Simultaneous Localization and Mapping

Shabnam Sahay

June 2020

These notes are a summary of my learning from the SLAM course videos offered by Cyrill Stachniss. Through these, I aim to provide an overview of robot mapping, SLAM, and ultimately its practical implementation via the Extended Kalman filter algorithm.

1 Introduction to Robot Mapping

1.1 Important Terms

Robot: a device which moves through the environment. It has some mechanical means that allows it to move. It is also equipped with sensors. The controls and sensor inputs are exploited to gain knowledge about the environment.

Mapping: modeling the environment. A map is some form of representation of the environment, able to be used for decision-making.

State estimation: knowing the position of the robot of the environment, the positions of some nearby landmarks, and using this to estimate the current state of the robot in the environment e.g. its position, the position of landmarks, etc. One way to do estimation is via a recursive base filter.

Localization: estimation of the location of the robot in relation to the world - the x,y,z coordinates, as well as the angular orientation. The location is sometimes also called the pose.

Mapping: having sensor data, and using it to estimate a model of the environment, i.e. to map it. To be able to map, we must know the position of the sensor itself.

SLAM: when the position of the sensor itself is not certain, SLAM is required. It simultaneously estimates the position of the robot itself, as well as the positions of nearby landmarks in the environment.

Navigation: the ability of the robot to make decisions related to which path to choose to go from one location to another. Usually limited to xyz movement.

Motion planning: connected to navigation. It entails seeking the optimal path to move from one location to another. It may not always involve rigid xyz movement, and allows one to plan a trajectory or a system of states that need to be moved through in order to change locations.

1.2 What is SLAM?

SLAM involves computing different positions of the robot at different points in time, as well as computing and updating a map or model of the environment simultaneously.

- Localization: estimating the robot's location or trajectory.
- Mapping: building a map
- SLAM: mapping and localizing simultaneously

In practical applications, one almost always needs to use SLAM - localization or mapping on their own are not accurate enough. Even if the ultimate goal is just something relatively simple, such as modelling the environment, SLAM is needed to do the job effectively.

Localization example

A landmark is something the robot can observe and recognise. As seen, localization involves knowing all the landmarks' positions at a point in time, and using them to find out where the robot itself is.

Let us look at a brief example of how localization works. Suppose due to some mismatches in the wheel dimensions, the robot thinks it is turning left, but it actually ends up turning right. After some movement in the new direction, it checks its distance from a previously known landmark, and realises that there is a difference between what it thinks the distance should be, and what it actually is. It uses this realisation to make a correction in its pose, and attempts to bring the estimated and actual poses closer to each other. The amount of correction depends on the accuracy of its sensors and the accuracy of its motion execution. How much weight we give to each of these depends on which one gives more precise data.

Mapping example

In this case, the motion execution is perfect, so the exact location of the robot itself is known. The sensors are used to continually detect the positions of the landmarks. Sensor data is often noisy, and in the ideal case the degree of accuracy of the sensors is known - for example, the actual landmark has a 95% chance of being within 10 cm of the detected position. This data can then be used to drawn an uncertainty ellipse around the detected position, and thus model the environement.

SLAM example

Looking back to the localization example, in a practical situation, when the robot's actual direction becomes different from the direction it thinks it is moving in, it is usually not able to realise this accurately. This is because the position of the landmark is generally not known to it from before, and hence it is not able to use the difference in the estimated and actual position to correct its trajectory. If we consider ideal trajectory accuracies:

Trajectory accuracy: Localization > Mapping > SLAM

However, for good localization, we need a perfect map or model of the environment. Conversely, for good mapping, we need extremely precise data on the location/pose estimate of the robot. Thus the use of either one becomes a type of chicken-and-egg problem.

This shows the high inter-dependency of localization and mapping on each other: they cannot be decoupled. This is why SLAM, i.e. simultaneous localization and mapping, is used for autonomous navigation.

SLAM applications

- At home: hoovers, lawn mowers
- Air: surveillance with UAVs; crops, traffic, etc.
- Underwater: reef monitoring
- Underground: exploration of mines, catacombs, etc.
- Space: terrain mapping for localization

Note: If the robot moves randomly while mapping the environment, there is a reasonable chance that it will miss some areas. This is because the mapping is a passive process - the SLAM system doesn't tell the robot *where* to go to do its mapping.

1.3 Defining the SLAM problem

Given

- The robot's controls

  $u_{1:T} = u_1, u_2, u_3, ..., u_T$

  These are control commands. For example, u_T may be a command to move one metre forward. They are closely related to the odometry of the robot, with some important distinctions.

  A control command is something that one physically sends to the robot, whereas odometry is based on the feedback that the robot sends back

based on its motion (e.g. an encoder on the wheels counting the number of revolutions, telling us that the robot actually moved only 99 cm).

In theory, use of the control commands is enough to direct motion.

- Observations

  $z_{1:T} = z_1, z_2, z_3, ..., z_T$

  These observations are indexed based on the time elapsed. They could be laser scans indicating the proximity to the closest obstacle, camera images detecting landmarks, etc.

Wanted

- Map of the environment

  m: an accurate model of the environment.

- Path of the robot

  $x_{0:T} = x_0, x_1, x_2, ..., x_T$

  The path variables start from the index 0, because each control command connects two poses. Suppose we have two commands u_1 and u_2: then the robot will change its pose two times, first from x_0 to x_1, and then x_1 to x_2. So x_0 in a sense acts as an origin, creating a frame of reference for the location of the robot.

Both these wanted quantities are highly likely to contain errors when obtained - these errors may be observational, or based on bad data associations, or so on. This is where the use of probabilistic approaches comes in.

1.4 Probabilistic approaches

It is wrong to assume that one knows exactly where the robot is. Even if the motion execution system is extremely accurate, there are always errors that creep in - even if they are extremely small, the uncertainty always remains. As the robot moves for longer times and over greater distances, the errors and uncertainty accumulate.

Due to this uncertainty in the robot's motions and observations, probability theory is used to explicitly represent and model the uncertainty.

In the probabilistic world

Estimation of the robot's path and the environment's map involves use of the following probability distribution:

$p(x_{0:T}, m | z_{1:T}, u_{1:T})$

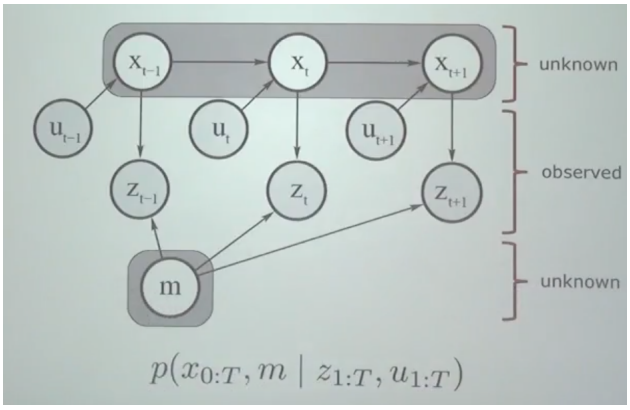
where

- p: probability distribution
- $x_{0:T}$: path/trajectory of the robot through its environment; the pose of the robot at discrete points of time
- m: the environment map
- $|$: given that
- $z_{1:T}$: observations
- $u_{1:T}$: control commands/odometry observations

The ultimate aim is to be able to estimate this probability distribution. The approach we use will change based on the assumptions we make. All estimation techniques make some or the other assumptions. Some may be restrictive, whereas some may be easier to implement or more efficient.

Graphical model

The robot's state is often depicted graphically, as shown.



The arrows represent dependencies; an arrow from a to b indicates that a influences b. For example, we can see that the observations made at a point depend both on the sensor's position, and the landmarks' positions.

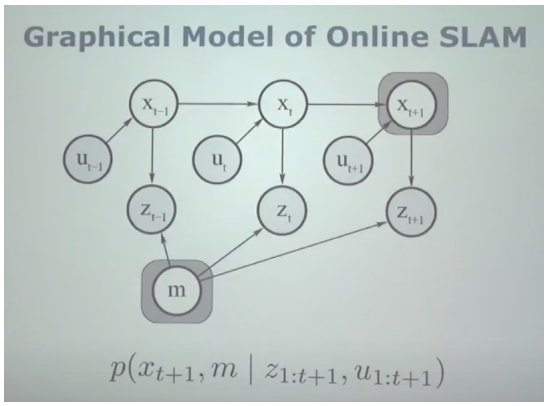
This graph represents full SLAM; estimating both the full trajectory of the robot as well as what the environment is. In contrast, online SLAM only estimates the current pose.

Full SLAM vs. Online SLAM

- Full SLAM estimates the entire path/trajectory and the map.
  $p(x_{0:T}, m|z_{1:T}, u_{1:T})$
- Online SLAM seeks to recover only the most recent pose.
  $p(x_t, m|z_{1:t}, u_{1:t})$

Online SLAM thus estimates only the current pose, and the map built up to the current point, based on all the previously collected sensor data. This is what most robots will use to decide where they are and where they want to go.

Most real-world robot applications will use online SLAM, since it isn't practical to collect all the sensor data throughout the movement and then estimate the poses afterwards - one wants to do it while the robot is moving.



Using online SLAM means marginalizing out the previous poses. The integrals are typically solved recursively, one at a time.

$p(x_t, m|z_{1:t}, u_{1:t}) =$

$$\int_{x_0} \dots \int_{x_{t-1}} p(x_{0:t}, m|z_{1:t}, u_{1:t}) dx_{t-1} \dots dx_0$$

So the map of the environment is the integral of all the possible positions, over x_0, over x_1, and so on. Integrating out the previous estimations together gives a joint probability distribution for the entire movement up to that point.

