekf and ukf, therefore there are a vast number of parameters common to both the algorithms, as most of the parameters control how data is treated before being fused with the core filters. Some common parameters include –

  - *frequency* - real-valued frequency, in Hz, at which the filter produces a state estimate.
  - *sensor_timeout* - real-valued period, after which sensor is considered to have timed out
  - *two_d_mode* – keeping it to false, will fuse 0 values for all 3D variables
  - *transform_timeout* - specifies how long to wait if a transformation is not available yet.
  - *transform_time_offset* - packages require that your transforms are future-dated by a time offset
  - *sensor* - For each sensor, users need to define this parameter based on the message type

- **robot_pose_ekf**

  robot_localization package of ROS is a collection of state estimation nodes, each of which is an implementation of a nonlinear state estimator for robots moving in 3D space. It contains two state estimation nodes, ekf_localization_node and ukf_localization_node. In addition, robot_localization provides navsat_transform_node, which aids in the integration of GPS data. The package provides nonlinear state estimation through sensor fusion of an abritrary number of sensors.

  Take a look at a few features of the robot_localization package that makes it superior then the other filter packages –

  - Fusion of an arbitrary number of sensors. The nodes do not restrict the number of input sources. If, for example, your robot has multiple IMUs or multiple sources of odometry information, the state estimation nodes within `robot_localization` can support all of them.
  - Support for multiple ROS message types. All state estimation nodes in `robot_localization` can take in nav_msgs/Odometry, sensor_msgs/Imu, geometry_msgs/PoseWithCovarianceStamped, or geometry_msgs/TwistWithCovarianceStamped messages.
  - Per-sensor input customization. If a given sensor message contains data that you don't want to include in your state estimate, the state estimation nodes in `robot_localization` allow you to exclude that data on a per-sensor basis.
  - Continuous estimation. Each state estimation node in `robot_localization` begins estimating the vehicle's state as soon as it receives a single measurement. If there is a holiday in the sensor data (i.e., a long period in which no data is received), the filter will continue to estimate the robot's state via an internal motion model.

  As mentioned above, robot_localization package provides support for both ekf and ukf, therefore there are a vast number of parameters common to both the algorithms, as most of the parameters control how data is treated before being fused with the core filters. Some common parameters include –

  - *frequency* - real-valued frequency, in Hz, at which the filter produces a state estimate.
  - *sensor_timeout* - real-valued period, after which sensor is considered to have timed out
  - *two_d_mode* – keeping it to false, will fuse 0 values for all 3D variables
  - *transform_timeout* - specifies how long to wait if a transformation is not available yet.
  - *transform_time_offset* - packages require that your transforms are future-dated by a time offset
  - *sensor* - For each sensor, users need to define this parameter based on the message type

# Simulating the Turtlebot3

Trutlebot3 is a low-cost two-wheeled diffrential drive platform designed to run on ROS and Ubuntu. A diffrential drive robot refers to the state where each of the robot wheels perform independently with respect to each other.

Before moving forward with turtlebot3 package installation, it is important to install the relevant ROS development packages to ensure that the turtlebot3 runs smoothly.

