## *Localization*

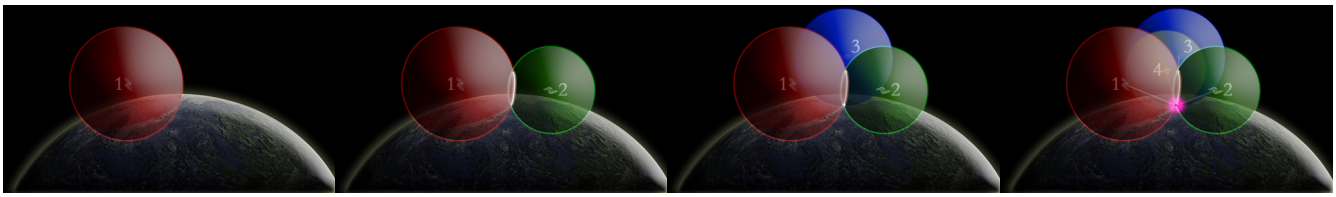
Most automatic systems including robots have to operate in complex environments, and in order to do that, the first necessary condition is for the robot to figure out their precise location within this environment. This problem is called localization, and needs to be addressed in a sound and robust manner.

You may wonder why we don't simply use a system like GPS to determine the location of a robot. GPS, at least in the versions available to regular users, is not precise enough. Let's see why that is.

### *How GPS works*

The principle of operation of GPS is fairly simple – each GPS satiellite emits a signal with information about the satellite's position and a timestamp for when the signal was sent. From the timestamp, a GPS received calculates the time-of-travel from the GPS satellite to the received. Time of travel can be used to determine the distance to the satellite – the receiver must be somewhere on a sphere centered at the satellite and whose radius was computed from the time of travel for the satellite's signal.

Given a set of 4 GPS satellites, an unambiguous location can be ontained in 3D as shown in the sequence of images below.




(Images from: https://oceanservice.noaa.gov/education/tutorial_geodesy/geo09_gps.html - Public Domain)

This is the nice theory – however, in practice the problem is incredibly complex and full of places where unknowns and uncertainties creep into the position computation:

- The orbit of the satellites is an imperfect ellipsoid, affected by gravitational variations as they move over Earth (which is not a perfect sphere, not does it have a uniformly distributed mass). Relativistic effects due to height variations during the satallite's orbit affect the satellite's clock, which in turn affects the time-stamp. All of these need to be compensated for as best as possible

- Atmospheric conditions affect the time of travel for the signal. Exact time of travel could be determined only if the signal was travelling in vacuum, but it's not. Depending on atmospheric conditions between the satellite and the received the signal will be delayed in ways that are hard to compensate.

- Often there is no direct line-of-sight to one or more of the GPS satellites. In cities, for example, buildings often get in the way and the received gets a signal that has been bounced off buildings and

other structures. This changes the time-of-travel (again in tiny yet significant ways), and this affects the distance computation.

- Synchronizing the satellite and received clocks is difficult – the system works only if the received can compute the time-of-travel accurately, and this requires the satellite clocks and the received clock to be synchronized. GPS satellites carry incredibly accurate atomic clocks, but GPS receivers don't have them. This is another source of uncertainty.

All of these effects need to be accounted for, and compensated for, if we are to obtain an accurate location fix. Modern GPS receivers do a lot of work and achieve a fairly good location fix when conditions are good, but even so there is a residual localization error that can be anywhere between a couple feet and several meters. This level of accuracy is often not sufficient if we aim to have a robot perform tasks requiring great accuracy, such as navigating around a room, picking up or moving objects, or keeping a car correctly position within a lane in a highway.

A very thorough reference for how GPS position is computed can be found here: https://www.telesens.co/2017/07/17/calculating-position-from-raw-gps-data/

There is another reason why we don't use GPS for robot localization: It doesn't work indoors.

***Robot Localization***

This leaves us with the following setup for the task we want to solve:

- Given a ***map*** of the environment the robot is operating in (the map is stored in a way that makes sense to the robot, what exactly it looks like depends on the application, as we will see)

- Use ***observations*** from the available sensors on the robot to accurately determine the robot's location within the environment

- The robot can perform ***actions*** (often this will mean move around, but it could also mean interacting with the environment in a way that is useful for localization)

- The localization process fully assumes the ***sensors are noisy***, but importantly the sensors' operation is not location dependent (i.e. they work just the same regardless of where the robot is within its environment)

- The ***map is static,*** this means that the properties of the environment that are useful for localization aren't changing or moving over time. Note this doesn't mean that the environment can't contain obstacles or other moving objects, it just means these moving features aren't noted on the map

At this point, we have to bring in the proper tools to work through this problem – we know that sensors are noisy, that the components that allow robots to perform actions are also noisy (e.g. requesting that a robot perform some action will result in a noisy version of this action). Therefore any

algorithm we put together to do the localization will have to account for ***uncertainty,*** and any proper framework for dealing with uncertainty will be ***probabilistic***.

This means that we can never be completely sure about a robot's location (or any other quantity we are estimating). Rather, we have a probabilistic quantity called ***belief***, that tells us how ***likely particular locations*** (or values for other quantities we are estimating) are. Probabilistic localization is the framework used for any realistic application involving robots.

