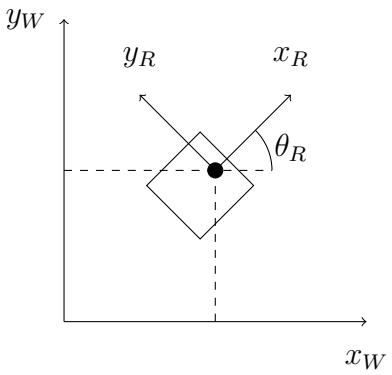
CO333 - Robotics (60019)

Lecture 1 - Introduction to Robotics

Lecture 2 - Robot Motion

A definition of a robot is something that can **move** and **sense**, and uses some sort of information processing to link the two. Robots might want to move in the water, fly in the air, walk (legged) on land, or work in space. The course will focus on wheeled robots that work on generally flat surfaces.

Coordinate Frames



We will be mostly focused on 2D coordinates, on the flat (close to planar) ground. The world frame W is anchored in the world, and the robot frame R is anchored to the robot (consider one point and orientation as the centre of the robot) - consider a set of axis carried by the robot. Often we are interested in the robot's location; the transformation between the world frame W and the robot frame R.

Degrees of Motion Freedom

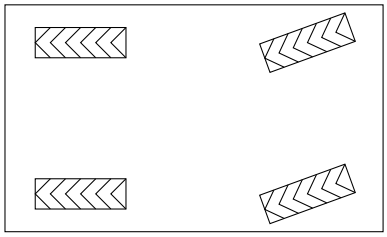
This is to do with how many parameters we need to specify the aforementioned transformation, generally related to the number of dimensions the robot is moving in. The simplest is a single degree of freedom, where a train moves along the x-axis - this position can be specified in one parameter. A rigid body which moves on a ground plane, such as an AV or robot vacuum cleaner has 3 DoF; two translational (x, y) and one rotational (typically θ). On the other hand, a rigid body which moves in 3D space has 6 degrees of freedom; three rotational and three rotational.

A **holonomic robot** is able to move instantaneously in any direction in its space of DoF, otherwise it is **non-holonomic**. Most are non-holonomic, but some holonomic robots do exist; ground-based robots can be made with omnidirectional wheels.

Standard wheel configurations (both non-holonomic; each has two motors but has three degrees of movement freedom, the number of control inputs are lower than the DoF) include;

- **drive and steer** (car) - not implemented in this course

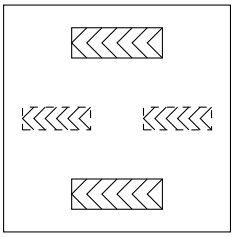
  The combination of acceleration and braking determines how fast it moves forwards, and the orientation of wheel determines direction.



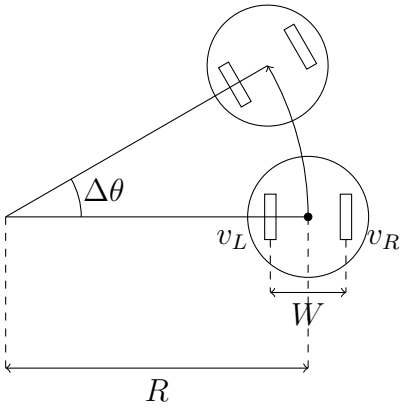
Rear wheels need a differential, variable (Ackerman) linkage for steering wheels.

- **differential drive** (robot vacuum)

  Has two driving wheels, both pointing forwards, and maybe castors which keep it balanced. Robot moves in different ways depending on the different speeds the wheels are turning at.



The caster wheels (dashed) are passive, and simply support the robot. The active wheels have one motor each. If the active wheels are running at equal speeds, the robot moves in a straight line, and wheels running at equal and opposite speeds turn on the spot. On the other hand, in general, other combinations lead to motion in circular arcs / curves.



Consider the left wheel at speed v_L and the right wheel at v_R (linear velocities over th ground, hence $v_L = r_L\omega_L$, where r_L is the radius of the wheel and ω_L is the angular velocity). We also assume no slipping (the intersection is the centre of rotation).

We want to determine R, which is the radius of the circle formed by the robot's centre moving. Consider a small period of time Δt, and an angle of movement $\Delta\theta$;

$$\Delta\theta = \frac{v_L\Delta t}{R - \frac{W}{2}} = \frac{v_R\Delta t}{R + \frac{W}{2}} \Rightarrow v_L\left(R + \frac{W}{2}\right) = v_R\left(R - \frac{W}{2}\right) \Rightarrow \frac{W}{2}(v_L + v_R) = R(v_R - v_L)$$

By rearranging the above, and substituting back in, we have the following;

$$R = \frac{W(v_R + v_L)}{2(v_R - v_L)}$$
$$\Delta\theta = \frac{(v_R - v_L)\Delta t}{W}$$

Actuation (DC Motors)

DC motors are controlled by a power signal, using **PWM (pulse width modulation)** and a fixed voltage. A gearing system is typically used to **gear down** the end effector to be slower with higher torque (compared to the DC motor's rapid rotation, but low torque). Many motors have a built in encoders (connected to the motor), which can be read to measure angular position by counting steps. This is required as a running motor's rotations depend on many conditions (including the load it's driving); for example a heavier robot moving on rough ground will move slower. To measure the rate the motor is moving at, we can feed the rotation back. We can then use feedback control (servo control) to adjust the motor to make it do what we want. In principle, we want to determine where the motor is, and where it actually is. At a high rate, we want to send pulses to reduce the error between the target location and actual location.

PID (Proportional / Integral / Differential) Control

The PID expression sets the power as a function of error;

$$P(t) = k_p e(t) + k_i \int_0^t e(\tau)\,\mathrm{d}\tau + k_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

In this, we have the following terms;

- $e(t)$ — demand minus position (error)

  For example, at each time step this is the angle we want the motor to be at minus the actual angle of the motor measured.

- k_p — main term (high values give rapid response)

  Continuing with the example above, $k_p e(t)$ will be higher if k_p is large. If there's a large error, send a large signal, and vice versa for small errors.

