**SLAM MODULE**

* Aryan Kayande

**Checkpoint 1 : -**

**Localisation** = estimating robot’s position

**Mapping** = building a map

**SLAM** = simultaneously building the map and estimating the robot's position in it. Basically SLAM involves estimating the robot's poses and the landmark’s position at the same time! This needs to be in real time, based on which the car takes its decisions of where to go next. The problem of SLAM is that a map is needed for localisation and pose is needed for mapping…

**Formal definition of SLAM** : -

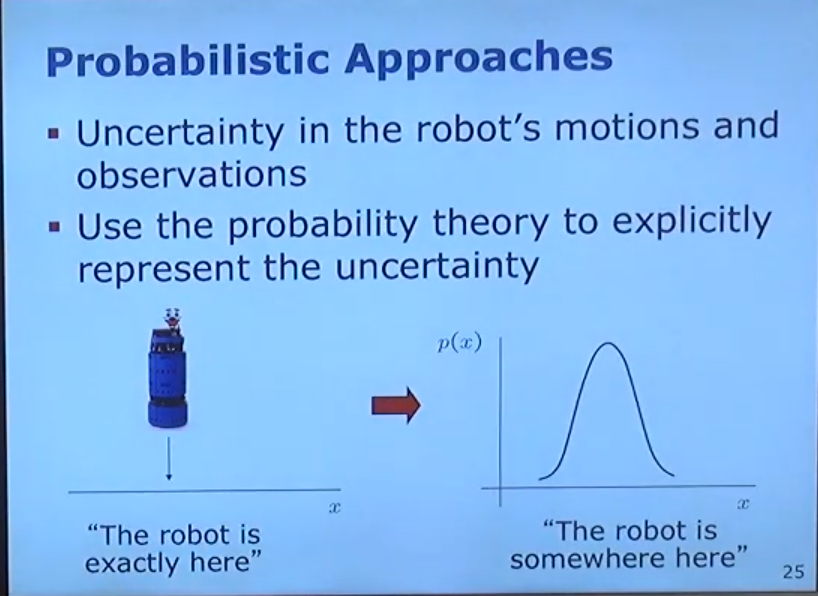
Given : -

1. Robot’s controls : {u1, u2, … , uT}
2. Observations : {z1, z2, … , zT}

Wanted : -

1. Map of environment : m
2. Path of the robot : {x0, x1, … , xT}

We do this as a **probabilistic** **approach** :



Thus, SLAM returns us a probability as follows : -

P (x0:T , m | z1:T , u1:T )

which is essentially, probability distribution of x and m **given** all the observations and controls.

**Full SLAM v/s Online SLAM** : -

Full SLAM estimates the entire path i.e. gives the information of the whole trajectory. On the contrary, online SLAM only gives the recent pose and makes decisions based on that. Thus, we can ignore the previous poses in online SLAM.essentially, the probability distribution returned in online SLAM is the integral over all the previous poses.

**ROS : -**

**Nodes :** essentially nodes are individual computational units that are responsible for a single, modular purpose. These nodes communicate with each other by publishing and subscribing to messages on ROS topics. Tried out various tasks related to nodes like ros2 node list, rqt graph and ros2 node info.

**Topics :** Topics are the communication channels that enable nodes to exchange messages. Nodes can publish messages or subscribe to topics to receive messages. Tried out many things like ros2 topic functions like list, echo, info etc. Topics provide asynchronous communication.

**Services :** Services are another method of communication for nodes in ROS. They are based on a call-and-response model. Basically, unlike topics, services provide data when they are specifically called by a client. Tried out service functions like interface show and service clear, call etc.

**Parameters :** Parameter is basically a mechanism for storing configuration data within a system. Each node in ROS stores parameters as integers/floats/booleans or strings. Each node maintains its own parameters. You can get the value of any parameter using ros2 param get or also set any parameters’s value using ros2 param set function. You can also store all the current parameter values in a file using ros2 dump function for future use using ros2 load. Tried out all these functions and changed the background colour of turtlesim for fun:)

**Actions :** Actions are also one of the communication types in ROS. However, these are intended for long running tasks. They provide steady feedback and also allow cancellation, unlike services. Actions use a client-server model, in which a node sends a goal to an ‘action server’ which then returns the feedback and the result. For eg. turtlesim turtle\_teleop\_key. When we press keys on our keyboard, the turtle moves accordingly. The list of actions can be seen using ros2 node info itself or we can also use ros2 action list. We can set the goal of the turtle using action send\_goal. Did some “experiments” on rotating the turtle and moving it all around the place :)

**Rqt-console :** Things like ‘Oh no I hit the wall’ are seen on the console log. We can also see the different logger levels like Fatal, Error, Warn, Info and Debug.