In our concrete case, we want an algorithm to estimate

$$Bel(\vec{x_k})$$

which for any position ***x*** on the map of the robot's environment corresponds to the probability that the robot is at location ***x*** at a specific moment ***k*** in time. The reason for specifying a particular instant in time becomes clear when we see how ***Bel($x_k$)*** is estimated:

$$Bel(\vec{x_k}) = p(\vec{x_k}|\vec{d_0}, \vec{d_1}, \ldots, \vec{d_k})$$

The equation above states that our ***belief that the robot is at location $x_k$*** (a vector with coordinates) is given by the ***probability that the robot is at location $x_k$ given all the information we have accumulated*** from the moment the robot started the localization process, ***$d_0$,*** (a vector with any measurements and data useful to help the localization process) to the current instant ***$d_k$***.

***Important note:*** In what follows, we will focus on estimating the physical location of a robot so we assume ***$x_k$*** contains only the spatial coordinates of the robot, however, the process is general and in practice we will be estimating the set of ***state variables*** the robot requires for its operation. This may include the robot's heading direction, its travelling speed, and any other values the robot needs in order to carry out its task.

Estimating belief from the equation above quickly becomes too difficult. Information accumulates over time, and the conditional distribution over possible locations becomes impossible to keep track of. In order to simplify the belief estimation process, we will make a ***simplifying assumption***. We will invoke the ***Markov property*** to remove all dependence on previous time instants ***with the exception of the immediately previous one***. This is a reasonable thing to do because if we knew the robot's position at time ***k-1***, and we know what action(s) the robot took at that point in time, and we have access to the sensor measurements for the current time ***k***, then this provides all the information we need in order to estimate belief for time ***k*** at every possible location. This leaves us with:

$$Bel(\vec{x_k}) = p(\vec{x_k}|\vec{x}_{k-1}, a_{k-1}, \vec{s_k})$$

where ***$a_{k-1}$*** represents the action or actions the robot took at time instant ***k-1,*** and ***$s_k$*** is a vector with all current sensor measurements the robot has access to. This makes the estimation process tractable.

So, to summarize:

- The localization process uses a ***map***, ***sensor measurements***, ***past actions, and previous belief values*** to estimate the current likelihood for all possible locations in the map being the corrent position where the robot is at.

- The process is tractable because we use the ***Markov assumption*** to remove dependence on historical information beyond the immediately previous time instant.

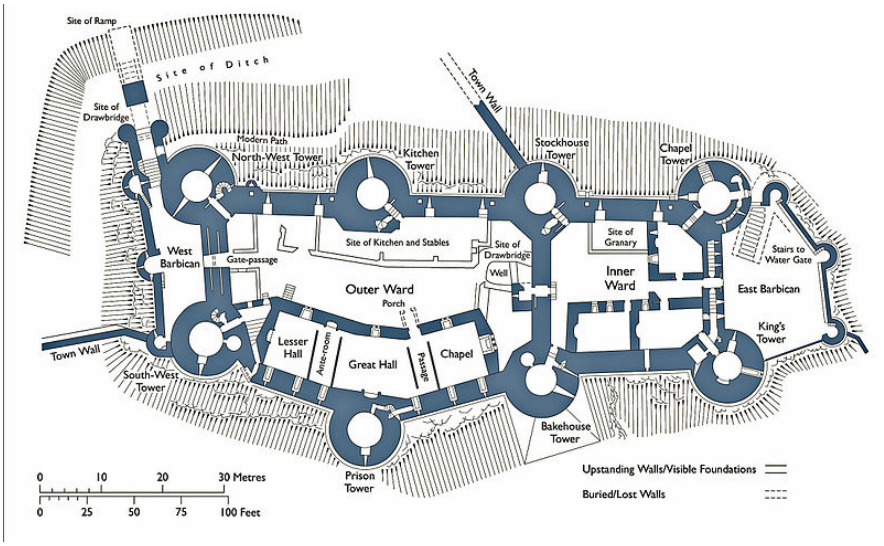
- Because of the noisy nature of robot sensors and robot actions, we expect the probability values to fluctuate over time. The localization process has to be carried out continuously. Even if we were quite certain about the robot's current location, without constantly adjusting for actions taken as well as accounting for sensor readings, we would quickly lose track of the robot's position.

In what follows, we will look at two of the major methods for estimating belief under the conditions set out above. Each of these has specific advantages as well as limitations, so it will be important to understand that while the underlying process is similar, the specific ways in which they differ are critical indeciding which of them you will need to implement for a particular sitiation or problem.

***Histogram Localization***

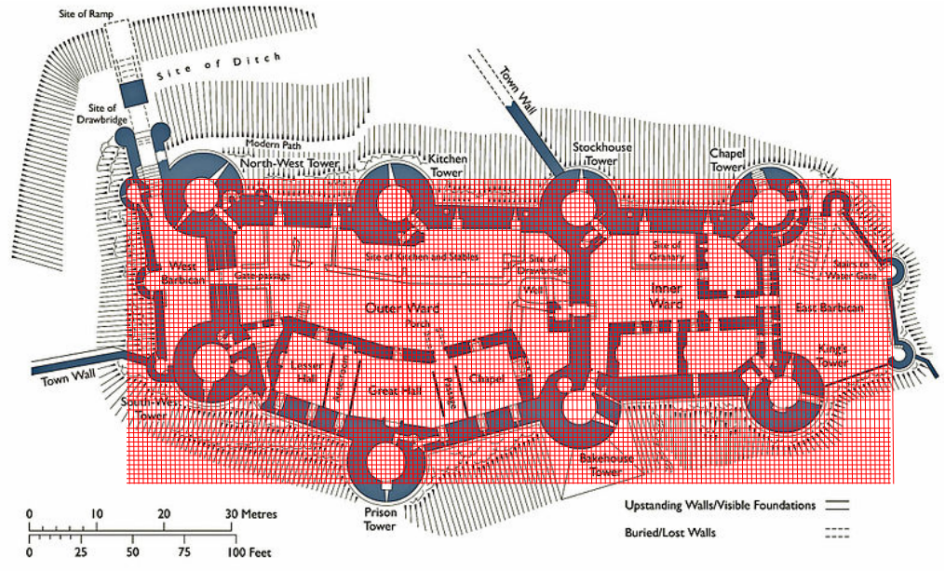
Histogram based localization relies on subdividing the map into small (usually uniformly shaped) regions, typically arranged on a grid. The process estimates belief for each of the grid locations. The resulting grid with associated probabilities is in effect a histogram (hence the name for this method). Let's see how that would work.