- k_i — integral term (increased to reduce steady state error)

  Imagine the motor is close to where it should be; this allows for the 'gap' to be closed.

- k_d — differential term (reduced settling time / oscillation)

Wheel Rotation Speed to Velocity

Consider a wheel of radius r_w rotating at ω (in radians per second). The speed of the wheel, in theory, would be $v = r_w\omega$. However, in practice, there are factors such as uneven ground and tyre softness, the radius may be hard to measure accurately. Another consideration would be possible slipping; in general it will be better to **calibrate** this for the surface by considering some constant of proportionality.

State in 2D

If a robot is moving on a plane, we can define the robot with a state vector $\boldsymbol{x}$, with three parameters;

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \begin{array}{l} x \text{ component of robot centre point in world frame} \\ y \text{ component of robot centre point in world frame} \\ \text{rotation angle between coordinate frames (angle between } x_W \text{ and } x_R \text{ axes)} \end{array}$$

The two frames coincide when the robot is at the origin ($x = y = \theta = 0$). Note that $-\pi < \theta \leq \pi$. During a straight line period of motion of distance D, we have;

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + D\cos\theta \\ y + D\sin\theta \\ \theta \end{bmatrix}$$

On the other hand, during a pure rotation of angle α we have (rotation to the left is positive by convention);

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + \alpha \end{bmatrix}$$

Lecture 3 - Sensors (Behaviours)

Motivation

Recall the second practical; there was a gradual drift from the desired path. There are aspects of the world that the robot can never fully understand (such as the bumpiness of the floor) - even simulations cannot do exactly the same thing every time. After calibration, the robot should return to the desired location (on average), but some scatter will remain due to factors we cannot control.

Systematic errors are removed, however we still have **zero mean errors** - these errors occur incrementally (where every movement or rotation adds a small amount of potential error). The zero mean errors can be modelled probabilistically, often with a Gaussian distribution.

Uncertainty in Motion

A better model (compared to the previous lecture) adds uncertain perturbations / motion noise;

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + (D + e)\cos\theta \\ y + (D + e)\sin\theta \\ \theta + f \end{bmatrix}$$

We will likely also notice that rotations are not exact either;

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + \alpha + g \end{bmatrix}$$

Sensors

Proprioceptive are self-sensing (such as motor encoders / internal force sensors) and improve a robot's sense of its own internal state. In general, these measurements may depend on the previous states as (as well as the current state); for example wheel odometry gives a reading depending on a difference between the current and previous state, on the other hand, a gyro in an inertial measurement unit will report a reading depending on the current **rate** of rotation (not an instantaneous state).

In this case, the value of the measurement $\boldsymbol{z}_p$ should be a function of the state of the state of the robot;

$$\boldsymbol{z}_p = \boldsymbol{z}_p(\boldsymbol{x})$$

On the other hand, exteroceptive are outward-looking. These allow the robot to generally be **aware** of its environment, letting it localise with respect to a map, recognise objects / locations, map out free space, avoid obstacles, and interact with objects.

In this case, we need both the sate of the robot $\boldsymbol{x}$ and the state of the world $\boldsymbol{y}$ (relative to the robot);

$$\boldsymbol{z}_o = \boldsymbol{z}_o(\boldsymbol{x}, \boldsymbol{y})$$

The state of the world may be parameterised (such as a list of geometric coordinates describing landmarks). This state may be uncertain.

Some sensors (such as touch, light, and sonar) will return a **single value** in a range. On the other hand, some sensors such as a camera or laser range-finder will return an **array** of values, with the former having an array of sensing elements (pixels of a camera's CCD chip), or the latter performing scanning.

Some common sensors are as follows;

- **touch sensor**

  This is a binary on / off state; an open switch will have no current flow, and a closed switch will have current flow (hit).

- **light sensor**

  This detects intensity of passive light incident from a single forward direction (some range of angle sensitivity); this is a continuous value. Multiple sensors can guide steering behaviours (such as driving towards a brighter light). In *Lego*, there is also a mode where the sensor can emit light, where reflections from close surfaces can be measured.

- **sonar (ultrasonic) sensors**

  Sonar measures distance by emitting ultrasonic pulses (tiny pulses of sound at a high frequency) - these beams typically have an angular width of $10° - 20°$. The sensors measure the time for a pulse to bounce back (which the sensor uses to calculate the distance). These are fairly accurate within a few centimetres in one direction (for simple shapes) - but can be noisy in the presence of complicated shapes. The maximum range is a few metres.

- **laser range-finder**

  The principle of this is similar to sonar, however it uses infrared laser beams instead of sound. This is also reflected, detected, and timed to calculate a distance. These are very accurate (sub-millimetre), and work on most surfaces. They can normally scan in a 2D plane (rotating on one axis), but can also scan in 3D. Previously, these were bulky and expensive, however nowadays they are present in modern smartphones.

Vision

This is the generalisation of a light sensor. A camera can be thought of as a large array of light sensors (or a light sensor is a single pixel of) which returns a large, rectangular array of measurements. A single camera measures light intensity (rather than direct information about geometry).

Bump Detection (Touch Sensor)

We can either perform this with a touch sensor, or by thresholding sonar (such as if the distance falls below some range).

Multiple touch sensors mounted inside a floating skirt can allow for detection of where an obstacle was hit. For example, if a robot was hit at the front, it may attempt to drive around it by going backwards, moving around, and continuing. Another simple strategy is to rotate through a random angle and go forward until the next collision (random bounce).

Servoing

Servoing is a control technique where control parameters (e.g. the desired speed of a motor) are coupled directly to a sensor reading and updated regularly in a **negative feedback loop** - this is also known as **closed loop control**. This requires high frequency updates of both the sensor and the update, otherwise the motion can oscillate.

Propeortional control sets the demand proportional to negative error - for example in velocity (where we look at the difference between the desired sensor value and actual sensor value);

$$v = -k_p(z_{\text{desired}} - z_{\text{actual}})$$

This proportional case is a specific case of PID control.