**Launching Nodes :** Instead of opening new terminals every time for new turtle sims, we can directly use ros2 launch to do this :) This runs a python file named multisim.launch.py which opens up 2 turtlesim windows.

**Filters : -**

**Bayes Filter :** Basically, it is a technique used to estimate the pose of a car in a recursive fashion. It estimates the state of the system based on the most recent command and observation. It determines the conditional probability of the system reaching the state x(t+1) from x(t) given the control command u. Various models used in Bayes filters are Kalman filter, Extended Kalman filter, Unscented Kalman filter etc.

**Kalman Filter :** Kalman filter is the Bayes filter for the Gaussian case. It performs recursive state estimation. The Kalman filter takes the weighted average of your predicted position and the position obtained from observations. Note : this is the optimal solution for linear models (meaning that the model can be represented as a linear function of the parameters) and gaussian distributions (probability distribution of x1, x2 is gaussian in nature) :) A Gaussian transformed through a linear model stays Gaussian!

Linear Model : -

x(t) = At \* x(t-1) + Bt \* u(t) + e(t)

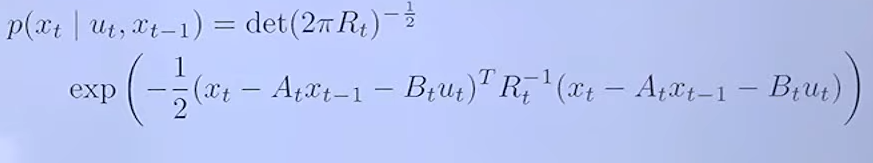
z(t) = Ct \* x(t) + d(t)

Here, x(t) represents the state of the system at the time t. u(t) is the control command at the time t. z(t) represents the predicted observation, based on the current state.

Components of Kalman filter :

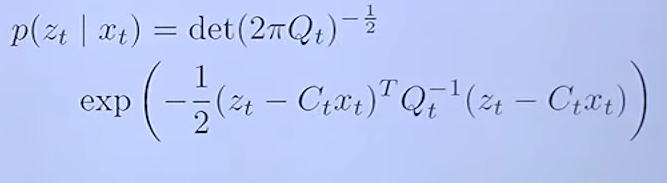
1. At : n x n matrix. It describes how the state transforms from (t-1) to t.
2. Bt: n x l matrix. Describes the effect of the control u(t) on the state from (t-1) to t.
3. Ct : k x n matrix. Gives a mapping between state x(t) and observation z(t).
4. d(t), e(t) are random variables representing the noise in the measurements.

Thus, a linear model returns the probability of the car achieving a state x(t+1) given the state x(t) and command u(t) as follows :



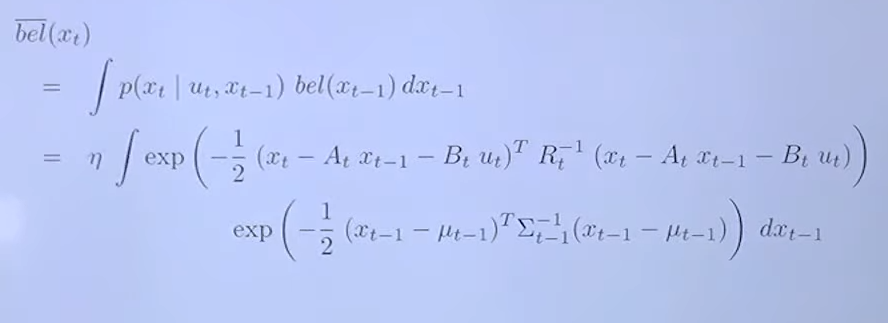
Here the matrices are nothing but the negative mean of the expected state based on the previous position and the control command. Also, R is the covariance matrix, which takes into account the uncertainty of the motion model.