Here's a map of Conwy Castle, let's imagine we're implementing a robot that will traverse the castle grounds picking up any garbage left behind by impolite visitors.




(original image: Wikimedia commons, author: Cadw, license: Open government license)

Histogram localization begins by setting up a grid of locations over ***possible places the robot may be***. This is shown below




Notice a few issues right from the start: Typical maps don't align neatly with artificial grids. Our grid contains many cells that correspond to locations where the robot can't physically be. There are also grid cells that are partly occupied by walls or other obstacles. The practical impact of these issues will be some waste of computation (for estimating belief over locations the robot can't possibly be at), and a limit to the accuracy we can achieve from the localization process. We'll get back to this later.

The process for estimating the belief values at each grid cell is iterative, and consists of two steps:

- ***Sensing:*** carrying out a sensor measurement, and using the sensor measurement to update belief

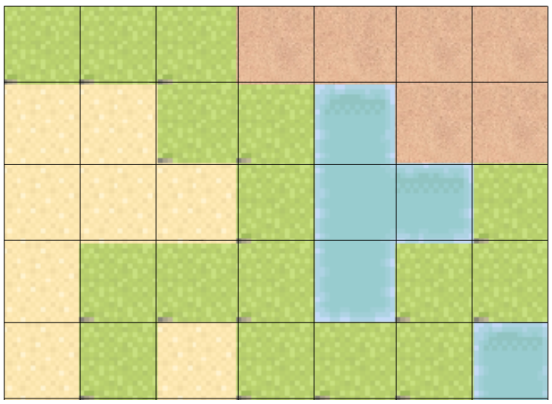
- ***Acting:*** the robot performs an action or actions (consistent with the goals it has to achieve) and we update beliefs based on the result of these actions.

In essence, we're breaking the process of computing ***Bel($x_k$)*** into two parts:

$$Bel(\vec{x_k}) = p(\vec{x_k}|\vec{x}_{k-1}, a_{k-1}, \vec{s_k}) = p(\vec{x_k}|a_{k-1})p(\vec{z_k}|\vec{x_k})$$

the first term accounts for actions, and the second term accounts for sensor readings. Let's see how these two steps eventually lead to localization. We will look at a very simple example of a small grid-world, where the map consists of ***color-coded locations*** (this is not as artificial as you may think, as many robotics localization problems use specially coloured or patterned landmarks) and we can see how the different steps of the process affect belief values.

Consider the small map shown below, where each grid location has an associated colour



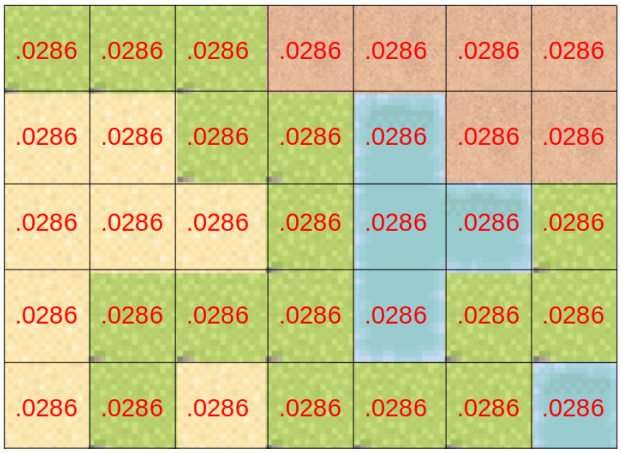
There are 4 different colour tiles in the map above, our robot has a sensor that can report the colour of the tile the robot is currently in, but:

* The sensor is noisy and imperfect – which means that sometimes it reads the wrong colour
  We need to model this – we would normally do this by calibration, that means taking a lot of readings on the map at different locations and keeping statistics on how often the sensor reads each colour correctly, and how often it reads the wrong colour (possibly for each wrong colour). For simplicity, here we will simply assume that the probability the robot reads the correct colour is .85, and the probability it reads the wrong colour is therefore .15

* We don't know where the robot is at the start of the localization process, and it could be anywhere. If we have any information regarding the likelihood the robot may start at specific locations, then we can and should use that.
  Here we will simply assume there's a uniform probability the robot will be at any of the grid locations in the map.

Given the above, our ***initial belief values*** are equal across all the map locations. Since belief is a probabilistic quantity, we must ensure the ***sum of beliefs over the map equals 1.0***.