Note that representing the probability in the form of such an integral simply makes use of the following property:

$$P(A|B) = \int_{allB} P(A|B) dB$$

1.5 Why is SLAM a hard problem?

- Robot path and map are both unknown
- Map and pose estimates are highly correlated
- The mapping between observations and the map itself is unknown
- Wrong data associations can have catastrophic consequences (divergence)

As seen, motion increases the uncertainty - as the robot moves, one needs to combine the uncertainty created by the robot's pose probabilities with the landmark's position probabilities.

As the robot moves further and re-observes certain landmarks, it can use these observations to make corrections in its estimates and thus improve accuracy.

However, if some landmarks are identical, the robot may confuse two or more landmarks, and cause bad data associations. Uncertainty in the pose estimate also changes the set of landmarks that may be thought of as being observed at that moment.

Hence if one makes a wrong association, a wrong estimate of the uncertainty will be obtained.

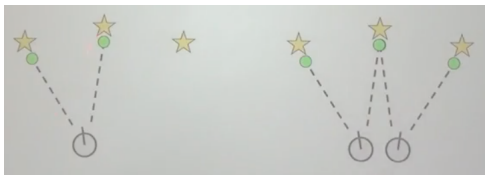
1.6 Taxonomy of the SLAM problem

Kinds of maps estimated

- Volumetric: gives a physical structure of all the surfaces being mapped, or identifies which space is free and which is occupied by obstacles. A good approach if one doesn't know what to expect, and if having obstacle avoidance ability is a requirement.
- Feature based approaches: do not map all the obstacles in environment, but instead maps specifically defined landmarks. Creates a more compact representation; however it requires a means to concretely say that a certain feature matches the given definition.

- Topological: shows only the connections between places. The positions indicated may be vastly different from their true locations. More compact.
- Geometric: the model used for most robot-applications. Creates a true-location-based mapping of the environment.

Known vs. unknown correspondence



Most approaches assume to have perfect data associations. This is unrealistic, but in fairness it is quite hard to track all possible data associations. A compromise would be to take into account a defined number of possible alternate estimations at each point.

Static vs. dynamic environment

There are approaches which explicitly allow the modelling of a dynamic environment, instead of taking the static assumption.

Small vs. large uncertainty

This is related to how much uncertainty the algorithm is willing and able to take into account at each stage and location.

Active vs. passive SLAM

Passive SLAM assumes that there is an external data stream being input, and simply follows it. In active SLAM, the robot can be thought of as exploring the environment on its own.

Any-time and any-space SLAM

This gives a pre-defined limit to the memory and time allowed for the robot to make its estimations at each stage.

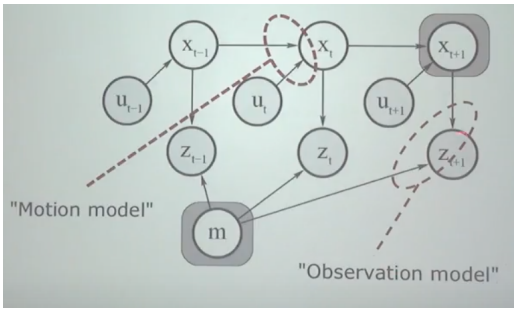
Single-robot vs. multi-robot SLAM

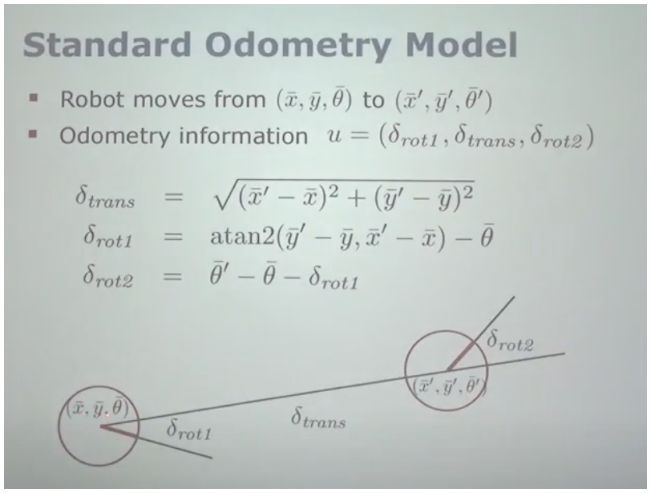
With more robots, data associations can be sent from one robot to another, increasing accuracy.

1.7 Motion and observation models

The motion model works to estimate the probability of a certain x_t given all data collected up to the point t. The observation model estimates the likelihood of a certain observation being correct, given knowledge of the robot's pose and currently known landmarks.

Motion model





Here, the motion model effects the transformation from the first pose to second is by a first rotation, a translation, and then a second rotation.

Observation model

This refers to what one expects the world map to look like, given current estimations. It includes both Gaussian and non-Gaussian models.

2 Homogeneous coordinates

2.1 Motivation

- Cameras generate a projected image of the world
- Euclidean geometry is quite sub-optimal when it comes to describing the central projection
- Projective geometry is an alternative algebraic representation of geometric objects and transformations

Sensors may sometimes not measure the distance to obstacles, but instead just the orientation of the obstacle with respect to the heading of the robot. For example, cameras know which pixels correspond to which angular orientation, and thus generate a projection of the 3D world onto a 2D image, without knowing exactly how far each point is.

Such an approach is quite hard to implement via Euclidean geometry due to complexities in obtaining the correct coordinates. Hence to make the math simpler, projective geometry is used (it does not change the relations between the objects and the surrounding space).

2.2 Homogeneous coordinates

- Points at infinity can be represented using finite coordinates - which is much harder to do in Euclidean geometry
- A single matrix can be used to represent affine as well as projective transformations

Affine transformations include rotations, translations, shearing and scale changes - being able to implement all of these in the same way, using a single matrix, is quite convenient. This is one of the main reasons for using homogeneous coordinates (HC).

Definition

The representation X of a geometric object is homogeneous if X and λX represent the same object for $\lambda \neq 0$.

$$X = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

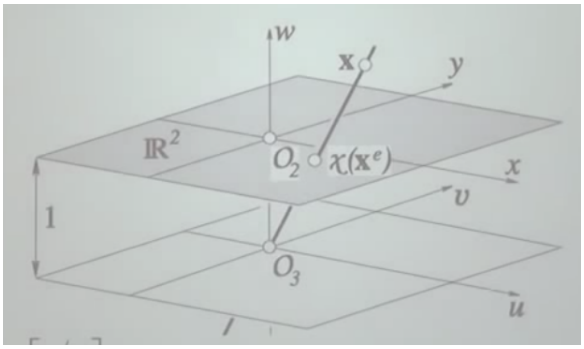
That is, we have some vector X, and this vector, as well as any scalar multiple of it, should refer to the same object.

Homogeneous to Euclidean

$$X = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, X = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} u/w \\ v/w \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} u/w \\ v/w \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Essentially, to map from the Euclidean space to HC, one simply adds a new dimension to the Euclidean 2D vector, which takes the value 1. For the reverse process, we normalize the first two dimensions using the third dimension.



Whichever Z-plane the HC point lies at on the line shown, it will ultimately intersect the base 2D plane at the same point. This is the principle of using HC - any point on that line can be used to represent the original required point.

Centers of the coordinate system (for 2D and 3D)

$$O_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad O_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Infinitely distant objects

Using HC, it is now possible to explicitly model infinitely distant points with finite coordinates:

$$X_\infty = \begin{bmatrix} u \\ v \\ 0 \end{bmatrix}$$

Being able to do this is an extremely useful tool when working with bearing-only sensors such as cameras which don't measure distances as such. Such a model also makes it much easier to check if lines are parallel or orthogonal, without having to go into the rigours of their representations.

3D points

This concept of transformation is analogous for 3D points as well, making use of a fourth dimension here.

$$X = \begin{bmatrix} u \\ v \\ w \\ t \end{bmatrix} = \begin{bmatrix} u/t \\ v/t \\ w/t \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} u/t \\ v/t \\ w/t \end{bmatrix}$$

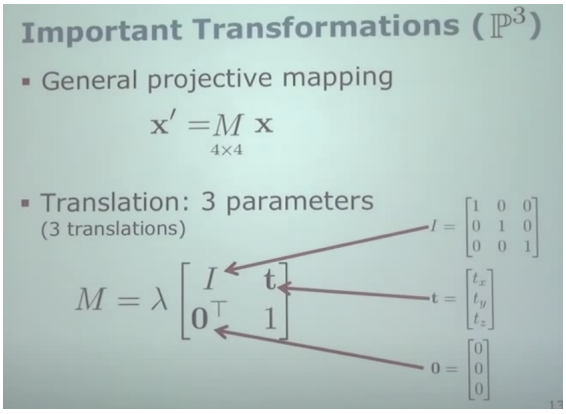
2.3 Transformations

One can now represent all required transformations using matrices. The only condition is that the matrix used should be invertible, which will be taken care of if the matrices are designed in the ways shown in this section.

A projective transformation is an invertible linear mapping. As already discussed, the λ used in the various matrices used to effect the transformations which will be shown, can take any value.

Matrices can thus be used to represent the various types of affine transformations mentioned earlier in a simple and efficient manner. Thus using HC to effect these is hugely beneficial, rather than having to add and multiply together separate vectors in a messy manner as is done with Euclidean coordinates.