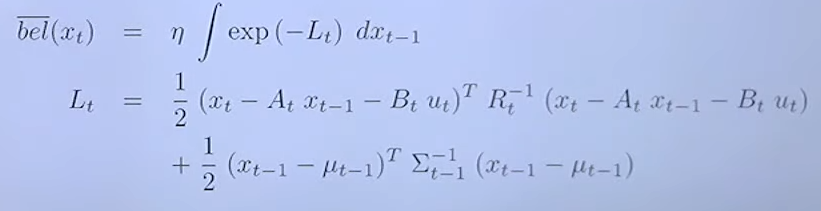
Similarly, the linear observation model gives the predicted state of the system z(t) based on the observed state x(t) by multiplying by the Ct. as above, we get the probability distribution p(z(t) given x(t)) as follows :

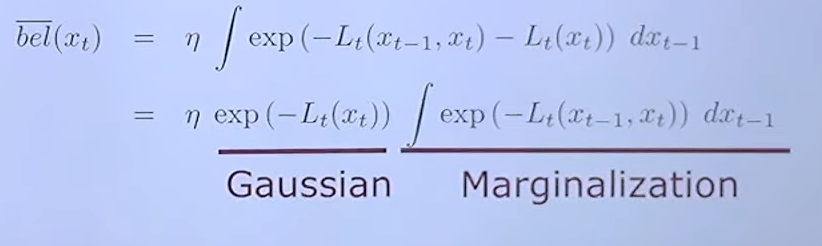


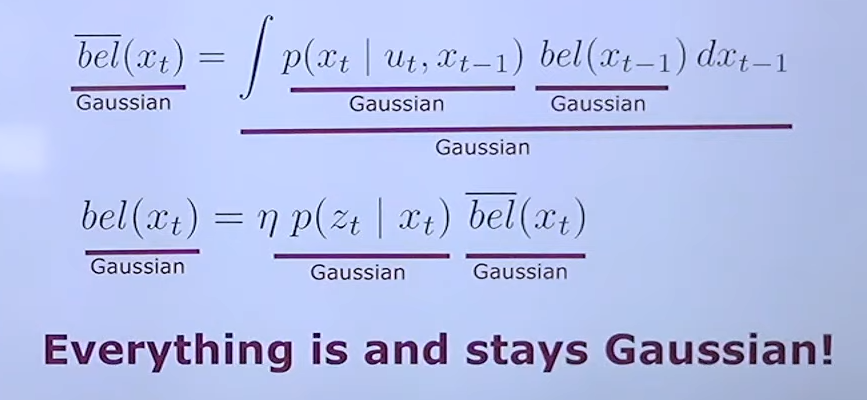
As above, we have the negative mean and also, the Qt matrix accounts for the noise in the measurement. Essentially, the z(t) - Ct \* x(t) term is the error i.e. the difference between what we observed and what we predicted.

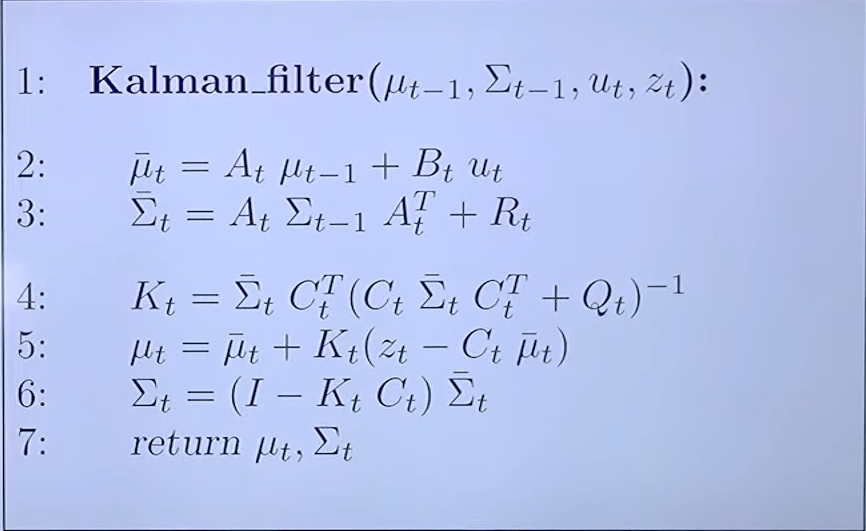
While predicting the belief(x(t)), everything remains gaussian and thus, we can write bel(x(t)) in terms of bel(x(t-1)) as follows :