There are 35 tile locations in the map, therefore the initial belief at each location is 1.0/35 and the initial setup for our localization process is as shown below:




Now recall we have a 2-step localization process. The first step is ***sensing***, so the robot will use its sensor to determine the colour of the tile it's on currently and use that to evaluate

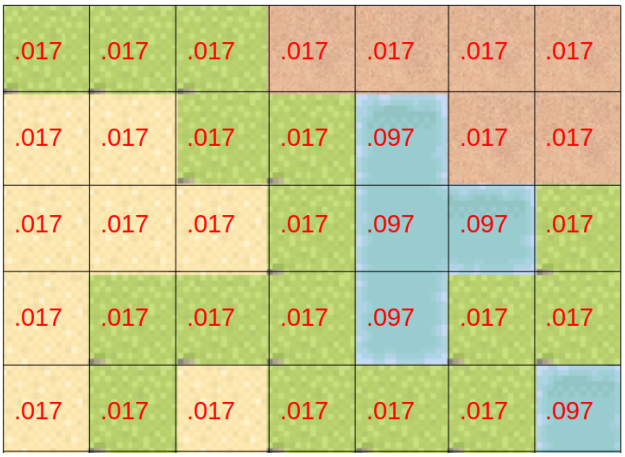
$$p(\vec{z_k}|\vec{x_k})$$

which is the probability that we would observe the curren sensor measurements $\boldsymbol{z_k}$ at each of the possible locations in the map.

In our example, let's assume the robot reads 'blue'. We update the value at each grid location using this new piece of information. For tiles that are blue, we ***multiply the value there by the probability the sensor read the colour correctly (.85)***. For any other colour tile, ***we multiply their value by the probability the robot read the colour erroneously (.15)***.

After updating the values in the map, ***we re-normalize the values so they add up to 1***. This is an essential step that has to be carried out after each sensor/update step.

The resulting values are shown below – ***note that these are not beliefs yet***, as we have yet to account for the action part of the belief update. Notice that the values at locations that are blue are now significantly larger than they were before. Similarly, values for tiles that are not blue have become significantly smaller.



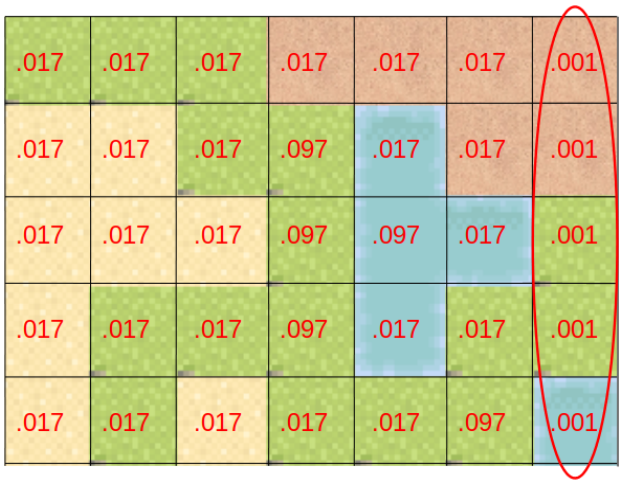

You may be tempted to simply take repeat measurements, which quickly reduces the uncertainty due to possible sensor read errors. However, sensor measurements alone will not allow the robot to determine where it is because of the presence of multiple tiles all with the same colour. The second step in the localization process is essential in determining localization.

The ***acting*** step requires the robot to perform an action that can help the localization process along. Typically this means moving around the map, ***exploring*** its surroundings. The specific action and how it's used for localization depends on the map and what the robot can do. But for the example above, let's assume our robot can move one unit in any of the directions ***up, down, left, or right***. One or more of these actions may not be available if the robot is at the border of the map.

For our example, let's have the robot move ***one unit to the left***. The important question is ***how does this modify the values in grid cells?***. Assuming the robot always carries out the selected action perfectly, then ***wherever the robot currently is, it will be exactly one tile to the left of where it was*** at the previous moment. This means ***values must be displaced one unit to the left***.

In this ideal case the action step simply ***shifts values over the map to reflect the motion of the robot***. This is easy because the robot always moves as expected, we will see in a moment what to do for a more realistic case in which the robot's actions are noisy and imprecise. For now, let's see what happens when the robot moves one unit to the left.

1) The current values are all shifted one unit to the left, those on the leftmost column drop off. One issue here is what to do with the rightmost column. If the robot moves perfectly, there is no chance the robot is anywhere along the rightmost column, so we could fill these cells with zero. However, ***in practice it's useful to use a small non-zero value*** instead, this is simply because in real-world situations, there is always a chance the robot didn't do what we asked. As long as no belief value ever goes to zero, the process can recover from mistakes.




Of course, we need to re-normalize the array after this step. The ***final belief values*** (which now account for both the sensor measurements and the action taken by the robot) are shown below.

| | | | | | | |
|---|---|---|---|---|---|---|
| .019 | .019 | .019 | .019 | .019 | .019 | .001 |
| .019 | .019 | .019 | .106 | .019 | .019 | .001 |
| .019 | .019 | .019 | .106 | .106 | .019 | .001 |
| .019 | .019 | .019 | .106 | .019 | .019 | .001 |
| .019 | .019 | .019 | .019 | .019 | .106 | .001 |

Notice that after just one step of the localization process, we have gone from complete uncertainty about where the robot is to a situation where just a handful of cells have high belief values. The process is then repeated.