Translation

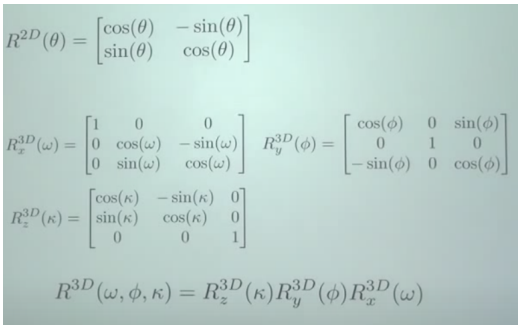


Here the vector $\mathbf{t}$ represents the required translation.

Rotation

$$M = \lambda \begin{bmatrix} R & 0 \\ 0^T & 1 \end{bmatrix}$$

The rotation matrix is the usual one used to rotate around an axis in Euclidean coordinates - it has three parameters for the 3D coordinate system due to the presence of three rotation axes.



The κ matrix effects the rotation around the X axis. Similarly, the ϕ matrix is for the Y axis, and the ω matrix is for the Z axis. Multiplying these together as shown effects all three rotations simultaneously.

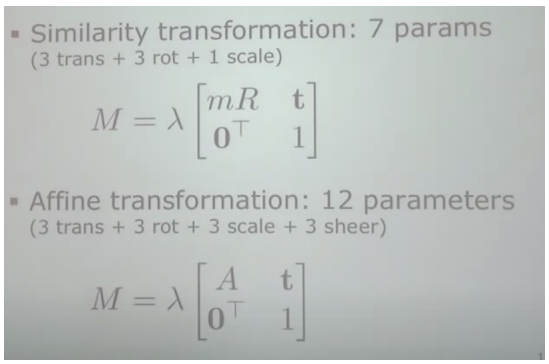
Rigid body transformation

Such a transformation has 6 parameters: 3 for rotation and 3 for translation. This is the main type of transformation which is used in robot mapping.

$$M = \lambda \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix}$$

where R and t represent the rotation matrix and translation vector respectively.

Other important transformations



The parameter m in the first equation above is used to scale the object being transformed. It cannot be pre-multiplied with the entire matrix, otherwise it would be scaled away by the λ.

Inverting a transformation

$$X' = MX$$
$$X' = M^{-1}X$$

Note: M is always invertible, given it is constructed according to the methods that have been shown.

Chaining transformations

The matrix product is not commutative, so one must be careful in which order the transformations are chained.

$$X' = M_1 M_2 X \neq M_2 M_1 X$$

3 The Bayes Filter

3.1 State estimation

The Bayes filter is a method of doing state estimation. One has observation data (z), as well as control data - the set of commands sent to the robot (u). Using these, we want to estimate the state of the world - the pose of the robot, the location of landmarks, etc.

$$Goal : p(x|z, u)$$

If we have no data yet, we begin with a uniform distribution - every state is equally likely. As one acquires more data and executes actions, one gets more certain about the state. Ultimately the aim is to obtain a peak-type distribution of the state.

The probability here is estimated using Bayes rule. One observation and one control are integrated at a time, recursively, to obtain the state of the system.

3.2 Recursive Bayes Filter

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t})$$

Here, 'bel' or 'belief', refers to the estimation of the probability distribution of x_t, given a sequence of sensor commands and a sequence of observations made up to that point in time t. Applying Bayes rule to this distribution, we get

$$bel(x_t) = \eta * p(z_t|x_t, z_{1:t-1}, u_{1:t}) * p(x_t|z_{1:t-1}, u_{1:t})$$

where η is a normalizing term.

In this new definition of *bel*, Bayes rule has been simply applied to make z_t the dependent variable in the probability estimation. Now, one applies the Markov assumption in order to simplify the first term.

The Markov assumption: given you know the current state of the world, you can estimate the probability distribution of the current distribution without considering any previous controls executed or observations made.

Applying this assumption to our definition of *bel*, we now get

$$bel(x_t) = \eta * p(z_t|x_t) * p(x_t|z_{1:t-1}, u_{1:t})$$

Now, looking at the second term of *bel*, it estimates the current state of the system, given observations up to time $t-1$, plus all the commands executed up

to that point. Thus it is like having a complete state estimate up to $t-1$, plus executing a motion command.

Using the law of total probability, we introduce the variable x_{t-1} to represent the state of the system at the time $t-1$, the just previous time step. The second term will thus turn into an integral.

$$bel(x_t) = \eta * p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, z_{1:t-1}, u_{1:t}) * p(x_{t-1}|z_{1:t-1}, u_{1:t}) dx_{t-1}$$

This transformation has been effected based on the rule of total probability, which makes use of a continuous known second probability distribution to estimate the first required one as follows:

$$P(A) = \int_B P(A|B) * P(B) dB$$

Now the Markov assumption is once again applied, this time to the first term of the integral - if one wants to know the current state of the world, and given that one knows the previous time step state, everything seen or done before that time step can be ignored.

So all the observations can be gotten rid of, and all the commands can be gotten rid of too, *except* u_t, as we need the current command to update the state from x_{t-1} to x_t. The equation now changes to

$$bel(x_t) = \eta * p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) * p(x_{t-1}|z_{1:t-1}, u_{1:t}) dx_{t-1}$$

So essentially, the new first term of the integral indicates that given one knows the previous time step's state, and the current command is executed, a probability distribution for the current state can be obtained.

Next, turning to the second term of the integral, one can assume that one does not need u_t to estimate x_{t-1}, as u_t will be executed in the future, not at the current time.

However, this is not always the case - knowing what command the robot will execute in the future can in certain circumstances allow one to make a better prediction of the current state.

For example, if one has to make an estimate between a pose which has an obstacle $50cm$ in front, and a pose which has its path clear, and one knows that the next command is to move $1m$ ahead, then the current pose is more likely to be closer to the latter option of the estimate.

Yet, this possibility is ignored - the assumption is made that the command executed in the future has no relevance to the current state of the system.

$$bel(x_t) = \eta * p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) * p(x_{t-1}|z_{1:t-1}, u_{1:t-1}) dx_{t-1}$$

Now, looking again at the second term of the integral, and looking back to the very first definition of *bel* (i.e. $bel(x_t) = p(x_t|z_{1:t}, u_{1:t})$ the second term is clearly of the same form as the definition, only defined for $t-1$ instead of t. And thus we obtain the recursive term in this definition.

$$bel(x_t) = \eta * p(z_t|x_t) \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) * bel(x_{t-1}) dx_{t-1}$$

So one now has a recursive update scheme which allows one to estimate the current state of the system based on the previous state x_{t-1}, the current motion command executed u_t, and the current observation obtained z_t.

This is the spirit of online SLAM - not needing any command or observation made in the future (taking into account the assumptions made).

3.3 Prediction and correction

The Bayes filter can be written as a two step process.

Prediction step

$$\overline{bel}(x_t) = \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) * bel(x_{t-1}) dx_{t-1}$$

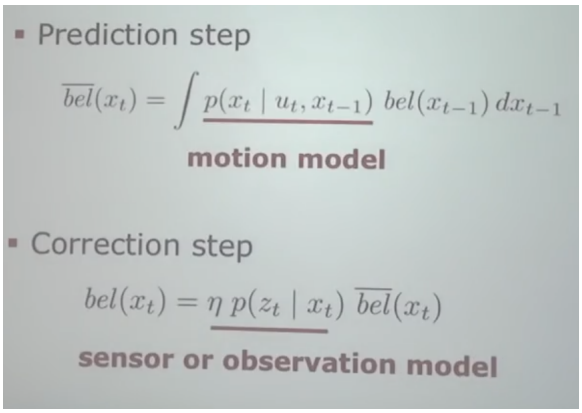
The prediction step takes into account the command executed. It entails integration over all the possible states in the estimation made for x_{t-1}.

Correction step

$$bel(x_t) = \eta * p(z_t|x_t) * \overline{bel}(x_t)$$

The correction step takes into account the observation made, and increases the likelihood of those states found in the prediction step which agree more with the sensors' observations. The normalizing term ensures that the sum or integral of all possible states equals to 1.

The prediction and correction steps also connect to the motion and observation models as such.



3.4 Different realizations

The Bayes filter is a framework for recursive state estimation, but it doesn't specify exactly how we should go about calculating the defined integrals, or which assumptions to take into account while doing so.

Properties that influence the way the filter is implemented include:

- Linear vs. non-linear models for motion and observation
- Type of distribution: e.g. Gaussian vs. uniform
- Parametric vs. non-parametric filters

Hence, there are several variants of the Bayes filter. Two important ones which will be discussed here are the Kalman filter and the Particle filter.

Kalman filter

- Gaussian
- Linear or linearized models

Particle filter

- Non-parametric
- Arbitrary models (sampling required)

The Kalman filter is highly optimised for systems that meet its specific requirements, and gives quite precise estimations. However, a robot's motion is often non-linear, and so in order to be able to take into account a wider range of states and state changes, the Particle filter is frequently used, though it has an increased computational cost.

3.5 Motion model

$$\overline{bel}(x_t) = \int_{x_{t-1}} \underline{p(x_t|x_{t-1}, u_t)} * bel(x_{t-1}) dx_{t-1}$$

To recap, this model looks into how to estimate the current state of the system, given the previous state and the command that was just executed.

Robot motion is inherently uncertain. The requirement is to be able to model the motion, given the uncertainty, since even small errors in the odometry information become big when accumulated over time.