## Launching the turtlebot3

The package command is –

*sudo apt-get install ros- foxy -joy ros-foxy-teleop-twist-joy *

*ros- foxy -teleop-twist-keyboard ros-foxy-laser-proc *

*ros-foxy-rgbd-launch ros-foxy-depthimage-to-laserscan *

*ros-foxy-rosserial-arduino ros-foxy-rosserial-python *

*ros-foxy-rosserial-server ros-foxy-rosserial-client *

*ros-foxy-rosserial-msgs ros-foxy-amcl ros-foxy-map-server *

*ros-foxy-move-base ros-foxy-urdf ros-foxy-xacro *

*ros-foxy-compressed-image-transport ros-foxy-rqt* *

*ros-foxy-gmapping ros-foxy-navigation ros-foxy-interactive-markers*

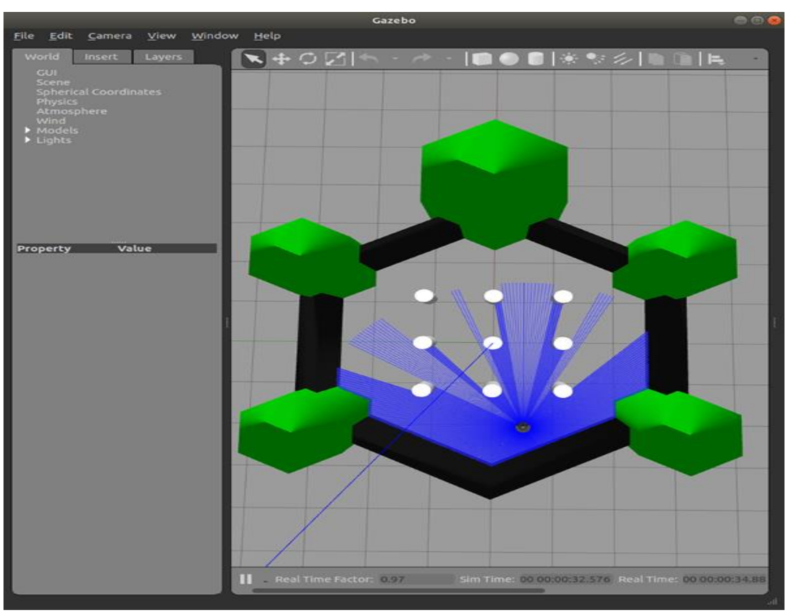
Then proceed ahead with the turtlebot3 package installation –

*sudo apt-get install ros-foxy-dynamixel-sdk*

*sudo apt-get install ros-foxy-turtlebot3-msgs*

*sudo apt-get install ros-foxy-turtlebot3*

After launching the sample turtlebot3 world, a following gazebo environment should appear –

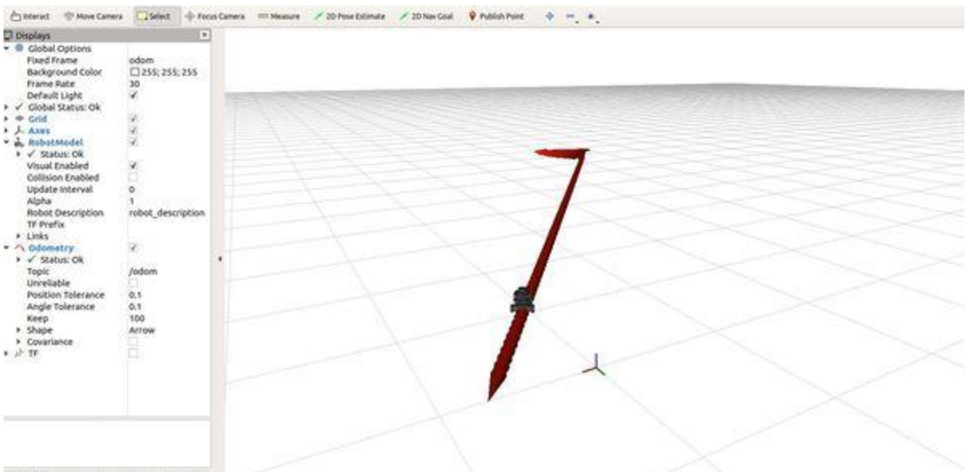


## Moving the turtlebot3

The command to move the turtlebot3 is –

*roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch*

The following command, allows you to move and increase the velocity of the turtlebot3 using w, a, s, d and x keys in such a way –




**Note**: Remember to declare the turtlebot3 type (i.e burger, waffle, etc) before running the teleop node

## Checking for topics

While the turtlebot3 is moving around the environment

**Incorporating EKF**

The package command is –

# Introducing Noise

Since, the Odometry and IMU sensor values for Gazebo are almost perfect and when simulating the turtlebot3 in actual environment, it may not be the case as multiple hardware issues or noise elements like weather noise, sensor noise or even surface based noises.
In order to make the Gazebo simulation more realistic, noise factor is introduced in the odometry motion model equation and IMU sdf tag of turtlebot3

## Odometry noise

The Odometry data provided by Gazebo is highly accurate thereby in certain situations, making the simulation less realistic. Therefore, a ROS node is developed to add some random noise to Gazebo's odometry data. Odometry measures the relative motion of the robot between time t and t-1. In a 2D environment, the robot co-ordinates are represented by points (x,y) and an orientation angle $\theta$.