In the ***sensing*** step, the robot now reads 'green' and the cell values are updated accordingly depending on whether the cell colour agrees with the sensor measurement or not. After re-normalization, the resulting values are as shown below.

| | | | | | | |
|---|---|---|---|---|---|---|
| .027 | .027 | .027 | .005 | .005 | .005 | .0001 |
| .005 | .005 | .027 | .154 | .005 | .005 | .0001 |
| .005 | .005 | .005 | .154 | .027 | .005 | .001 |
| .005 | .027 | .027 | .154 | .005 | .027 | .001 |
| .005 | .027 | .005 | .027 | .027 | .154 | .0001 |

Now suppose the robot performs the ***action*** 'move up one square', and then the following ***sensing*** step reads 'red'. The resulting values after carrying out the appropriate updates and re-normalizing the array are shown below (notice once more these are not beliefs, as we have yet to carry an action update for the latest step!).

| | | | | | | |
|---|---|---|---|---|---|---|
| .002 | .002 | .014 | .506 | .014 | .014 | .0003 |
| .014 | .014 | .014 | .087 | .014 | .014 | .003 |
| .003 | .016 | .016 | .089 | .003 | .016 | .0008 |
| .003 | .016 | .003 | .003 | .016 | .089 | .0001 |
| .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 |

At this point, there is a single location in the array whose value is singificantly larger than anything else – the localization process has converged and we can claim the robot is at the top row, in the circled location.

Keep in mind that it may take fewer or more rounds of localization to converge to a single location that is much more likely than the rest. This will depend on the map itself and how distinct different locations are, as well as on the accuracy of the sensor, and the ability of the robot to carry out actions precisely. You should keep in mind that ***the localization loop doesn't stop once it selects a location***. As there is always a chance we have made a mistake due to an unfortunate sequence of bad readings and/or particularities of the map, ***the localization process is active as long as the robot is working***. This means that the belief values are always up to date with our most recent sensor measurements and actions, and if we made a mistake earlier on, the process will eventually sort out the correct location.

Also note that the above process does not depend on having a very precise model for sensor noise – as long as the sensor reads values close to the correct ones more often than not, and as long as the sensor update makes the probabilities of cells that agree with sensor values larger by a reasonable amount, and the beliefs for cells that don't agree with the sensor smaller by a reasonable amount, the process will eventually converge.

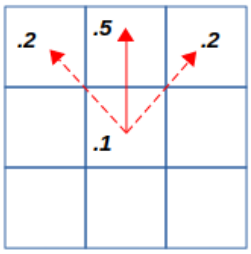
Having an accurate sensor model and updating probabilities in a more realistic fashion will allow the process to converge in fewer steps, and with lower likelihood of choosing the wrong location at some point along the process.

Up to this point, we have assumes that robot actions are perfectly carried out but with real robots, this is never the case. For most situations, we will need to account for uncertainty in the results of the ***action*** in how we update the belief values in the array.

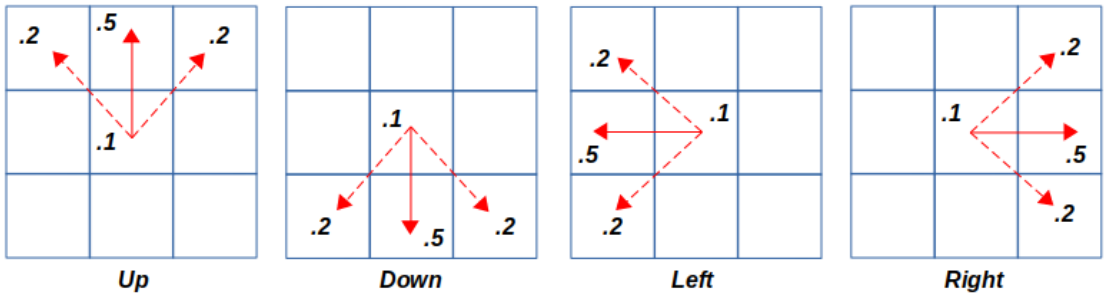
***The Motion Model***

Since the action step typically involved the robot moving around exploring its environment, we need to account for uncertainties in how the robot moves. This is done through a motion model that specifies the probability that having chosen a specific move to make, the robot will end up at various possible locations. Just like the sensor model, we can obtain a motion model by having the robot perform each possible action repeatedly, and collecting data of where it ends up.

To continue our example above, suppose we have collected such information and have determined the motion model for our little robot is as shown below:




The model above applies when the robot's action is to move up by one unit, and it tells us that the robot's motion is imprecise. The expected result (the robot ends up in the square above the current one) only happens half the time. It ends up moving diagonally 40% of the time, and with some probability it actually stays in the same square. Of course the motion model is relative to the direction the robot is facing, so accounting for the four possible directions of motion, we have




The ***action*** update to the beliefs in the map now becomes probabilistic, and is given by:

$$p(\vec{x_k}|a_{k-1}) = \sum_{\vec{x}_{k-1} \in \mathcal{N}_x} p(\vec{x_k}|\vec{x}_{k-1}, a_{k-1}) Bel(\vec{x}_{k-1})$$

this equation states the following idea: The probability that the robot ends up at location $x_k$ (at the start of step ***k*** of the localization process), given that it performed action $a_{k-1}$ (at the previous localization step, ***k-1***) is the result of considering ***all possible locations it could have been at step k-1 that are near x*** (the locations $x_{k-1}$ in the neighbourhood $\mathcal{N}$ of ***x***), the ***probability the robot would travel from each of these locations to x given the motion model*** for the action it carried out, and ***the belief that the robot was at each of these locations in the previous step.***

Let's look at an example on our map from above. Suppose that we have the following belief values for the map just after the last ***sensing*** step, and that the robot's action was to move ***left***:


suppose we want to update belief for the location circled in the map above. There are 4 possible locations that the motion model says could have resulted in the robot being at the circled location. It could have moved from the red cell above and to the right, and moved to the circled location with probability .2; it could have been at the blue cell immediately to the right, and ended up at the circled location with probability .5; it could have been at the green cell below and to the right, and moved to the circled location with probability .2, and it could have already been at the circled location and stayed there with probability .1.