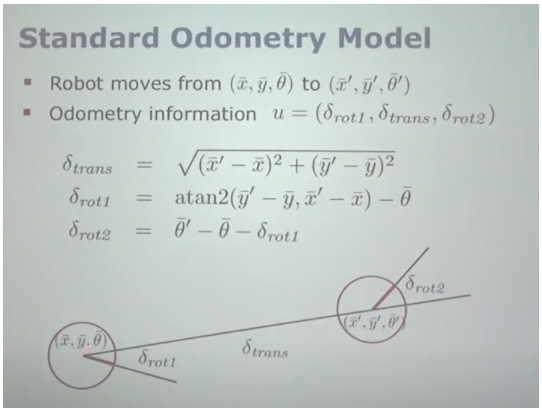
Probabilistic motion models

Such models specify a posterior probability that action u_t carries the robot from x_{t-1} to x_t:

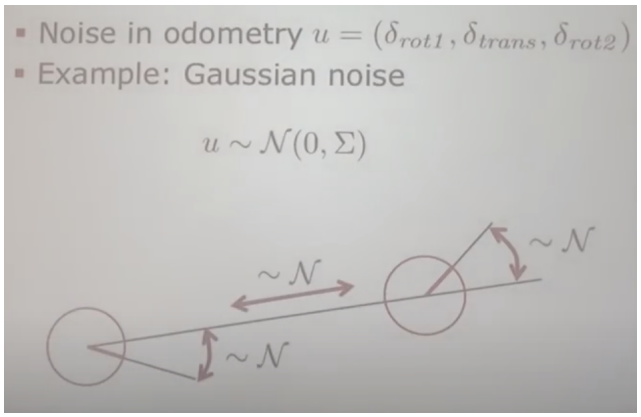
$$p(x_t|u_t, x_{t-1})$$

There are two different kind of models which can implement such a distribution.

1. **Odometry-based model**: encoders are attached to the wheels of the robot, and they count the revolutions of the wheels. This gives one an estimate of where the robot is going. However, it can be inaccurate when the wheel dimensions don't match or the ground is uneven, causing drift. This is the easier-to-handle model of the two.



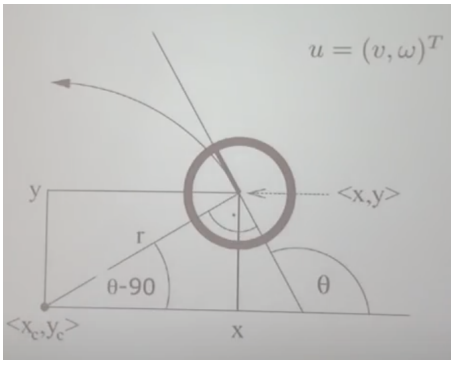
   Suppose one is introducing Gaussian errors to the individual three components (rotation, translation, rotation) in the system.



However, the net error produced does not have a Gaussian distribution, due to the non-linear relations between each step. This leads to the production of banana-shaped distributions, which are better represented as histograms. This is the standard model which is used in most cases.

2. **Velocity-based model**: encoders are not available. Velocity commands are instead sent to the system, and it is assumed that the system follows these commands to a certain accuracy. Used more in systems such as flying vehicles, or humanoid bots with legs.

   This model assumes that the motion command sent to the robot consists of two velocities - translational and rotational. The ideal result is the robot driving along a circular arc for short time intervals, as shown.



The model assumes that there only a discrete number of opportunities to change the velocities, due to the discrete time steps at which the commands are sent to the robot. However, this is not always the case - there will be delays in executing the commands, so the time intervals will not always match - yet this is the assumption that is followed.

- Robot moves from (x, y, θ) to (x', y', θ')
- Velocity information $u = (v, w)$

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v}{w}sin\theta + \frac{v}{w}sin(\theta + \omega\Delta t) \\ \frac{v}{w}cos\theta - \frac{v}{w}cos(\theta + \omega\Delta t) \\ \omega\Delta t \end{pmatrix}$$

where $\theta + \omega\Delta t$ is the current orientation of the robot.

Comparing the velocity-based model to the odometry-based model, the former has a translational and a rotational velocity, whereas the latter has a rotation, translation, and then another rotation. So essentially the difference is two parameters vs. three parameters.

Thus, in the velocity-based model, the circle constrains the final orientation. So one needs an additional term to account for this orientation. The fix is to introduce an additional noise term on the final orientation.

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v}{w}sin\theta + \frac{v}{w}sin(\theta + \omega\Delta t) \\ \frac{v}{w}cos\theta - \frac{v}{w}cos(\theta + \omega\Delta t) \\ \omega\Delta t + \gamma\Delta t \end{pmatrix}$$

The γ term introduced thus indicates how much rotation is required in the end to achieve the final orientation.

The probability distributions produced by the velocity-based model are similar to the odometry-based ones. The main difference is that the estimates of the velocity-based models are much more noisy, hence the odometry-based model gives more accurate estimates.

3.6 Sensor model

$$bel(x_t) = \eta * \underline{p(z_t|x_t)} * \overline{bel}(x_t)$$

This model obviously depends on which sensor being used. For example, the laser rangefinder, which gives the distance to the closest obstacle in a certain direction, is vastly different in the implementation of its model from a camera or a radar. The sensor first discussed here is the laser rangefinder (LR).

The LR typically has a mirror which rotates and reflects a laser beam, and a time of flight sensor which measures the time taken for the beam to bounce back from any obstacle to the receiver. So one obtains proximity measurements for various angular intervals at short time intervals.

Model for laser scanners

- Scan z consists of K measurements.

$$z_t = z_t^1, \ldots, z_t^k$$

- Individual measurements are independent, given the robot position.

$$p(z_t|x_t, m) = \prod_{i=1}^{k} p(z_t^i|x_t, m)$$

Each observation consists of k proximity measurements. By knowing the orientation of the mirror, one can deduce the positions of the obstacles detected.

Additionally, the measurements in each direction are assumed to be independent of each other. The probability distribution over the whole scan is taken to be the product of the distributions of the individual beams, which are modeled in each direction given that the robot knows its current state and what the environment looks like.

There are different ways for describing the individual beam distributions, which are so-called beam-based models. The simplest of these is the beam-endpoint model.

Beam-endpoint model

Suppose the robot sends out a beam in a particular direction. Depending on where the endpoint of the beam is calculated to be located, ignoring the data that the map provides in that specific direction, the position of the nearest obstacle in that direction is newly estimated.

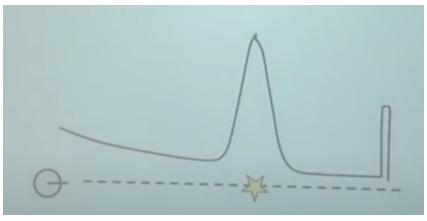
From a physics point of view, this model may not seem quite sound. However, it works surprisingly well in practice. It is also extremely efficient to compute. This is because if one has their map, then this model simply creates a kind of Gaussian distribution indicating the position of each obstacle. It essentially boils down to looking up a value in an array - the further away one is from the obstacle, the lower the probability value retrieved, and vice versa.

The typical maps created by this are occupancy grid maps, which can also be converted to likelihood fields, where the brighter the map is at a point, the greater the probability of an obstacle being present at that location.

Ray-cast model

This is more expensive to compute, but physically more accurate. Suppose the map indicates that an obstacle is present at a certain location. The question the model asks is, what is the likelihood that one would measure a certain length, given it is known that the obstacle is four metres away?

The resulting distribution is a kind of mixture of four distributions:



- A gaussian distribution around the position of the actual obstacle
- A component which describes exponential decay - this allows one to ac-

count for dynamic obstacles, which only affect the primary obstacle's location up to the point where the obstacle itself is located

- A peak at the back, which is due to the maximum range reading of the sensor - the maximum distance the sensor can detect obstacles till
- A uniform distribution which accounts for any random remaining obstacles (will be covered in more detail later).

Thus, this model results in a distribution that is a lot more physically plausible than the one produced by the beam-endpoint model.

Model for perceiving landmarks with range-bearing sensors

- Range bearing: $z_t^i = (r_t^i, \phi_t^i)^T$
- Robot's pose: $(x, y, \theta)^T$
- Observation of feature j at location $(m_{j,x}, m_{j,y})^T$

$$\begin{pmatrix} r_t^i \\ \phi_t^i \end{pmatrix} = \begin{pmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ atan2(m_{j,y} - y, m_{j,x} - x) - \theta \end{pmatrix} + Q_t$$

The final model aims to give positions of distinct, defined landmarks using distance and orientation parameters. In these equations,

- r is the distance of the landmark measured by the LR
- ϕ is the orientation of the landmark with respect to the heading of the robot
- $(x, y, \theta)^T$ gives the current position of the robot
- $m_{j,x}, m_{j,y}$ give the location of the landmark in x, y according to the current map m
- Q_t refers to some noise; may be Gaussian or not

So for dense maps, the beam-endpoint and ray cast models will work better, and if one is working with defined landmarks, the perceiving model will be more suitable.

Altogether, given one is working with a particular odometry model, and has reached a certain position following the execution of a command, using any of these models allows an observation probability distribution to be estimated, and thus the Bayes filter to be applied.

4 Extended Kalman filter

The Kalman filter is a specific implementation of the Bayes filter, and it along with its extended version, comprise two applications of the Kalman paradigm. It is one of the most frequently used Bayes filters. When one has Gaussian distributions and linear models, it is indeed the optimal estimator. Although such assumptions are rarely satisfied in reality, this filter is an important example in the application of SLAM.