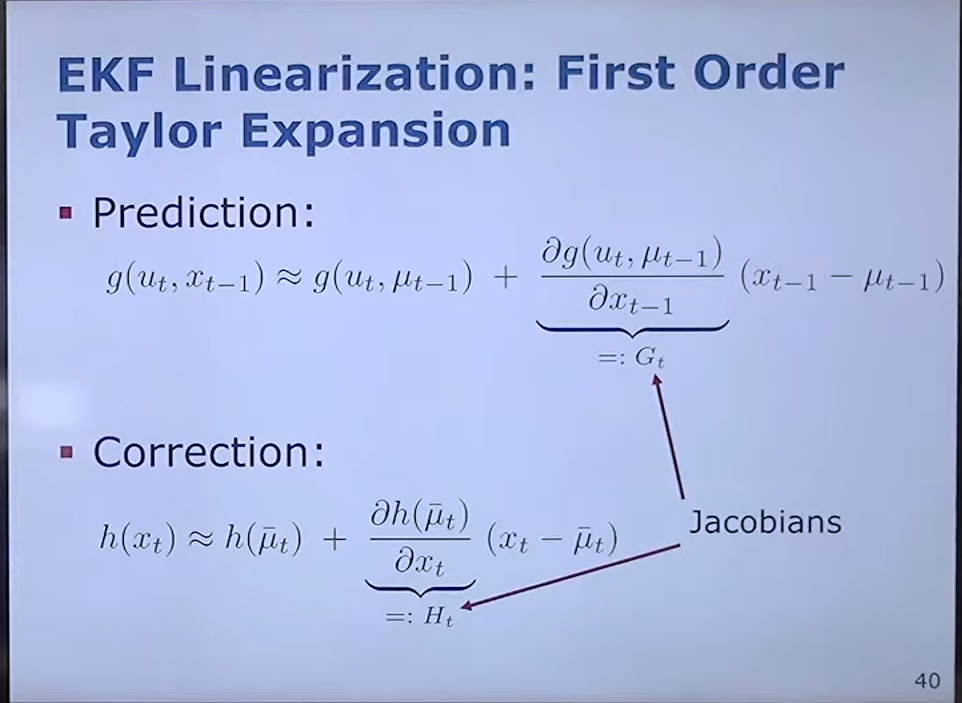
**Kalman Filter Algorithm : -**

In steps 2 and 3, we compute the mean and the uncertainty at time t based on its value at time t. Step number 4 represents the calculation of a weight called Kalman gain. It is basically a weighting factor that measures whether we trust our observations more or our predictions more. Steps 5 and 6 introduce the correction terms, and give the final value of mean and uncertainty, taking into account the Kalman gain as well.

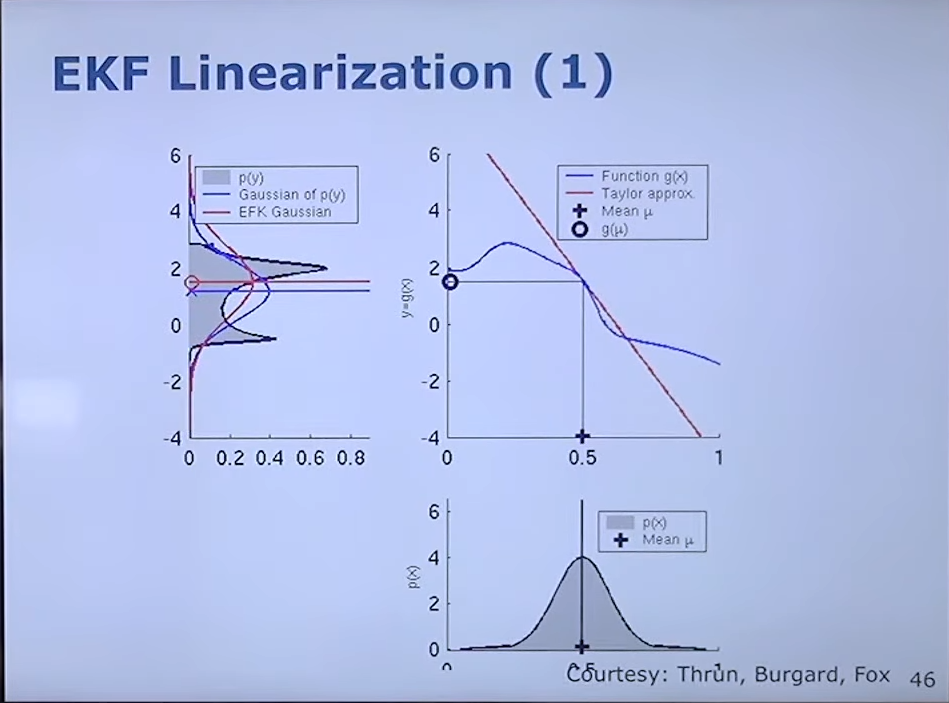
What if the noise is not Gaussian or the motion is not linear?!