The final belief value for this location is therefore (.2 * .005) + (.5 *.005) + (.2 * .027) + (.1 * .027) = .0116. The corresponding computation has to be carried out for each cell in the map, and once belief values have been updated they have to be re-normalized, as always. You should spend a moment considering what the update looks like for the map above, the same action (move left), and for the locations on the rightmost column of the map.

Bear in mind that the specific motion model you need to use depends on your particular localization problem. But just like with the sensor model, the process should be able to handle an approximate motion model and still converge to the correct location. A better motion model will allow the model to converge in fewer steps and with less lieklihood of picking the wrong location at some point in the process.

***Limitations and Practical Considerations***

Histogram based localization is fairly straightforward, but there are a few limitations and practical considerations that need to be noted.

Firstly, There is a direct tradeoff between the size of the grid cells used in localization (which determines the accuracy of our location estimate), and the computational cost of the process. To see how this may be a problem let's consider a realistic problem: Using histogram localization to determine a car's location within the area of a ciy.

An estimate of the surface area of the GTA gives a total of 7,125 square kilometers. If the GTA was a perfectly square region, this would correspond to a square just under 2.7 x 2.7 Km in size. If we sub-divided this square into cells that are each 1m x 1m, the total number of cells would be 7,290,000. These cells would all need to be updated at each step of the localization process as per the algorithm discussed above.

The process would be computationally quite intensive, and given that a car needs to have accurate location in real-time, we may not be able to handle such a large grid. And this assumes we only need to determine the car's location. In practice we will also need to estimate the car's heading direction, and its speed (these are all relevant state variables required to allow a self-driving car to operate).

Suppose we discretize the heading direction into 16 bins, and we discretize the velocity of the car into 100 bins (e.g. for speeds between 20 and 120 KM/h in 1 Km/h increments). This means 7,290,000 x 100 x 16 bins in total = 11,664,000,000.

That is clearly too many to handle in any practical way.

Just like estimating joint probability distributions for multiple variables, performing histogram localization with multiple state variables for realistic situations becomes impractical very quickly.

Making the bins larger doesn't help much, by the time we have reduced the size of the problem to something manageable, our bins will be too big to provide a good estimate of the state variables we care about.

One practical way to handle this problem is to perform localization ***over multiple scales***. This means: start with a grid that has large cells (e.g. 100m x 100m in the example above), once we have identified a specific cell in this large grid, perform localization ***within that cell only*** using the significantly smaller 1m x 1m cells – and at this point estimate the remaining state variables.

Different sensors and measurements will be needed to perform localization at each scale, and both the sensor and action updates need to be adjusted accordingly. But it is possible in this way to manage the computational cost of the process.

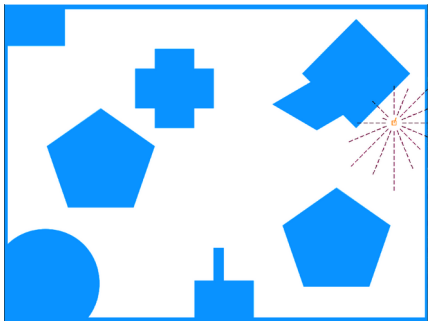
A second potential issue with histogram based methods is that real-world spaces often don't subdivide cleanly into square cells. This is less of a problem as long as we don't have significant portions of the map that are covered by regions the robot can't actually visit – if such regions exist, the update process needs to account for them in some way.

With the above considerations in mind, let's now have a look at an alternate method for localization that avoids some of the limitations of histogram based localization. It is a powerful and often used method for estimating, and keeping track of, state variables for a dynamical system.

***Particle Filters for Localization***

A particle filter is a method for estimating a probability distribution over possible states for a dynamical system. It is an iterative, sampling-based method, applies to a wide range of problems, and scales better in terms of computational cost than a comparable histogram based method.

Let's assume we have a localization problem for which we are attempting to estimate the value of a set of state variables $\boldsymbol{x}$. For the example in this section we will assume the state variables we want to estimate comprise the spatial location of a robot as well as its heading direction. As with histogram-based methods, we require a map, and we assume our robot has access to a sensor or sensors providing information that is useful for localization; and that the robot can carry out actions intended to help it explore the map in order to achieve localization. Consider the setup shown below:



we have a robot somewhere in a room with open areas and walls or obstacles. The robot is characterized by its position and its heading direction (so 3 state variables for this problem). And it uses an ultrasonic sensor to measure the distance of nearby walls at various orientations around it (this is shown as the dashed lines).

Like all sensors, the ultrasound sensor is noisy and the distances returned are not exact. It also has a maximum range, beyond which it can't sense any obstacles.

***Particles***

A particle filter creates a large number of ***virtual robots*** – which we are going to call ***particles***. These are placed all over the map (on locations that are possible valid positions for the bot). Each particle is characterized by the same state variables the particle filter is attempting to estimate. So in this case, each particle will have a position and a heading direction.

Initially these are all random, and the particles are randomly placed over the map.

Each particle will have an associated ***Bel(x)***, which will measure the ***likelihood the particle is actually at the same location, and has the same heading direction as the real robot.*** Much like histogram-based methods, initially the belief values are uniform and the same for all particles.