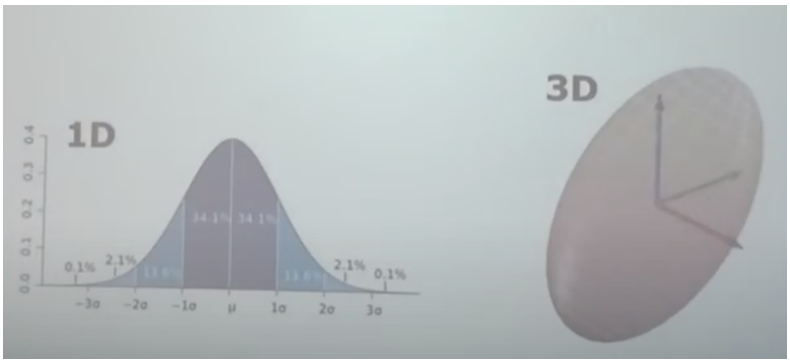
4.1 Gaussian distributions

Since this model assumes all distributions are Gaussian, here is a look at the algebraic and graphical forms of the Gaussian distribution.

$$p(x) = det(2\pi\Sigma)^{-\frac{1}{2}} exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu))$$

where

- μ is the mean estimate or mode of the distribution
- Σ is the co-variance matrix - the higher the values in the matrix, the higher the uncertainty. The matrix should be invertible, as evident from the equation



In the 1D plot, the range on the x-axis from -3σ to 3σ covers approximately 99% of the probability mass, so most events will fall in this area. This distribution can be represented in 2D by an ellipse, and 3D by an ellipsoid.

Marginalization and conditioning

Given

$$x = \begin{pmatrix} x_a \\ x_b \end{pmatrix} \qquad p(x) = \mathcal{N}$$

The marginals are Gaussians

$$p(x_a) = \mathcal{N} \qquad p(x_b) = \mathcal{N}$$

as well as the conditionals.

$$p(x_a|x_b) = \mathcal{N} \qquad p(x_b|x_a) = \mathcal{N}$$

Here, given that the distribution has two variables: x_a and x_b, which both have Gaussian distributions - the corresponding marginal, conditional, as well as convolutional models, then all have Gaussian distributions too.

Marginalization

Given

$$p(x) = p(x_a, x_b) = \mathcal{N}(\mu, \Sigma)$$
$$with \qquad \mu = \begin{pmatrix} \mu_a \\ \mu_b \end{pmatrix} \qquad \Sigma = \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix}$$

The marginal distribution is

$$p(x_a) = \int p(x_a, x_b) dx_b = \mathcal{N}(\mu, \Sigma)$$
$$with \qquad \mu = \mu_a \qquad \Sigma = \Sigma_{aa}$$

This is how to marginalize out a variable - a relatively simple process. So if one has a high-dimensional Gaussian distribution, and wants the marginal for a small number of elements, all that is needed is to cut out the relevant part of the mean vector, and cut out the relevant part of the co-variance matrix.

Conditioning

Given

$$p(x) = p(x_a, x_b) = \mathcal{N}(\mu, \Sigma)$$
$$with \qquad \mu = \begin{pmatrix} \mu_a \\ \mu_b \end{pmatrix} \qquad \Sigma = \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix}$$

The conditional distribution is

$$p(x_a|x_b) = \frac{p(x_a, x_b)}{p(x_b)} = \mathcal{N}(\mu, \Sigma)$$
$$with \qquad \mu = \mu_a + \Sigma_{ab}\Sigma_{bb}^{-1}(b - \mu_b)$$
$$and \qquad \Sigma = \Sigma_{aa} - \Sigma_{ab}\Sigma_{bb}^{-1}\Sigma_{ba}$$

The conditional distribution here is normally distributed, but the mean and the co-variance matrix are much harder to calculate. The important term to

note is the Σ_{bb}^{-1}. This implies that if one has a high-dimensional Gaussian distribution, and wants to estimate a small quantity out of that, given that the rest is known, it is a very costly operation, because a large part of the original co-variance matrix needs to be operated on.

Also, if one has very little knowledge about the variable b, i.e. it has an extremely high uncertainty (and so a very high Σ), then when inverted the term Σ_{bb}^{-1} basically goes to zero. So the derived mean becomes dependent on only a.

4.2 Linear models

To use the Kalman filter, both the motion and observation models must be linear functions. They are represented as follows.

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t$$
$$z_t = C_t x_t + \delta_t$$

where

- A_t, B_t and C_t are time-varying matrices
- ϵ_t and δ_t are noise terms
- all remaining variables are the same as usual

The presence of the matrices makes both the equations linear (in more than one dimension). For example, the second equation becomes a linear mapping from the world state to the observation state.

The matrix A can be said to represent how the world state changes when no motion command is executed. So typically, it would be the identity matrix (causing no change in the robot's pose if it is at rest). However, in some cases, if an object is already in motion, then A would be different. Similarly, B is a representation of how the physics of the robot's motion affects its world state.

The two equations in a sense are indicative of the mean values for both models; the uncertainty must be taken into account and generated in a later step of the process.

Components of a Kalman filter

- A_t: an nxn matrix that describes how the state evolves from $t-1$ to t without controls or noise.
- B_t: an nxl matrix that describes how the control u_t changes the state from $t-1$ to t. Here, n is the dimensionality of the state, and l is the

dimensionality of the odometry command. In reality, B should be non-linear (e.g. containing terms of *sin*, *cos*, and the like), but since the requirement of the model is to be linear, it must be made linear in some way.

- C_t: a kxn matrix that describes how to map the state x_t to an observation z_t. Here, k is the dimensionality of the observation. In a sense, this matrix defines what one should expect to observe, given that the world is in the current state.
- ϵ_t and δ_t: random variables representing the process and measurement noise that are assumed to be independent and normally distributed, with co-variance matrices R_t and Q_t respectively.

Note that if nothing is known about the world, the corresponding Gaussian could be thought of as having a zero mean and a co-variance matrix with close-to-infinity entries - this would represent a kind of uniform distribution.

Linear motion model

Under the Gaussian noise assumption, with the linear requirement, how does the motion model now look?

$$p(x_t|u_t, x_{t-1}) = \eta * exp(-\frac{1}{2} * \mathbf{?})$$

To deduce the mean of the required Gaussian, consider that one knows the previous state of the system, and which command has just been executed. This allows one to compute for every possible pose, the current state, using the linear model just defined.

$$p(x_t|u_t, x_{t-1}) = det(2\pi R_t)^{-\frac{1}{2}} * exp(-\frac{1}{2} * (x_t - A_t x_{t-1} - B_t u_t)^T * R_t^{-1} * (x_t - A_t x_{t-1} - B_t u_t))$$

where R_t describes the noise of the motion.

Linear observation model

Using a similar approach to derive the Gaussian distribution using the linear approach here,

$$p(z_t|x_t) = det(2\pi Q_t)^{-\frac{1}{2}} * exp(-\frac{1}{2} * (z_t - C_t x_t)^T * Q_t^{-1} * (z_t - C_t x_t))$$

where Q_t describes the measurement noise. Thus, having described both models using Gaussian distributions, one can now plug them into the Kalman filter.

4.3 Putting the models together

Given an initial Gaussian belief, the new belief obtained is always Gaussian too. Knowing the truth of this statement, and referring back to the two main steps of the Bayes filter (prediction and correction):

$$\overline{bel}(x_t) = \int_{x_{t-1}} \underline{p(x_t|x_{t-1}, u_t)} * \underline{bel(x_{t-1})} dx_{t-1}$$

$$bel(x_t) = \eta * \underline{p(z_t|x_t)} * \underline{\overline{bel}(x_t)}$$

Since it is now known how to specify each of the distributions used in these equations, and also that each of them is Gaussian, it is evident that their sum (integral) will be Gaussian as well.

Kalman filter algorithm

1. Kalman-filter $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$:
2. $\overline{\mu}_t = A_t \mu_{t-1} + B_t \mu_t$
3. $\overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
4. $K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + Q_t)^{-1}$
5. $\mu_t = \overline{\mu}_t + K_t(z_t - C_t \overline{\mu}_t)$
6. $\Sigma_t = (I - K_t C_t)\overline{\Sigma}_t$
7. return μ_t, Σ_t

Lines 2 and 3 are the prediction steps; lines 4 and 6 are the correction steps. The various expressions are derived by replacing the Gaussians produced in the previous section into the two integrals at the beginning of the current section. The overline bar over certain symbols indicates a mean.

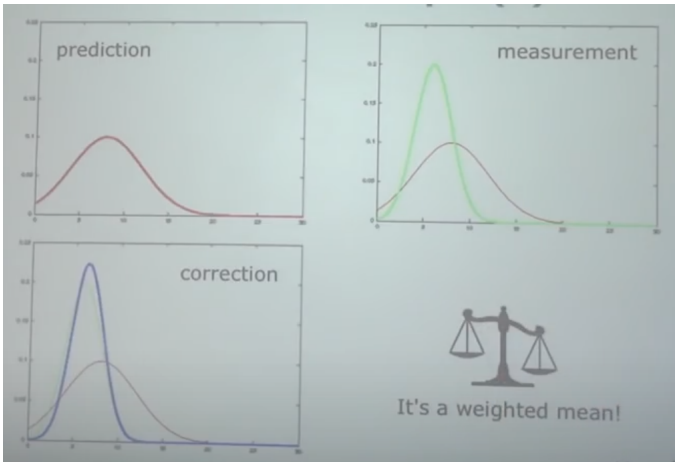
In line 3, one sees that the new uncertainty is the old uncertainty plus the uncertainty created by the current motion step. The matrix A is used here partly to scale the system.

Going to the correction step and looking at the original integral expression, it can be seen that two Gaussians are being multiplied in this expression. When two Gaussians are multiplied together, the mean of the new Gaussian produced is the weighted mean of the means of the individual Gaussians, the weights being the uncertainties of the individual Gaussians. Basically, this means that if there is one Gaussian that is very certain and one that is very uncertain, the product of both will be a Gaussian which is very close to the certain distribution.