A simple steering law to guide the robot to collide with the target would be as follows (for example - if the obstacle is straight ahead ($\alpha = 0$), then don't steer);

$$s = k_p\alpha$$

On the other hand, to guide the robot to avoid the obstacle at a safe distance (R) would subtract an offset (D is the distance from the robot to the object, and α is the angle between the robot's rotation and the direction to the object);

$$s = k_p(\alpha - \sin^{-1}\frac{R}{D})$$

Wall Following

Consider the following scenario, where a differential drive robot desires to travel along a wall, at some distance d. This uses a **sideways-looking sonar** to measure some distance z. In the following diagram, the wall can be curved; I just don't know how to do that in TikZ.

Note that we are moving left to right (along the diagram), and we want to steer towards the wall (since we'd ideally have $d = v$). To do this, the left wheel must turn slower than the right wheel. Note that we can also achieve symmetric behaviour with a constant offset v_C (standard speed for a straight line). We can then set the left and right wheel velocities as follows;

$$\begin{aligned} v_R - v_L &= k_p(z - d) \\ v_R &= v_C + \frac{1}{2}k_p(z - d) \\ v_L &= v_C - \frac{1}{2}k_p(z - d) \end{aligned}$$

Combining

More complex robots may have multiple servos, and we can combine sensing-action loops. We can instead consider each local servo-like sensing-action loop as a **behaviour**. These behaviours can then be combined in an **arbiter**, which feeds the instructions into a motor controller.

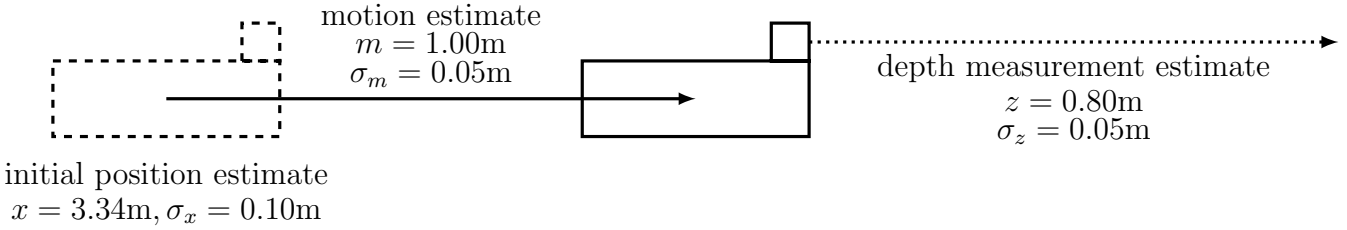
To go further, we need to go beyond a coupling between sensors and actions, and instead build a model of the world. Here we plan a sequence of actions to achieve a goal, and then execute the plan. However, if the world changes during the execution, we need to re-plan the actions.

Lecture 4 - Probabilistic Robotics

From the previous lab, we notice an issue when the robot is at a large angle to the wall; we realistically want the perpendicular distance between the robot and the wall, but this gets much larger if the robot is at a large angle. One solution is to mount the camera in front of the robot, as this couples rotation with distance.

Probabilistic Localisation

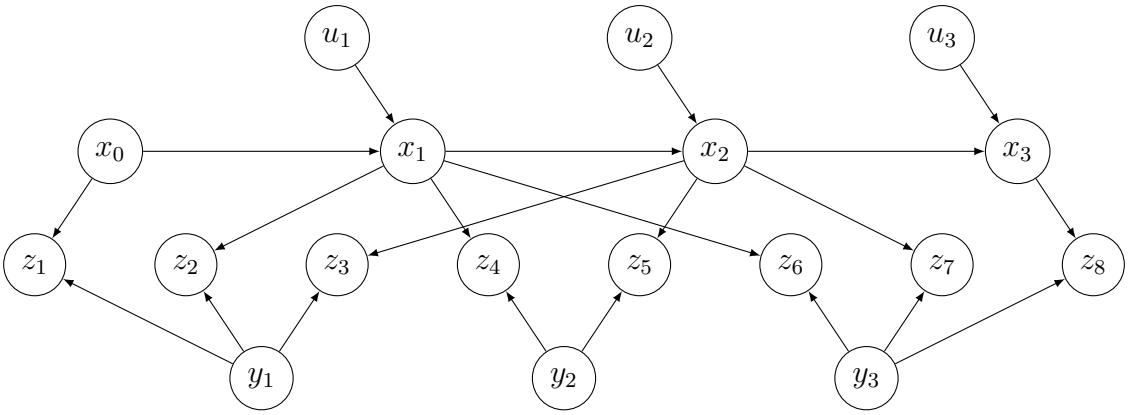
Notice from the previous labs that attempting to estimate the location of the robot can lead to errors adding up. We assume some knowledge of the scene, but we do not know the location of the robot relative to the map. Long-term estimation based on logical reasoning fail when presented with real data; advanced sensors cannot be easily analysed like bump and light sensors, and all information received is uncertain (every action as well as every sensor measurement). A probabilistic robot acknowledges the uncertainty (and uses models to abstract useful information).



When we start with an uncertain position, and make an uncertain motion, the robot will have an uncertain state measurement. Once the robot then performs some measurement, there will likely be some uncertainty in the result.

Probabilistic Inference

We think of what we want to estimate, and then their probabilistic relation to things we can actually observe. The following graph is one visualisation of a SLAM problem;



Note that in the above diagram, the nodes x_i denote the estimate of the location of the robot at some point in time. We are also interested in estimating the locations of landmarks in the world (denoted by y_i - fixed landmarks). The sensors (such as camera or sonar) will make some sort of measurement relative to the position robot (z_i) - for example, at x_1 the robot observes y_1, y_2, y_3 with measurements x_2, z_4, z_6 respectively. Finally, the control inputs (for the movement) are denoted by u_i.

The process of **sensor fusion** combines data from many different sources to create useful estimates. This state estimate can be used to decide the next action.