Robot pose at timestep t-1 and t is indicated by –

$$p_{t-1} = (x_{t-1}, y_{t-1}, \theta_{t-1})$$
$$p_t = (x_t, y_t, \theta_t)$$

Now, consider if the robot makes a rotational motion and then a translational motion –

$$u_t = (\delta_{rot1}, \delta_{trans})$$

In an Ideal case, the odometry values of rotation and translation can be calculated as –

$$\delta rot1 = atan2(yt-yt-1, xt-xt-1)-\theta t-1$$
$$\delta trans=(xt-xt-1)2+(yt-yt-1)2$$

In actual situations, there is always some sort of noise and these noises can be functioned as random normal distribution noise with mean and standard deviation. Further, we can add the random normal distribution term to the equation –

**Rotational noise element**

$\hat{\delta}_{rot1} = \delta_{rot1} + N(0, \sigma_{rot^2}2)$

**Translational noise element**

$\hat{\delta}_{trans} = \delta_{trans} + N(0, \sigma_{trans}2)$

**IMU (Inertial Measurement unit) noise**

Gazebo has a built in noise model that can apply Gaussian noise to a variety of sensors. While Gaussian noise may not be very realistic, it is better than nothing and serves as a good first-pass approximation of noise. Gaussian noise is also relatively easy to apply to data streams.

In case of IMU sensors, two kinds of disturbance, noise and bias, are incorporated, to angular rates and linear accelerations. Angular rates and linear accelerations are considered separately, leading to 4 sets of parameters for this model: rate noise, rate bias, accel noise, and accel bias. No noise is applied to the IMU's orientation data, which is extracted as a perfect value in the world frame.

Noise is sampled from a Gaussian distribution and is additive. The Gaussian distributions from which noise values will be sampled can have their means and standard deviations set.
Each component (X, Y, and Z) of each sample is independently sampled to obtain a noise value, which is then added to that component.

Although bias is sampled just once, at the beginning of the simulation, it is also additive.
The Gaussian distributions from which bias values will be sampled can have their means and standard deviations set. The assumption is that the specified mean reflects the bias's magnitude and that it is equal. Bias will be sampled in accordance with the provided parameters, then with equal probability eliminated.

In order to create a noisy IMU data,

## Tuning EKF parameters

Since, the Odometry and IMU sensor values for Gazebo are almost perfect and when simulating the turtlebot3 in actual environment, it may not be the case as multiple hardware issues or noise elements like weather noise, sensor noise or even surface based noises.

## Surface Friction of Gazebo

Since, the Odometry and IMU sensor values for Gazebo are almost perfect and when simulating the turtlebot3 in actual environment, it may not be the case as multiple hardware issues or noise elements like weather noise, sensor noise or even surface based noises

## Conclusion

Since, the Odometry and IMU sensor values for Gazebo are almost perfect and when simulating the turtlebot3 in actual environment, it may not be the case as multiple hardware issues or noise elements like weather noise, sensor noise or even surface based noises

## Future Scope

Since, the Odometry and IMU sensor values for Gazebo are almost perfect and when simulating the turtlebot3 in actual environment, it may not be the case as multiple hardware issues or noise elements like weather noise, sensor noise or even surface based noises

## References

Since, the Odometry and IMU sensor values for Gazebo are almost perfect and when simulating the turtlebot3 in actual environment, it may not be the case as multiple hardware issues or noise elements like weather noise, sensor noise or even surface based noises