Moving to the term K_t, it is referred to as the Kalman gain. It represents the trade-off between motion uncertainty and observation uncertainty. Looking at some examples to understand this,

- Suppose the measurement noise, Q_t is zero. This makes $K_t = C^{-1}$, which when plugged in to the next equations, ultimately leads to μ_t being dependent fully and only on z_t - that is, the zero uncertainty in the observation model has made the estimated current state completely based on the current observation made.
- Conversely, if Q_t tends to infinity, then K_t tends to zero. This ultimately leads to μ_t having zero dependence on the observation made.

An example



- red curve: prediction distribution
- green curve: observation distribution
- blue curve: final distribution with correction

The corrected curve in this case is closer to the observation/measurement curve in terms of the mean and variance, as the observation has less uncertainty here.

As the robot makes its next movement, the odometry uncertainty will remain the same while the observation uncertainty will increase due to the uncertainty of the previous state. Hence the corrected curve obtained now will have a greater spread (more uncertainty) than the previous corrected curve.

4.4 Kalman filter assumptions

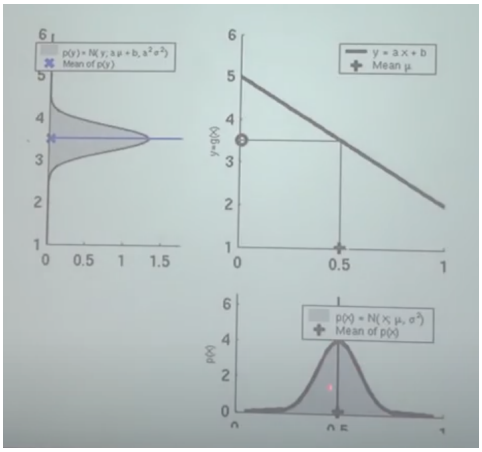
What if the system does not follow the assumptions of Gaussian distributions and noise, and linear models, as in reality? Most realistic problems in robotics involve non-linear functions. This would give rise to equations like the following:

$$x_t = g(u_t, x_{t-1}) + \epsilon_t \qquad z_t = h(x_t) + \delta_t$$

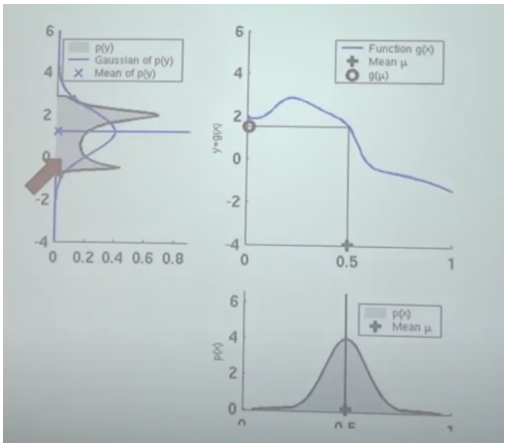
where g and h are non-linear functions.

The linearity assumption

An important property that the Kalman filter makes use of, is that a Gaussian distribution, when mapped through a linear function, results in another Gaussian, as shown below.



However, when one maps it through a non-linear function, a vastly different distribution is obtained:



Thus the application of the non-linear function in a sense destroys the Gaussian distribution, which the Kalman filter can no longer be applied to. The way to resolve this is to use local linearization, which leads to an extension of the Kalman filter.

The extended Kalman filter fixes this problem of non-linear functions by linearizing those functions, and then proceeding with the steps of the normal filter.

4.5 EKF linearization: first order Taylor expansion

Prediction step:

$$g(u_t, x_{t-1}) \approx g(u_t, \mu_{t-1}) + \frac{\delta g(u_t, \mu_{t-1})}{\delta x_{t-1}} * (x_{t-1} - \mu_{t-1})$$

Correction step:

$$h(x_t) \approx h(\overline{\mu}_t) + \frac{\delta h(\overline{\mu}_t)}{\delta x_t} * (x_t - \overline{\mu}_t)$$

where

$$\frac{\delta g(u_t, \mu_{t-1})}{\delta x_{t-1}} =: G_t \qquad and \qquad \frac{\delta h(\overline{\mu}_t)}{\delta x_t} =: H_t$$

are Jacobians. A Jacobian matrix is a non-square matrix mxn in general.

Note that given a vector-valued function,

$$g(x) = \begin{pmatrix} g_1(x) \\ g_2(x) \\ \vdots \\ g_m(x) \end{pmatrix}$$

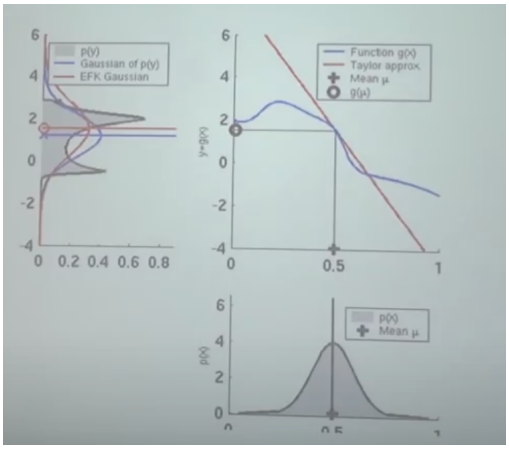
the Jacobian matrix is defined as

$$G_x = \begin{pmatrix} \frac{\delta g_1}{\delta x_1} & \frac{\delta g_1}{\delta x_2} & \cdots & \frac{\delta g_1}{\delta x_n} \\ \frac{\delta g_2}{\delta x_1} & \frac{\delta g_2}{\delta x_2} & \cdots & \frac{\delta g_2}{\delta x_n} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\delta g_m}{\delta x_1} & \frac{\delta g_m}{\delta x_2} & \cdots & \frac{\delta g_m}{\delta x_n} \end{pmatrix}$$

where m is the dimension of the vector function, and n is the number of variables involved. It is a kind of generalization from the 1D derivative, for the higher-dimensional case.

Thus, the Jacobian gives the orientation of the tangent plane to the vector-valued function at a given point. It can also be said to generalize the gradient of a scalar-valued function.

Hence the presence of the Jacobian matrices in the prediction and correction steps linearizes the respective functions - but only at that specific linearization point. So for different linearization points, i.e. different states, the Jacobian will need to be recomputed.



The resulting curve is thus once again a Gaussian, represented by the red curve in the figure above. Hence, when we linearize the function at a certain point, the less the spread of the initial Gaussian about that point, the less the difference between the initial Gaussian (blue curve in final figure) and the Gaussian produced using the EKF linearization (red curve in final figure), i.e. the better the approximation is.

Applying EKF to the models

Motion model:

$$p(x_t|u_t, x_{t-1}) \approx det(2\pi R_t)^{-\frac{1}{2}} * exp(-\frac{1}{2} * (x_t - g(u_t, \mu_{t-1})) - G_t(x_{t-1} - \mu_{t-1})^T$$

$$*R_t^{-1} * (x_t - g(u_t, \mu_{t-1})) - G_t(x_{t-1} - \mu_{t-1}))$$

Observation model:

$$p(z_t|x_t) = det(2\pi Q_t)^{-\frac{1}{2}} * exp(-\frac{1}{2} * (z_t - h(\overline{\mu}_t) - H_t(x_t - \overline{\mu}_t))^T * Q_t^{-1} * (z_t - h(\overline{\mu}_t) - H_t(x_t - \overline{\mu}_t)))$$

where R_t and Q_t describe the motion noise and measurement noise respectively.

Extended Kalman filter algorithm

1. Extended-Kalman-filter $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$:
2. $\overline{\mu}_t = g(u_t, \mu_{t-1})$
3. $\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$

4. $K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + Q_t)^{-1}$

5. $\mu_t = \overline{\mu}_t + K_t(z_t - h(\overline{\mu}_t))$

6. $\Sigma_t = (I - K_t H_t)\overline{\Sigma}_t$

7. return μ_t, Σ_t

The only differences in this implementation of the Kalman filter are that the original linear functions have been replaced with the derived linearized functions, that is A_t has been replaced with G_t, and C_t has been replaced by H_t.

So the extended Kalman filter is the same as the standard Kalman filter, except that it uses linear approximations of non-linear functions. Also, G_t and H_t need to be recomputed at every time interval, contrary to A_t and C_t in the standard filter. Also, if g and h turn out to be linear functions, the implementation will give the exact same result as the standard filter does in this case.

The complexity of the filter can be represented as $O(k^{2.4} + n^2)$. The exponent of the first term is derived from the fact that a matrix inversion is required in the observation model, and that of the second term is derived from the fact that nxn square matrices are present. Hence, the larger the number of observation parameters, or world-state parameters, the more costly the calculation gets.

5 EKF SLAM

Now, one can start to implement the EKF SLAM method into the localization algorithm of the robot. Here, the map being considered will be in the form of landmark locations - things the robot can identify via its sensor readings or camera images, and calculate the corresponding poses for.

As usual, the Kalman filter will be used as a solution to the online SLAM problem:

$$p(x_t, m|z_{1:t}, u_{1:t})$$

5.1 State representation

Beginning with defining the state μ of the robot (assuming it is in the 2D plane), μ will be a vector having its first three dimensions x, y and θ describing the pose of the robot, and the remaining dimensions describing the x-y coordinates of the landmarks' positions.

$$x_t = (x, y, \theta, \qquad m_{1,x}, m_{1,y}, \dots m_{n,x}, m_{n,y})^T$$

Thus, a map with n landmarks will give rise to a $3 + 2n$ dimensional Gaussian distribution.