Probabilities describe our state of knowledge (not randomness, which is rare in the real world). For example, the physics for a coin flip are deterministic, however without this knowledge, we can only guess the outcome.

$$P(XZ) = P(Z \mid X)P(X) = P(X \mid Z)P(X) \Rightarrow \underbrace{P(X \mid Z)}_{(1)} = \frac{\overbrace{P(Z \mid X)}^{(2)}\overbrace{P(X)}^{(3)}}{\underbrace{P(Z)}_{(4)}}$$

(1) posterior

(2) likelihood

(3) prior

(4) marginal likelihood

In the state diagram, we have things we are interested in estimating (state variables) and things we can get from sensors (measurements / observations). Typically state variables are represented with X in the probabilities, with measurements being represented by Z. Bayes' rule is used to incrementally digest new information; we have a prior estimate of X already - $P(X)$. From this, we want to get the

probability of X given Z - $P(X \mid Z)$. We want to obtain $P(Z \mid X)$, which is the likelihood of the measurement given the state variable.

Consider the following example;

kitchen	bathroom	living room
$P(K) = 0.3$	$P(B) = 0.2$	$P(L) = 0.5$

The robot starts completely lost, let the probability of it being in each room be proportional to the area (note that the probabilities are normalised). Assume we also have some light sensor (which gives a thresholded value of it either being light or dark), which has had experiments performed in each of the rooms to obtain probabilities. Assume the sensor also now reports it as being bright (hence we want to find the probability of it being in a room, **given** that it is bright).

$$
\begin{aligned}
P(b \mid K) &= 0.8 && \text{bright 80\% of the time in the kitchen} \\
P(K \mid b) &= \frac{P(K)P(b \mid K)}{P(b)} \\
&= \frac{0.3 \times 0.8}{P(b)} \\
&= \frac{0.24}{P(b)} \\
&= 0.4 \\
P(b \mid B) &= 0.3 && \text{bright 30\% of the time in the bathroom} \\
P(B \mid b) &= \frac{P(B)P(b \mid B)}{P(b)} \\
&= \frac{0.2 \times 0.3}{P(b)} \\
&= \frac{0.06}{P(b)} \\
&= 0.1 \\
P(b \mid L) &= 0.6 && \text{bright 60\% of the time in the living room} \\
P(L \mid b) &= \frac{P(L)P(b \mid L)}{P(b)} \\
&= \frac{0.5 \times 0.6}{P(b)} \\
&= \frac{0.3}{P(b)} \\
&= 0.5
\end{aligned}
$$

However, note that the sum of the posteriors must also be equal to 1, hence;

$$P(K \mid b) + P(B \mid b) + P(L \mid b) = 1 \Rightarrow \frac{0.24 + 0.06 + 0.3}{P(b)} = 1 \Rightarrow P(b) = 0.6$$

Note that in the example above, we've digested information that suggests the robot being in the kitchen (due to the brightness), hence the probability of it being in the kitchen has gone up, whereas the probability of it being located in the bathroom has gone down.

Probability Distribution

Discrete inference generalises to more possible states as the bins get smaller and smaller (for example, if we were to divide the floorplan into more zones). As the width of the bins tend to 0, we have a continuous probability density function $p(x)$ at the limit. The probability that a continuous parameter lies in the range a to b is;

$$P_{a \to b} \int_a^b p(x)\,\mathrm{d}x$$

However, as the resolution becomes higher, it is becomes more expensive in terms of memory and computation.

Gaussian distributions often represent the uncertainty in sensor measurements well, parametising a 1D distribution in two parameters (μ and σ);

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

A Gaussian prior multiplied by a likelihood will produce a posterior tighter than both. The product of two Gaussians is **always** another Gaussian. However, there is only one peak, which may not always fit.

Particle Filtering

Another representation is to use particles. A probability distribution here is represented with a finite set of weighted samples (typically $N = 100$) of the state;

$$\{\boldsymbol{x_i}, w_i\} = \left\{ \begin{bmatrix} x_i \\ y_i \\ \theta_i \end{bmatrix}, w_i \right\} \text{ where } \sum_{i=1}^{N} w_i = 1$$

Instead of storing a large table of numbers, we almost entirely ignore where the probability is low (close to 0), and focus on the areas where there is higher probability. We want to store the probability distribution within the degrees of freedom of the robot. The idea here is to test the hypothesis of where the robot is (the cloud of points) and narrow it down based on sensor observations - a good estimate is a tight clump of particles.

In general, these steps are repeated every time the robot moves and makes measurements;

1. motion prediction based on **proprioceptive** sensors

   With blind motion, the particles should spread out more and more (due to a growing uncertainty). For this, we use the state changes with the zero mean noise terms (e, f, g) - these are generated randomly for **each** particle.

   Also note that the robot may move through variable step sizes, and we will need to scale the variance accordingly.

2. measurement update based on outward-looking sensors
3. normalisation
4. resampling

   Redistributes positions of particles - particles will spread out under motion, but will acquire different weights based on how well they agree with the sensor measurements, according to Bayes' rule (higher weights for more probable locations).

At any point in time, our uncertain estimate of the location of the robot is represented by the whole particle set; we can make a point estimate of the current state by taking the (weighted) mean of the particles;

$$\bar{\boldsymbol{x}} = \sum_{i=1}^{N} w_i \boldsymbol{x_i}$$

This state can be used to plan waypoint navigation.

Position-Based Path Planning

Assume the current state of the robot is (x, y, θ), and we want to travel to a waypoint set at (W_x, W_y). We need to first rotate towards the waypoint, with the following direction vector;

$$\begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} W_x - x \\ W_y - y \end{bmatrix}$$

This can be used to obtain an **absolute** angular orientation α, which the robot must drive in;

$$\alpha = \tan^{-1} \frac{d_y}{d_x}$$

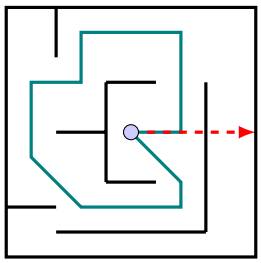
We need to make sure that α is in the correct quadrant of $(-\pi, pi]$, as a standard $\tan^{-1}$ function would give a value in the range $(-\frac{\pi}{2}, \frac{\pi}{2}]$. This can also be achieved with `atan2(dy, dx)` in many languages.