***Running the Particle Filter***

The particle filter has three steps that are carried out as part of a loop. These steps are

***1) Action***
***2) Sensing***
***3) Resampling***

The first two steps are the same as for histogram-based localization.

***Action Step***

The action step is fairly straightforward, and much simpler than it is for histogram-based localization. First, the robot chooses an action to perform and carries out this action. The action may be random, or may form part of a pre-programmed pattern, but in either case the goal at this point is to explore the environment and accumulate information that will allow the robot to determine its location.

For each particle ***i***, update the state variables ***as they would be affected by the action performed by the robot.*** For instance, suppose the robot's chosen action is ***to move forward by 1 meter***. Then we would look at each particle's current position, and update its location by calculating where it would be it if moved 1 meter forward along its current heading direction.

***Very importantly:*** The action update ***must simulate the noise and imperfections*** in how the robot performs actions. This means that if, for example, the robot's action is moving 1m forward, the actual distance traveled will be close to, but not exactly 1m. Also, the robot will move in a direction that is close to, but not exactly the current heading direction for the particle.

We simulate these imperfections by building a motion model (by calibration), or by using a simple Gaussian noise model. With the Gaussian noise model the process is very simple:

Simulate the action 'move forward by 1m' by:

1) Computing a displacement distance as
$d = 1 + \mathcal{N}(\mu_d, \sigma_d)$
which means adding a value randomly drawn from a Gaussian distribution with carefully chosen mean and standard deviation. Because it's Gaussian, the noise value will more often be close to zero, but there's always a chance of getting larger noise amounts.
2) Compute the heading direction as
$\theta = \theta + \mathcal{N}(\mu_\theta, \sigma_\theta)$
which means adding a random value drawn from a Gaussian distribution to the current heading. The mean and standard deviation are carefully chosen to approximate the behaviour seen in the real robot.
3) Updating the position of particle ***i*** by displacing it by the specified distance ***d*** along direction ***θ***.

For different actions, the process above would be adjusted accordingly.

***Sensing step***

At step of the localization process the robot takes a set of sensor readings ***z***. In the case above, this corresponds to the sonar distances to nearby walls.

These readings will be used to estimate the probability ***that we would obtain the sensor readings in z if the robot was located at the same position as particle i***.

$p(\vec{z}|\vec{x_i})$

Here is how this is done in the context of particle filtering. For each particle ***i,***
a) Estimate the distances from particle ***i*** to walls around it, using the map. This is ***ground truth***.

b) Compare the ***ground truth for particle i*** with the ***sensor readings from the robot***.

c) Compute the probability that any differences are the result of noise in the sensor.

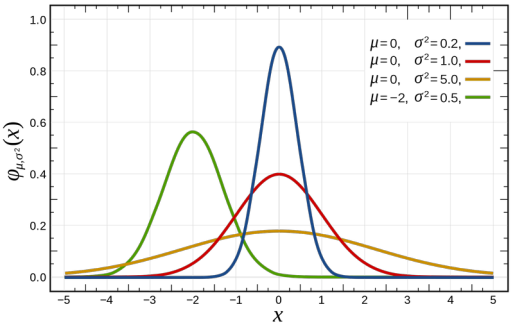
Just as with histogram localization, we need a sensor model to estimate this probability. And just like with histogram localization, we can obtain this model by calibration. If we don't have the chance to perform calibration, there is a general sensor model that works well in many cases.

***Gaussian sensor model***

A common, generally useful model for sensor noise assumes that noise is ***Gaussian with zero mean***. A Gaussian probability distribution is given by the equation

$$\mathcal{N}(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}},$$

where ***μ*** is the mean of the distribution (the location of the center of the curve), and ***σ*** is the standard deviation which controls its width. Do not confuse the ***x*** here with our state variables for particle filtering, this ***x*** is just the horizontal axis of our probability distribution. Curves for different values of the mean and standard deviation are shown below:




(Image: Wikimedia commons, by inductiveload, public domain)

note that the wider the distribution, the 'shorter' it is at its centre. Conversely, the narrower the distribution the 'taller' its peak. The area under the curve must be equal to 1 for this to remain a proper probability density.

With a zero-mean Gaussian, our sensor model states that:

- ***Error values close to zero are more likely,*** this is reasonable since we expect a good sensor to return a value close to the real value.

- ***Error values away from zero become less likely the farther they are***, once more this is reasonable since we don't expect a sensor to return values that are arbitrarily far away from reality.

In addition to the zero-mean Gaussian noise assumption, we often also assume that if multiple sensor readings are available (either from multiple different sensors, or as in the case above, multiple readings from the same ultrasonic sensor but for different directions) these measurements are independent – the amount of noise for each measurement does not depent in any way on the remaining sensor readings.

Putting everything together, we have a collection of ***ground truth (GT) values for particle i***, and a corresponding collection of ***sensor values acquired by the robot***.

For each pair ***j*** of ground-truth and corresponding sensor measurement, we estimate an error value as $\boldsymbol{err_j = GT_j - Sensor_j}$. This is simply the difference between the value we would expect the sensor to read (ground truth) if the robot was exactly at the same location as particle ***i***, and the actual measurement returned by the sensor.