The belief will then be represented by:

$$\begin{pmatrix} x \\ y \\ \theta \\ m_{1,x} \\ m_{1,y} \\ \vdots \\ m_{n,x} \\ m_{n,y} \end{pmatrix} \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xm_{1,x}} & \sigma_{xm_{1,y}} & \dots & \sigma_{xm_{n,x}} & \sigma_{xm_{n,y}} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{ym_{1,x}} & \sigma_{ym_{1,y}} & \dots & \sigma_{ym_{n,x}} & \sigma_{ym_{n,y}} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} & \sigma_{\theta m_{1,x}} & \sigma_{\theta m_{1,y}} & \dots & \sigma_{\theta m_{n,x}} & \sigma_{\theta m_{n,y}} \\ \sigma_{m_{1,x}x} & \sigma_{m_{1,x}y} & \sigma_{\theta} & \sigma_{m_{1,x}m_{1,x}} & \sigma_{m_{1,x}m_{1,y}} & \dots & \sigma_{m_{1,x}m_{n,x}} & \sigma_{m_{1,x}m_{n,y}} \\ \sigma_{m_{1,y}x} & \sigma_{m_{1,y}y} & \sigma_{\theta} & \sigma_{m_{1,y}m_{1,x}} & \sigma_{m_{1,y}m_{1,y}} & \dots & \sigma_{m_{1,y}m_{n,x}} & \sigma_{m_{1,y}m_{n,y}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_{m_{n,x}x} & \sigma_{m_{n,x}y} & \sigma_{\theta} & \sigma_{m_{n,x}m_{1,x}} & \sigma_{m_{n,x}m_{1,y}} & \dots & \sigma_{m_{n,x}m_{n,x}} & \sigma_{m_{n,x}m_{n,y}} \\ \sigma_{m_{n,y}x} & \sigma_{m_{n,y}y} & \sigma_{\theta} & \sigma_{m_{n,y}m_{1,x}} & \sigma_{m_{n,y}m_{1,y}} & \dots & \sigma_{m_{n,y}m_{n,x}} & \sigma_{m_{n,y}m_{n,y}} \end{pmatrix}$$

where the column vector and the right-hand-side matrix are μ, the pose of the robot, and Σ respectively. Σ can be considered as having three main sections:

- The $3x3$ matrix in the upper left corner which is the co-variance matrix corresponding to the pose of the robot
- The $2nx2n$ matrix in the bottom right corner which holds the uncertainties of the landmarks' positions and their correlations
- The remaining isolated rectangular matrices which indicate the correlations between the robot's pose and the locations of the landmarks

A more compact representation of this belief of state is as follows:

$$\begin{pmatrix} x_R \\ m_1 \\ \vdots \\ m_n \end{pmatrix} \begin{pmatrix} \Sigma_{x_R x_R} & \Sigma_{x_R m_1} & \dots & \Sigma_{x_R m_n} \\ \Sigma_{m_1 x_R} & \Sigma_{m_1 m_1} & \dots & \Sigma_{m_1 m_n} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{m_n x_R} & \Sigma_{m_n m_1} & \dots & \Sigma_{m_n m_n} \end{pmatrix}$$

An even more compact representation could be (note $x_R \to x$):

$$\begin{pmatrix} x \\ m \end{pmatrix} \begin{pmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{pmatrix}$$

5.2 The filter cycle

1. State prediction: take the control command and estimate the new state of the robot given this command.
2. Measurement prediction: evaluating the function h at the predicted mean μ - i.e. what one is expected to observe given the current belief of where the robot is.
3. Measurement: making observations via sensor input.
4. Data association: understanding which landmark(s) correspond(s) to what the robot is currently observing, and obtaining the difference between the expected and actual observation.
5. Update: correcting the belief of state based on the observations.

5.3 State prediction

Assuming that the robot does not modify the environment (it only changes its own state, not that of the landmarks), when the robot receives a control command and makes a movement, only x_R (the robot's pose) must be updated.

In the co-variance matrix, the first row and first column of the above representation must be updated. The computational complexity of such an operation is O(n), i.e. linear in n, the number of landmarks. This is quite efficient for such an operation.

Once the control command execution is complete, the robot takes into account the current belief of its state, and computes the predicted measurements for the landmarks around it - it assumes their positions based on its current knowledge of the state.

Next, the real measurements are taken. The robot now must make a data association by deciding which landmark(s) the observation(s) correspond to, and calculate the difference(s) between its estimate and the actual measurement.

Based on this difference, the mean vector for the state can then be updated, leading to the entire co-variance matrix getting updated via the Kalman gain method. The complexity of this computation is $O(n^2)$. Hence, the sensor observation is much more costly than the initial state updation.

5.4 A concrete example

Initialization

The setup for the problem:

- Robot moves in the 2D plane (pose described by (x, y, θ)
- Velocity-based motion model
- Robot observes point landmarks (x,y)
- A range-bearing sensor: measures distance and orientation of the landmark with respect to the heading of the robot
- Known data associations: whenever one sees a landmark, one knows which landmark it corresponds to in the map
- Known number of landmarks

$$\begin{pmatrix} x_R \\ m_1 \\ \vdots \\ m_n \end{pmatrix} \begin{pmatrix} \Sigma_{x_R x_R} & \Sigma_{x_R m_1} & \dots & \Sigma_{x_R m_n} \\ \Sigma_{m_1 x_R} & \Sigma_{m_1 m_1} & \dots & \Sigma_{m_1 m_n} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{m_n x_R} & \Sigma_{m_n m_1} & \dots & \Sigma_{m_n m_n} \end{pmatrix}$$

To start off with, the mean vector and co-variance matrix must be initialized. The initial position of the robot can be taken as the origin, in order to define the frame of reference. Hence, we initialize the parameters as such:

$$x_R = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad \Sigma_{x_R x_R} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Now since the robot starts in its own reference frame, all the landmarks are unknown. Hence the uncertainty in their positions can be considered to be infinite. So the complete initialization of the belief would be as follows:

$$\mu_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad \Sigma_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \infty & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \infty \end{pmatrix}$$

Practically, the initial belief should only have three dimensions, that is the ones corresponding to the robot's pose, and the matrix sizes should be updated as and when the robot encounters a new landmark. However, this is algorithmically a lot more complex to implement.

Now, the steps of the EKF algorithm must be followed:

1. Extended-Kalman-filter $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$:
2. $\overline{\mu}_t = g(u_t, \mu_{t-1}$
3. $\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
4. $K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + Q_t)^{-1}$
5. $\mu_t = \overline{\mu}_t + K_t(z_t - h(\overline{\mu}_t))$
6. $\Sigma_t = (I - K_t H_t)\overline{\Sigma}_t$
7. return μ_t, Σ_t

Prediction step (motion)

Starting with line 2, one sees that the new mean estimate is the output of some function g that takes as input the previous mean and the control command.

Using the velocity based motion model, the goal is to update the state space based on the robot's motion. The robot's new pose can be estimated by:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v_t}{\omega_t} sin\theta + \frac{v_t}{\omega_t} sin(\theta + \omega_t \Delta t) \\ \frac{v}{\omega_t} cos\theta - \frac{v}{\omega_t} cos(\theta + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix}$$

where $\theta + \omega \Delta t$ is the current orientation of the robot. The right hand side of the above matrix equation is the definition of the function g, i.e.

$$g_{x,y,\theta}(u_t, (x, y, \theta)^T)$$

Theoretically, g should map the entire $2n + 3$ state vector to itself, but here it only considers the robot's pose, i.e. only the first 3 dimensions of the entire

state vector. So how does one map the output of g to the $2n+3$ dimensional space then? This can be done by modifying the equation as shown:

$$\begin{pmatrix} x' \\ y' \\ \theta' \\ \vdots \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \\ \vdots \end{pmatrix} + F_x^T \begin{pmatrix} -\frac{v_t}{\omega_t} sin\theta + \frac{v_t}{\omega_t} sin(\theta + \omega_t \Delta t) \\ \frac{v}{\omega_t} cos\theta - \frac{v}{\omega_t} cos(\theta + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix}$$

where F_x is the 3 x $(2n+3)$ matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \end{pmatrix}$$

Updating co-variance

Now one moves to the third line of the algorithm, for which G, the Jacobian of g needs to be computed. (As seen, R_t is the uncertainty of the motion model, which is assumed to be given.)

It is known that g only affects the position of the robot and not the landmarks. Since the position of the landmarks are not changing in the update step, the portion of G_t that represents them will be a $2n$ x $2n$ identity matrix.

$$G_t = \begin{pmatrix} G_t^x & 0 \\ 0 & I \end{pmatrix}$$

where G_t^x is the 3x3 Jacobian of the motion, and I is the $2n$ x $2n$ identity matrix.

To find G_t^x, one needs to derive the non-linear function which maps the odometry information to the state update.

$$G_t^x = \frac{\partial}{\partial (x,y,\theta)^T} \left[\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v_t}{\omega_t} sin\theta + \frac{v_t}{\omega_t} sin(\theta + \omega_t \Delta t) \\ \frac{v}{\omega_t} cos\theta - \frac{v}{\omega_t} cos(\theta + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix} \right]$$

$$= I + \frac{\partial}{\partial (x,y,\theta)^T} \begin{pmatrix} -\frac{v_t}{\omega_t} sin\theta + \frac{v_t}{\omega_t} sin(\theta + \omega_t \Delta t) \\ \frac{v}{\omega_t} cos\theta - \frac{v}{\omega_t} cos(\theta + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix}$$

The remaining matrix has no dependencies on x and y, so the first two columns of the resulting derivative matrix will be zero.

$$= I + \begin{pmatrix} 0 & 0 & -\frac{v_t}{\omega_t} cos\theta + \frac{v_t}{\omega_t} cos(\theta + \omega_t \Delta t) \\ 0 & 0 & -\frac{v}{\omega_t} sin\theta + \frac{v}{\omega_t} sin(\theta + \omega_t \Delta t) \\ 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & -\frac{v_t}{\omega_t} cos\theta + \frac{v_t}{\omega_t} cos(\theta + \omega_t \Delta t) \\ 0 & 1 & -\frac{v}{\omega_t} sin\theta + \frac{v}{\omega_t} sin(\theta + \omega_t \Delta t) \\ 0 & 0 & 1 \end{pmatrix}$$

Looking at the G_t matrix as a whole now, it is clear that it is simply an identity matrix with two additional non-zero elements in the third column.