Using this, we can obtain the angle the robot must rotate through as β (and we should take care to shift this angle into the range $(-\pi, \pi]$ for efficient movement). The robot should then drive through a distance d in a straight line;

$$\beta = \alpha - \theta$$
$$d = \sqrt{d_x^2 + d_y^2}$$

Lecture 5 - Monte Carlo Localisation

In this practical, we aim to perform localisation within a map (which is known by the robot in advance).



MCL uses a cloud of weighted particles to represent the uncertain position, and can be thought of in two ways;

- a Bayesian probabilistic filter
- similar to a genetic algorithm (survival of the fittest)

  Each particle is a hypothesis, and we can test whether this is sensible or not by using the new measurement from the sonar. The particles are then assigned a fitness, either increased if they agree with the measurement, and decreased if not. The fitter particles will make more copies of themselves in the resampling stage (leading to the unfit ones dying out).

Note that with the weighted particle distribution, we can interpret it as a probability that the robot is within any multi-dimensional region of the state space being the sum of the weights within that region. However, this could break if we select a very small region between particles - but this is a trade-off we make for performance.

MCL targets both of the following problems;

- **continuous localisation**

  This is a tracking problem, where we want to estimate the robot's new position given the an estimate of its location in the last time-step. If we assume that we start at a perfectly known location, we can set the state of all particles to be the exact same. The weights would all be equal to $\frac{1}{N}$, and we have a spike representing the robot's location.

- **global localisation**

  In contrast, we only know the robot is within a certain region (somewhere within a room). Particles can be initialised to be random positions and orientations within the room, the weights are also all equal again. This is difficult with a single sensor

Our estimate of the robot's location is represented by th particle set, and we can get a point estimate of the current state by taking the mean of all the particles (if the weights weren't equal, it would be more biased towards the higher weights);

$$\bar{\boldsymbol{x}} = \sum_{i=1}^{N} w_i \boldsymbol{x_i}$$

Measurement Updates

An update consists of applying Bayes Rule to each particle. When we have a measurement z, the weights are updated as follows (the denominator in Bayes' rule is a constant that will be removed by normalisation, hence we ignore it);

$$w_{i(new)} = P(z \mid \boldsymbol{x_i}) \cdot w_i$$

The likelihood of particle i is the probability of getting the measurement z given that $\boldsymbol{x_i}$ represents the true state.

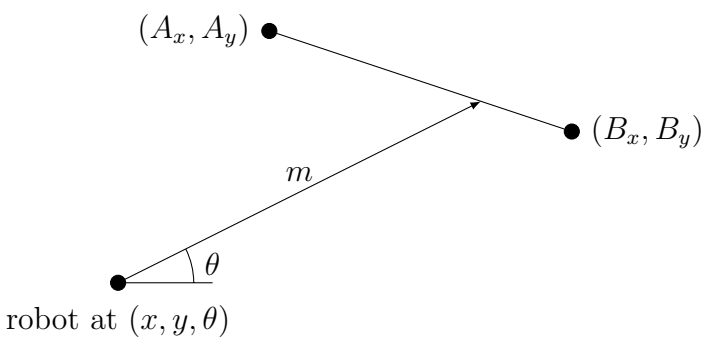
Likelihood Functions

A likelihood function fully describes the sensor's performance. This function has two inputs; the measurement we've achieved (z) and the state variables (ground truth) v. If there is no systematic error, and the uncertainty is constant, we would have multiple Gaussian distributions (assuming the sensor can be modelled as such) along the line $z = v$. This is slightly difficult to visualise, but consider a Gaussian hill (multiple distributions). If we took one slice (a specific v) value, we'd have a distribution with the mean at v.

Different sensors will have different characteristics. A sonar typically doesn't have much variation no matter the distance (when plotted, the shape stays the same and only the mean changes), whereas other sensors such as stereo vision camera system will have less accuracy the further away it is (hence the Gaussian becomes flatter as v increases, representing larger spread).

Obtaining Ground Truth

We can obtain the ground truth as follows;



Note that m is the forward distance (at angle θ) to an **infinite wall** passing through the points $\boldsymbol{A}$ and $\boldsymbol{B}$;

$$m = \frac{(B_y - A_y)(A_x - x) - (B_x - A_x)(A_y - y)}{(B_y - A_y)\cos\theta - (B_x - A_x)\sin\theta}$$

For each wall, we work out the value m. One criteria is that the m must be positive (since the sonar would give a positive measurement). Furthermore, we also need to check if the point it intersects is actually between the endpoints of the wall (since the walls are considered to be infinite). Additionally, if several points are valid, the closest one is the one it is actually responding to. The point on the wall can be calculated as;

$$\begin{bmatrix} x + m\cos(\theta) \\ y + m\sin(\theta) \end{bmatrix}$$

Likelihood

The likelihood should depend on the difference $z - m$; if this is small then it validates a particle (and weakens the particle if large). A Gaussian function is usually a good model for the mathematical form for this, with the standard deviation σ_s being based on our model of how uncertain the sensor is (and may either be constant or depend on z).

$$p(z \mid m) \propto e^{-\frac{(z-m)^2}{2\sigma_s^2}} + K$$

We may consider modifying the Gaussian distribution to make the likelihood function more **robust**. This is more applicable in real-world robotics (which may give garbage data, possibly due to poor electronics etc.). A Gaussian normally states that there is almost no chance of it getting a value very far from the mean, however by **adding a constant**, it creates a **heavy tail**, which gives a constant probability the sensor may report garbage data, uniformly distributed across the range of the sensor.

In MCL, this causes the filter to be less aggressive in removing particles which are far from agreeing with a measurement. This prevents a garbage measurement killing off particles in good positions.