Non linear functions transform a gaussian graph to a non gaussian one. How to resolve this?

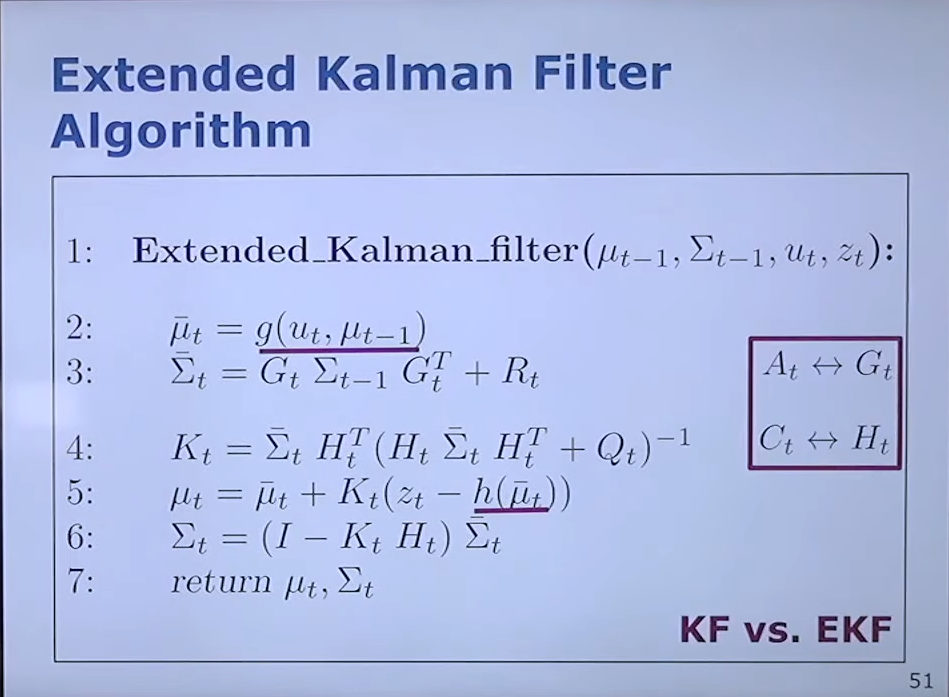
We first linearise the system dynamics and then apply the Kalman filter iteratively. For linearising a nonlinear function, we use first order Taylor expression, wherein we add the product of the Jacobian of the function with respect to the (interested) variable and the difference indicating how far are we from the linearisation. Similar thing is done for the correction term as well. This is done as follows:



For a n dimensional vector valued function g, the Jacobian is defined as a matrix consisting of partial derivatives of all the individual functions with respect to all variables.



In the above figure, the blue curve in the middle is the nonlinear function, which is then linearized to the red line. If we observe, the transformation about the blue curve leads to a non-Gaussian result, whereas if we transform about the now linearized red line, we again get a Guassian curve. The EKF algorithm works as follows :



The only difference between KF and EKF is in steps 2 and 5, where the linear model is replaced by the non linear one. In step 1, we use the non linear model to predict the mean. Note : - in the equations, G and H are not the linear matrices A and C, rather they are the Jacobians of the functions g and h respectively.

Consider a case, wherein we have a perfect sensor i.e. no noise. Now, if we consider step 4, where we calculate the Kalman gain, the inverse teams all cancel out and eventually we are left with the Ht inverse i.e. the inverse of the Jacobian of the h function. Thus, when we plug in this value in the next steps, we get the mean as H inverse multiplied by zt. Which is essentially mapping from the state of observations into the state space. This will give where the car is, given the sensor reading, which makes sense, as we should get exact results, provided there is no noise in the sensors :)

Another case is if the sensor has a huge amount of noise, the worst possible sensor :( here, the Q matrix will be very large, infinite for all practical purposes. We take the inverse of Q, and thus our Kalman gain would be 0. So, the predicted belief would remain as the state estimation. So, in the extreme cases, either you will completely trust your observation, or completely ignore it and trust your predictions,