The presence of these two elements linearizes the dependencies on sin and cosine of the function. For linear movement these two would hence be zero.

Now that all the required quantities are known, the third step of the algorithm can be carried out in full:

$$\overline{\Sigma}_t \quad = \quad G_t \Sigma_{t-1} G_t^T + R_t$$

$$= \begin{pmatrix} G_t^x & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{pmatrix} \begin{pmatrix} (G_t^x)^T & 0 \\ 0 & I \end{pmatrix} + R_t$$

$$= \begin{pmatrix} G_t^x \Sigma_{xx} (G_t^x)^T & G_t^x \Sigma_{xm} \\ (G_t^x \Sigma_{xm})^T & \Sigma_{mm} \end{pmatrix} + R_t$$

This form of the equation now perfectly shows exactly which parts of the covariance matrix get updated via G_t^x.

Summarising the prediction step

$$F_x = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \end{pmatrix}$$

The matrix which maps from the low dimensional to the high dimensional space is defined.

$$\overline{\mu}_t = \mu_{t-1} + F_x^T \begin{pmatrix} -\frac{v_t}{\omega_t} sin\theta + \frac{v_t}{\omega_t} sin(\theta + \omega_t \Delta t) \\ \frac{v}{\omega_t} cos\theta - \frac{v}{\omega_t} cos(\theta + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix}$$

The predicted mean is updated via the non-linear function which updates the state based on translational and rotational velocities.

$$G_t = I + F_x^T \begin{pmatrix} 0 & 0 & -\frac{v_t}{\omega_t} cos\theta + \frac{v_t}{\omega_t} cos(\theta + \omega_t \Delta t) \\ 0 & 0 & -\frac{v}{\omega_t} sin\theta + \frac{v}{\omega_t} sin(\theta + \omega_t \Delta t) \\ 0 & 0 & 0 \end{pmatrix} F_x$$

The Jacobian G of the non-linear function g is computed.

$$\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + F_x^T R_t^x F_x$$

The change in the robot's pose does not affect the uncertainty in the landmark's positions, hence R_t is simply mapped back to the high-dimensional space via F_x at this point.

The correction step

To carry out this step, one would proceed along the following lines making the given assumptions:

- Data associations are known
- $c_t^i = j$: the ith measurement taken at time t observes the landmark with index j
- The landmark is initialized if previously unobserved
- The expected observation is computed
- The Jacobian of the measurement function h is computed
- The remaining process is completed with computation and application of the Kalman gain

Often, for data association, the nearest neighbour method is used - the algorithm tries to identify which landmark is the best fit for the observation it has just made, and if it cannot make such a connection, then the observation is, in a sense, discarded.

Range-bearing observations

A range-bearing observation takes the form of

$$z_t^i = (r_t^i, \phi_t^i)^T$$

If a landmark has not been observed, it's coordinates in the state get updated as shown:

$$\begin{pmatrix} \overline{\mu}_{j,x} \\ \overline{\mu}_{j,y} \end{pmatrix} = \begin{pmatrix} \overline{\mu}_{t,x} \\ \overline{\mu}_{t,y} \end{pmatrix} + \begin{pmatrix} r_t^i cos(\phi_t^i + \overline{\mu}_{t,\theta} \\ r_t^i sin(\phi_t^i + \overline{\mu}_{t,\theta} \end{pmatrix}$$

i.e. observed location of landmark j = estimated robot's location + relative measurement. Hence, via this calculation, the position of the landmark is obtained, using the assumptions that the measurement made by the sensors is correct and that the robot is at the right position while making the measurement.

The expected observation

Next, the expected observation of the landmark(s) must be updated based on the current estimate of the state.

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \overline{\mu}_{j,x} - \overline{\mu}_{t,x} \\ \overline{\mu}_{j,y} - \overline{\mu}_{t,y} \end{pmatrix}$$

Here, δ_x and $delta_y$ represent the difference in the x and y coordinates between the position of the robot and the position of the landmark.

$$q = \delta^T \delta$$

So q represents the square of the Euclidean distance between the robot and the landmark.

$$\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ atan2(\delta_y, \delta_x) - \overline{\mu}_{t,\theta} \end{pmatrix} = h(\overline{\mu}_t)$$

These quantities are then used to compute the tuple of distance and orientation that represents the predicted observation. Thus, h is obtained, the function that maps the current state to this observation.

Jacobian for the observation

The H_t^i of the low dimensional state space will first be computed. This is done due to the same principle as the previous Jacobian - all the remaining parts of the high dimensional matrix at this stage will be unchanged and hence their derivative will anyway be zero.

$$lowH_t^i = \frac{\partial h(\overline{\mu}_t)}{\partial \overline{\mu}_t}$$

This low dimensional H will be a five-element vector: $(x, y, \theta, m_{j,x}, m_{j,y})$. The two additional elements over the pose of the robot correspond to the coordinates of the landmark being observed.

$$lowH_t^i = \begin{pmatrix} \frac{\partial \sqrt{q}}{\partial x} & \frac{\partial \sqrt{q}}{\partial y} & \dots \\ \frac{\partial atan2(\dots)}{\partial x} & \frac{\partial atan2(\dots)}{\partial y} & \dots \end{pmatrix}$$

Looking more closely at how to compute the first element of this matrix:

$$\frac{\partial \sqrt{q}}{\partial x} = \frac{1}{2} \frac{1}{\sqrt{q}} 2\delta_x(-1)$$

$$= \frac{1}{q}(-\sqrt{q}\delta_x)$$

Performing similar computations for all the elements of the matrix, one obtains

$$lowH_t^i = \frac{\partial h(\overline{\mu}_t)}{\partial \overline{\mu}_t}$$

$$= \frac{1}{q}\begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & +\sqrt{q}\delta_x & +\sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x \end{pmatrix}$$

This low dimensional H now once again needs to be mapped back to the high dimensional space, with the help of another mapping function $F_{x,j}$:

$$F_{x,j} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 & 0 & \dots & 0 \end{pmatrix}$$

where the first section of zeros has $2j-2$ columns, and the second section has $2n-2j$ columns (i.e. the pair of 1s in the middle corresponds to the landmark's coordinates). So finally,

$$H_t^i = lowH_t^i * F_{x,j}$$

Summarising the correction steps

Now that H has been well defined, one can proceed with the next steps of the algorithm. Q_t, the sensor noise term, is something which depends on the specification of the sensor (not taken into account here yet).

Using all the quantities now known, the Kalman gain can be computed easily, and the remaining steps completed. Thus, to summarise the correction process:

$$Q_t = \begin{pmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\phi^2 \end{pmatrix}$$

For all observed features $z_t^i = (r_t^i, \phi_t^i)^T$ do

- $j = c_t^i$

- if landmark j is never seen before:

$$\begin{pmatrix} \overline{\mu}_{j,x} \\ \overline{\mu}_{j,y} \end{pmatrix} = \begin{pmatrix} \overline{\mu}_{t,x} \\ \overline{\mu}_{t,y} \end{pmatrix} + \begin{pmatrix} r_t^i cos(\phi_t^i + \overline{\mu}_{t,\theta} \\ r_t^i sin(\phi_t^i + \overline{\mu}_{t,\theta} \end{pmatrix}$$

- endif
- $$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \overline{\mu}_{j,x} - \overline{\mu}_{t,x} \\ \overline{\mu}_{j,y} - \overline{\mu}_{t,y} \end{pmatrix}$$
- $q = \delta^T \delta$
- $$\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ atan2(\delta_y, \delta_x) - \overline{\mu}_{t,\theta} \end{pmatrix} = h(\overline{\mu}_t)$$
- $$H_t^i = \frac{1}{q} \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & +\sqrt{q}\delta_x & +\sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x \end{pmatrix} * F_{x,j}$$
- Kalman gain, $\overline{\mu}_t$ and $\overline{\Sigma}_t$ are computed

endfor

$\mu_t = \overline{\mu}_t$

$\Sigma_t = \overline{\Sigma}_t$

return μ_t, Σ_t

Implementation notes

- The measurement update in a single step requires only one full belief matrix update (and one partial). The landmark information can also be updated simultaneously - it need not be iterated over - however, this is not optimal.
- Always normalize the angular components (be wary of the wraparound).

5.5 Loop closure

After a long traversal through an unknown environment, a robot may come back to a certain point that it had visited before. The ability to recognize and register that it has been at this place before is known as loop closure.

This process mainly involves data association, and may involve high uncertainties. There is a possibility that it is simply visiting a symmetric area of the environment, and this can cause ambiguities. Hence loop closure is a non-trivial decision.

However, the uncertainties tend to collapse after a loop closure (whether the closure was correct or not). This is because the motion uncertainty of the robot increases as it moves through the environment for a greater time. So coming back to a location visited long ago, and recognizing it as already visited, allows the smaller motion uncertainty of that previous time to propagate through the algorithm and reduce all the following uncertainties.

Thus, loop closing reduces the uncertainty in the robot and landmark estimates. This can be exploited when exploring an environment for the sake of better and more accurate maps.

However, wrong loop closures lead to filter divergence, due to the entirely wrong mean estimates and uncertainty estimates that are produced on such a decision.