Another factor we may want to consider is the angle between the sonar direction and the normal to the wall, β. If this is too large, the sonar reading may not be sensible.

$$\beta = \cos^{-1}\left(\frac{\cos\theta(A_y - B_y) + \sin\theta(B_x - A_x)}{\sqrt{(A_y - B_y)^2 + (B_x - A_x)^2}}\right)$$

Resampling

All the weights should now be scaled to add up to 1;

$$w_{i(new)} = \frac{w_i}{\sum_{i=1}^{N} w_i}$$

Resampling involves generating the new set of N particles, each having equal weights, but with a spatial distribution reflecting the probability density. To do this, we copy the state of one of the previous particles with the probability according to the particle's weight. This can be achieved by generating the cumulative probability distribution of the particles and generating a random number (between 0 and 1 if normalised), and picking the particle that this random number intersects. It's possible to skip the normalisation step and resample from an unnormalised distribution in a similar way (for efficiency).

Compass Sensor

This process is quite general, and can also be done with other sensors. Consider a compass that measures the bearing β relative to magnetic north. The likelihood $P(\beta \mid \boldsymbol{x_i})$ only depends on the θ component of $\boldsymbol{x_i}$. If the compass is working perfectly, then we have the following (where γ denotes the magnetic bearing of the x coordinate axis of frame W);

$$\beta = \gamma - \theta$$

The likelihood then depends on the difference between β and $\gamma - \theta$;

$$e^{-\frac{(\beta-(\gamma-\theta))^2}{2\sigma_c^2}}$$

Lecture 6 - Advanced Sonar Sensing

Global Localisation

MCL can be used to attempt global localisation by initialising particles within a room (and random orientation for each particle). However, this is computationally expensive due to the number of particles, and requires a number of measurements before a location is obtained. A ring of sensors may improve this.

After a single depth measurement, for example 20cm, only the ones (relatively) closer to the boundaries of the room would be kept - but this may still be ambiguous. With more measurements, we narrow down the robot's position further and further. Ambiguity can also be reduced with extra measurements from other sensors, such as with a compass.

Recognition

This is for when the robot can be in one of N possible positions. In a prior training phase, the robot is manually placed in each of these locations, and rotated to take a spaced set of sonar measurements (such as one per degree). This creates a **signature / place descriptor** for the region (which would could be a plot of robot angle (x axis) against depth measurement on the y axis, essentially just a table of numbers).

If the environment is varied, such as with different shapes or obstacles, the signature will be different for each place. At run-time, when the robot is actually lost (in one of these random locations), the robot takes a new signature and checks if any of the existing descriptors are similar.

Two histograms can be compared with the sum of squared differences D_k, note that $H_m(i)$ is the new measurement, and $H_k(i)$ is the saved signature;

$$D_k = \sum_i = (H_m(i) - H_k(i))^2$$

The location with the lowest D_k is the most likely, however this should be checked against a threshold in the case the robot isn't in one of these locations.

If the test histogram and the saved histogram can be brought into agreement with a shift, the robot is likely to be in the same place but rotated - at test time, all shifts are tested (the amount of shift to get the best agreement is the measurement of the rotation).

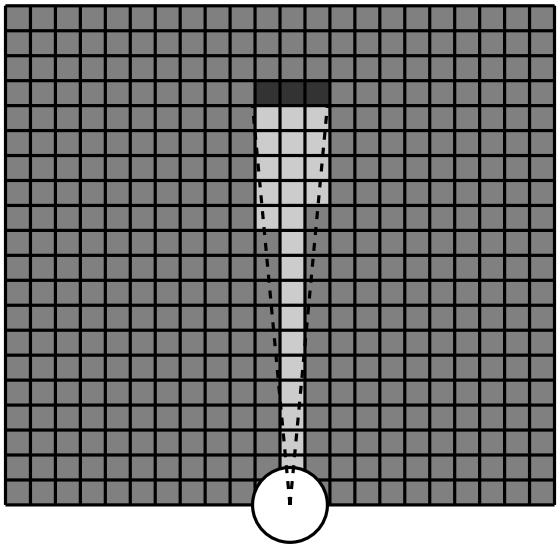
However, this is quite computationally expensive. An alternative is to use a **depth histogram**, which has depth on the x axis and frequency on the y axis - note that this is now **rotation invariant** and can be tested in one check (rather than for each rotation step). Each of the depths are binned (for example in 10cm increments) and counted for frequency. Once the location is found, the shifting procedure can be performed for a single location to determine the robot's orientation.

This could be trained as a neural network, with the input being a test signature and the output being x, y coordinates representing the location of the robot. The neural network could attempt to interpolate between the known location signatures.

Probabilistic Occupancy Grid Mapping

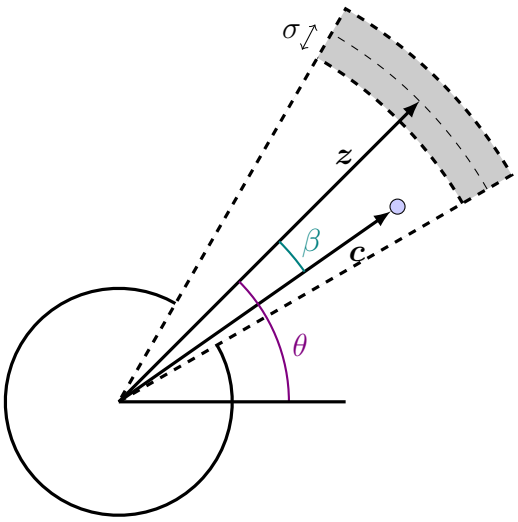
In contrast to before, we know exactly where the robot is, however we do not know what the scene is.

We define an area on the ground in which we would like to map, and choose a resolution for a grid. For each cell i, we store and update the probability of occupancy - $P(O_i)$. Of $P(O_i) = 1$, we are certain that the cell is occupied by an obstacle (normally represented by a black cell). Similarly, if $P(E_i) = 1$ (white cell), we are certain the cell is empty (note that $P(O_i) + P(E_i) = 1$) - the intermediate probabilities represent uncertain knowledge. By default, all the probabilities of occupancy are set to some constant prior value (such as 0.5) - however, it may be reasonable to initialise this at a lower value, since it may be more likely that the room has more empty space than not. Our map is of a high quality when the majority of the cells are close to 1 or 0.