Now we can evaluate the probability that the observed differences are due to noise (as opposed to being the result of particle ***i*** not being at the same location as the robot):

$$p(\vec{z}|\vec{x_i}) = \mathcal{N}(\vec{E}, \mu, \sigma) = \Pi_{j=1}^{w} \frac{1}{\sqrt{2\pi\sigma^2}} e^{(\frac{err_j - \mu)^2}{2\sigma^2})}$$

we evaluate this probability for each particle ***i*** and update the belief for particle ***i*** as

$$Bel(\vec{x_i}) = Bel(\vec{x_i})p(\vec{z}|\vec{x_i})$$

intuitively, we expect that particles whose ground-truth values ***agree reasonably well with the robot's sensor measurements*** will have their belief values reduced by a small amount, while particles whose ground-truth values ***disagree significantly with the robot's sensor measurements*** will have their belief values greatly reduced.

After the sensing update is completed, we need to ***re-normalize beliefs*** to ensure that the sum of beliefs for all particles adds up to 1 and thus remains a valud probability distribution.

Having completed the first two steps of the process, we have an updated set of particles (their state variables have changed) with updated beliefs. Consider for a moment the follwing statement:

If we just repeat steps 1 and 2, and we have a large enough set of particles to begin with, there's some chance that we will discover one or a small number of particles that are close to where the robot is, and after a couple of iterations their belief values will be fairly large, while for the rest of the particles belief values will be small.

However, there is very little chance we will be lucky enough that any of these particles will be close enough to the robot's actual position and heading so as to be actually useful for the robot to carry out its task.

The third step in the particle filter process is intended to produce particles that are better and better at approximating the robot's state variables, whatever those may be.

***Resampling step***

The final step of the partivle filtering loop creates a new set of particles from the current set as follows:

Given the current set of ***N*** particles
Sample a ***new set of N particles*** by randomly choosing particles from the current set with probability proportional to ***Bel($x_i$)***. Given the new set of particles, re-normalize their beliefs so the set remains a proper probability distribution.

Take a moment to consider what this step is doing:

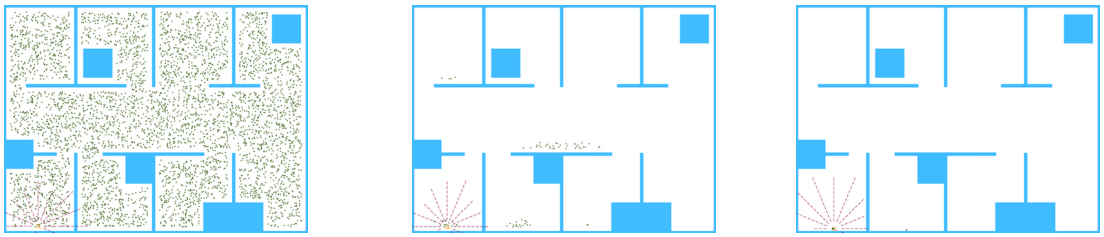
- Particles whose belief values are large, are likely to be choosen multiple times
- Particles whose belief values are very tiny are unlikely to be chosen, but can be on occasion

Overall, the new set will have multiple copies of particles whose belief values are high relative to the rest of the set – the resampling process favours choosing particles whose ground-truth is in better agreement with the robot's sensor measurements. Thus, the new set of particles contains better particles. But, many of these are duplicates. How can this be useful?

Let's go back to the start of the loop – the action step now takes these duplicate particles, and simulates whatever action was taken by the robot. ***Because of noise, each of the originally duplicate particles will end up with slightly different state variables***. They won't be too different from each other, but they will all be different. As a result of this, some of the updated particles are likely to be ***even better matches for the robot's state variables***.

Continuing the process yields sets of particles that progressively get closer to the robot's correct location and heading. Given enough iterations, and assuming the initial set of particles had at least one within a reasonable distance from the initial robot's position, the process will result in a small cloud of variables concentrated around the robot's location, the one with the largest belief should have state variables that are very close matches to the robot's actual state.

The sequence below shows how the set of particles evolves toward convergence over a few iterations of the process



***Limitations and Practical Considerations***

The process depends on having a sufficiently large number of particles in the initial set. However, notice that while histogram-based methods grow in computational burden as a function of the size of the map, or the number of state variables; with particle filters this is not the case. The number of particles is fixed and all that is required is that there is a reasonable chance that a handful of particles will be at a location that is similar to where the robot is. The computational cost of the computation is fixed by the size of the particle set.

Particle filters require some number of iterations to arrive at a good estimate of the robot's state variables, this is because convergence relies on the effect of noise on re-sampled particles in order to

progressively approximate the robot's state. This can take time especially if the initial set of particles is small and the initial particle closest to the robot doesn't have good agreement with the robot itself.

Environments with symmetry and lacking in distinctive features may cause the filter to arrive at an incorrect estimate of the robot's position. Once more the likelihood of this depends on the size of the initial set of particles, as well as the map's characteristics. If the particle set converges to a cloud of similar particles that are all in the wrong place, the particle filter won't recover.

To alleviate this problem, it is common practice to alter the resampling step so that a fixed percentage of the new particles (say 5%, less or more depending on the size of the set and the characteristics of the map) is always drawn uniformly and randomly from all over the map. In this way, if the particle filter converges to the wrong spot, over time there is a chance that one of the randomly sampled particles will be in better agreement with the robot and can re-generate the particle set this time with the correct location.

Finally, it is worth reiterating that the particle filter works well even when we are trying to estimate multiple state variables, and that particle filters are generally applicable to problems other than localization, where we require an estimate of the state of a time-changing system of some kind.

The above are by no means the only or the most powerful robot localization methods. There exists a wide range of approaches both for general localization problems, and for specific tasks/environments. The choice must always consider the characteristics of the problem, the type of environment the robot will be working in, the information provided by the map and the sensors, and the presence of possible obstacles and other moving objects.