Note that for sonar in reality, we have a narrow cone, rather than a single beam. Suppose a sonar measurement Z gives a depth of d (bouncing off the closest object). This not only tells us that there is likely an obstacle at a distance of d in front of the robot (thus increasing the probabilities), but the cells in front of the robot which are less than d away are also likely to be empty (thus we should decrease the probabilities). This measurement doesn't give any information about the other cells, which aren't in the beam. We must test if each cell lies in the beam.

Consider the following diagram; note that the size of vector $\boldsymbol{z}$ is d. The vector $\boldsymbol{c}$ is distance from a robot at (x, y, θ) to a single cell $\boldsymbol{g}$.



We can obtain the following vectors, and use them as follows;

$$\boldsymbol{z} = \begin{bmatrix} d\cos\theta \\ d\sin\theta \end{bmatrix}$$

$$\boldsymbol{c} = \begin{bmatrix} g_x - x \\ g_y - y \end{bmatrix}$$
$$\beta = \cos^{-1} \frac{\boldsymbol{z} \cdot \boldsymbol{c}}{|\boldsymbol{z}| \cdot |\boldsymbol{c}|}$$

Note that we want to check if β is less than the half-angle of the beam (such that it's actually in the beam). If the following holds, then the cell is in the grey region (endpoint of the beam);

$$||\boldsymbol{z}| - |\boldsymbol{c}|| < \sigma$$

On the other hand, if the following holds, then the cell is in the white region;

$$|\boldsymbol{c}| < |\boldsymbol{z}| - \sigma$$

For each cell, we can apply Bayes rule to get a posterior probability for each cell. Note that we are assuming independence, which is an approximation as this isn't necessarily true in reality.

$$P(O_i \mid Z) = \frac{P(Z \mid O_i)P(O_i)}{P(Z)}$$

Note that since $P(O_i \mid Z) + P(E_i \mid Z) = 1$, we can avoid calculating $P(Z)$ by calculating $P(E_i \mid Z)$;

$$P(E_i \mid Z) = \frac{P(Z \mid E_i)P(E_i)}{P(Z)}$$

We can divide the two equations as follows, to obtain a ratio;

$$\left(\frac{P(O_i \mid Z)}{P(E_i \mid Z)}\right) = \left(\frac{P(Z \mid O_i)}{P(Z \mid E_i)}\right)\left(\frac{P(O_i)}{P(E_i)}\right)$$

Note that we can use the odds notation (since $E_i = \bar{O}_i$);

$$o(A) = \frac{P(A)}{P(\bar{A})}$$

For example, if $P(O_i)$ is high, then $P(E_i)$ must be low, hence the odds are very high (close to positive infinity for a high probability), and vice versa (will be very close to zero). This can be used as follows (and logs can be taken);

$$o(O_i \mid Z) = \left(\frac{P(Z \mid O_i)}{P(Z \mid E_i)}\right) o(O_i) \Rightarrow \ln o(O_i \mid Z) = \ln\left(\frac{P(Z \mid O_i)}{P(Z \mid E_i)}\right) + \ln o(O_i)$$

We typically store the log of odds allowing for updates to be performed in an additive way. Note that probability 0.5 will have log odds of 0, a positive log odd means a probability > 0.5 (and similarly < 0.5 for negative log odds) - these are typically capped at certain limits.

Note that the update term is defined as follows, for each cell;

$$U = \ln \underbrace{\overbrace{\frac{P(Z \mid O_i)}{P(Z \mid E_i)}}^{(1)}}_{(2)}$$

(1) probability of obtaining the sensor value we did, given the cell is occupied

(2) probability of obtaining the sensor value we did, given the cell is empty

Typically, we assign U (the log ratio) a constant value, such as 5 (a constant positive value) for cells at the end of the beam (distance of the cell is approximately the measured depth), and -2 (a constant negative value) for cells inside the beam (cell is closer than the measured depth). This will gradually converge.

Lecture 7 - SLAM (Simultaneous Localisation and Mapping)

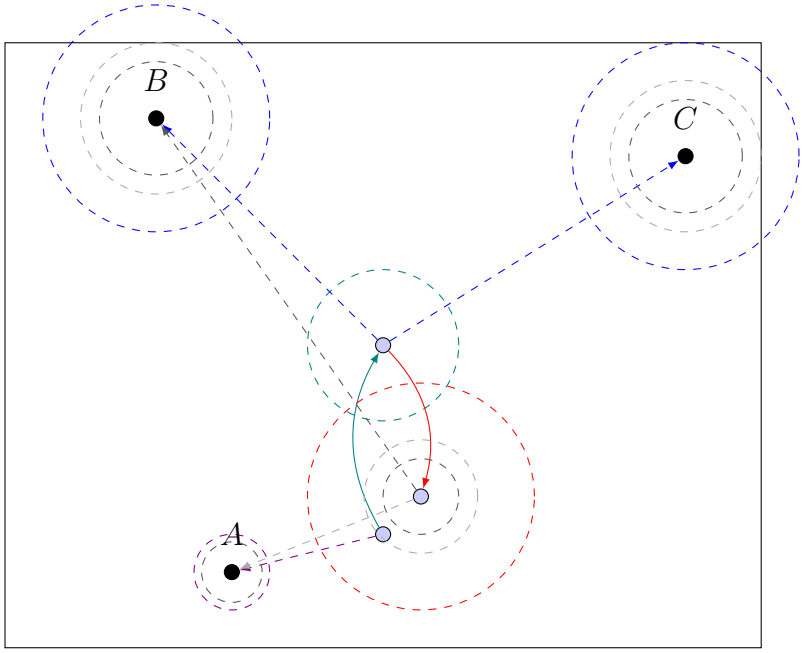
SLAM can be formally defined as a body (robot) with quantitative sensors (cameras / odometry) moving through a previously unknown, static environment, mapping it and calculating its egomotion. This is needed when the robot must be truly autonomous, little is known about the environment, when we can't place artificial beacons, and when the robot actually needs to know where it is (for example, a robot vacuum may not need SLAM as it can eventually reach all of the room with bounces).

SLAM is uncertain of the position of the robot and observes the world through sensors. In MCL, we were given a map of the environment, which we don't have here. Features are values in the robot's memory that can represent the shape of the world (such as corners in the scene). Ideally we have a finite set of features that can be observed from different viewpoints in order to localise the robot's position.

Propagating Uncertainty

Since we need to map and localise, it seems like a chicken and egg problem; however, we can make progress if we assume the robot is the only thing moving (the world is static). Note that both the map and the location are uncertain. We extend probabilistic estimation to the features of the map; a joint distribution over the states of both the mapped world and the robot, which is gradually updated over time. Instead of just estimating the uncertain position of the robot, we also estimate the uncertain position of features in the scene.

In SLAM, we don't have a coordinate frame, therefore we typically denote the starting position of the robot as the origin.



Looking at the diagram above, we can observe the following steps;

1. (violet) - the robot starts with no uncertainty, and observes landmark A with some uncertainty (due to sensor variance) and acknowledges this (possibly represented with a point cloud, or similar)
2. (teal) - the robot drives forward (assume blindly) with some odometry, but we know there is some error; the uncertainty in the position of the robot grows
3. (blue) - the robot observes new landmarks B and C, which has some sensor variance and also inherits the uncertain position of the robot (hence the uncertainty is the sum of the robot's position, and some sensor error) - however, note that the positions relative to each other (B, C, and the robot) are well known, but has higher global uncertainty due to the robot
4. (red) - robot drives back (once again, assume blindly) towards start; the uncertainty grows even further

5. (light grey) - the robot remeasures A (causing a loop closure); importantly A has very little uncertainty, and by observing it, we can reduce the robot's uncertainty - another important point is that due to the correlation between the position of the robot and the points B and C, the uncertainty in the two other points also reduce
6. (dark grey) - the robot remeasures B, note that the uncertainties of C and A also shrink

Note that the uncertainty in the position of the landmarks will reduce, but the uncertainty in the position of the robot may fluctuate with movement. With enough time moving around the scene, the landmarks should eventually converge and we have known locations - meaning we have a situation similar to MCL where the only uncertainty is the position of the robot.

Joint Gaussian Uncertainty

We assume all measurements are reasonably well characterised by Gaussian distributions. The Extended Kalman Filter is a non-linear version of a Kalman Filter.

We have a state vector $\hat{\boldsymbol{x}}$ which represents all the things we're interested in estimating, for example $\hat{\boldsymbol{x_v}}$ (the coordinates of the robot), and the coordinates of all the features / landmarks $\hat{\boldsymbol{y_1}}, \hat{\boldsymbol{y_2}}, \ldots$ - this may be a **long** state vector (for instance, the robot may be 3 parameters, and each of the features are 3D point positions - hence we'd have a total of 303 elements if we had 100 features), representing the mean / 'single best estimate';

$$\hat{\boldsymbol{x}} = \begin{bmatrix} \hat{\boldsymbol{x_v}} \\ \hat{\boldsymbol{y_1}} \\ \hat{\boldsymbol{y_2}} \\ \vdots \end{bmatrix}$$

We also have a covariance matrix, which has the same dimension (therefore we'd have a 303×303 in our previous example). Note that the off-diagonal blocks represent the correlation information (the diagonals are similar to the matrices we made in the first lab assignment) - for example $\boldsymbol{P_{x_v,y_2}}$ represents the correlation of our estimate of the robot's position with the estimate of the position of landmark 2.

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{P_{x_v,x_v}} & \boldsymbol{P_{x_v,y_1}} & \boldsymbol{P_{x_v,y_2}} & \cdots \\ \boldsymbol{P_{y_1,x_v}} & \boldsymbol{P_{y_1,y_1}} & \boldsymbol{P_{y_1,y_2}} & \cdots \\ \boldsymbol{P_{y_2,x_v}} & \boldsymbol{P_{y_2,y_1}} & \boldsymbol{P_{y_2,y_2}} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

SLAM with Single Camera

The most useful features are point landmarks. If there are a set of landmarks in the (3D) world, they will project into different positions depending on the location of the camera. We typically use the image position of landmarks, and as the camera moves around, the image positions change - giving us information both about how the camera is moving as well as the 3D structure of the map.

MonoSLAM has the goal of building a map of the scene with a single camera. In 3D space, texture patches represent each landmark. These patches will start with higher uncertainty when initialised (depth uncertainty), which will reduce as it's observed more frequently.

This idea is used in consumer products, such as robot vacuums, *ARKit* / *ARCore* in iOS / Android respectively, as well as headsets (such as *Oculus*).

Limitations

The technique of having a joint probabilistic state for everything doesn't extend to all problems we're interested in. Some issues include the size of the computation (with regards to the state vector, as well as the large covariance matrix), which increases further with more landmarks (becoming impossible

to update). When the uncertainty becomes too large, it becomes difficult to obtain the positions accurately.

The key idea is to make a large map out of a set of local tiles. Locally, we perform the joint estimation previously mentioned, but at some point, once the map becomes 'large' enough, we save the map as a tile and start making a new tile. Each of these tiles are relatively accurate, and we store a relative position of this tile. However, note that when more tiles are built, the incremental uncertainty between tiles will increase (for example, once we perform this around a building, we may end up back where we start, but we've estimated a different location). We need to add a place recognition component, which allows us to check if we're at a previously visited location (similar to signature recognition). If we determine that we're back to a previous position, we can perform some global optimisation which pulls the tiles together to make it globally consistent; this is **loop closure**.

SLAM can be **purely topological**, with a graph-based representation by only using place recognition. A record of visited places is kept (as well as how they connect together).

When two places are close together (according to the recognition component), we can update the metric map (which has incorrect drift) using **pose graph optimisation / relaxation**. Essentially, if we have some particular estimate of the nodes (when we end a tile) in the map (at a particular time), we want to determine the likelihood of the relative measurements that we've made. We want the measurements that are most likely, given the new information that two nodes are close together.