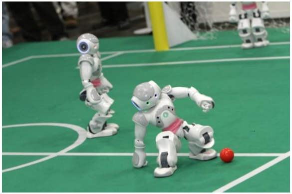
Robot Localisation

3007/7059 Artificial Intelligence

School of Computer Science
The University of Adelaide

Robot localisation

Many robotic applications require finding out where things are. **Localisation** is at the core of any successful physical interaction with the environment.



Examples:

- Knowing the relative position of the soccer ball and the robot(s) is crucial for robot soccer.
- In order to move towards the washing machine the domestic helper robot needs to know where it currently is.

Additional reading

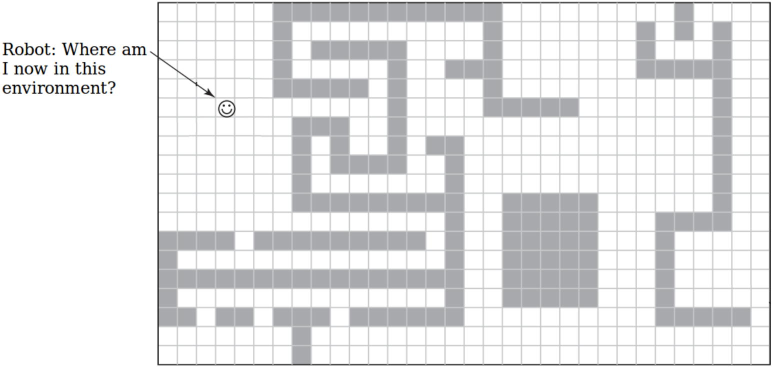
Much of the material in this and the next two lectures is discussed in much greater detail in:

Thrun, Burghard, Fox
Probabilistic Robotics
MIT Press, 2005

[note that it also has an alternative treatment of MDP and POMDP]

Self localisation

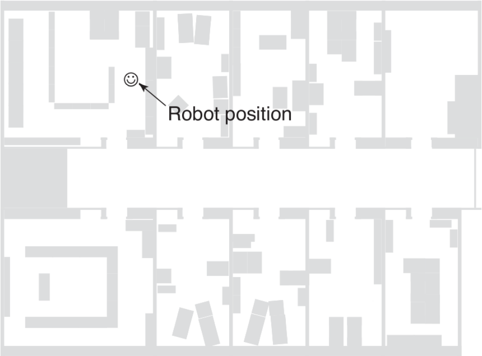
Self localisation is the task of finding the position of the robot within an environment.



We assume **we have the precise map**, but...

Self localisation (cont.)

... real environments is not discretised into a grid.

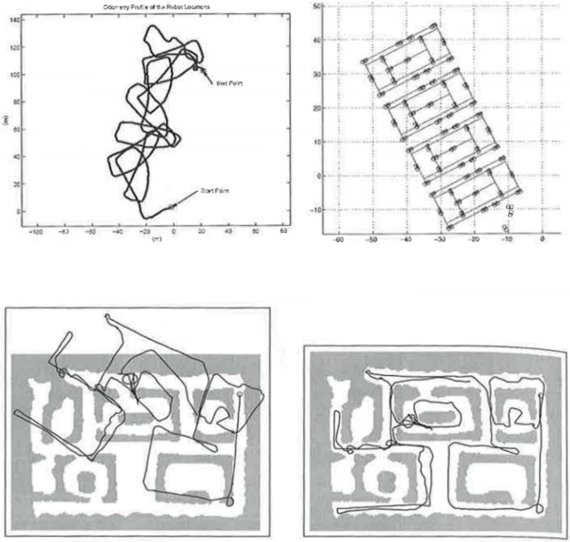


So we have an **infinite number of continuous states** instead of a finite number of discrete states.

Self localisation (cont.)

Q: Why is this so hard? Why can't we just remember where we started, record all the moves we have performed, and integrate over them to calculate the current position?
A: ???

Self localisation (cont.)



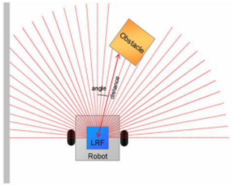
[figures from Thrun et al]

Self localisation (cont.)

A purely **feedforward** method is unlikely to work well, since the actions/moves are subject to noise or error.

Moreover, we **may make mistakes** with the moves (e.g., bump into a wall or other moving obstacles) and a purely feedforward method cannot predict what could have happened.

We need to take into account **feedback** from the environment, e.g., detecting walls with a laser scanner (with the aid of the known map, this may tell us roughly where we are).

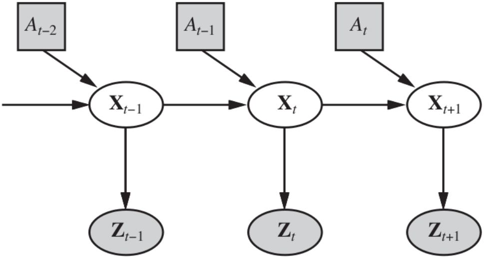


Localisation as state estimation

Localisation can be cast as the problem of estimating the **current state** $\mathbf{X}_t$ of the robot, based on

- the action $\mathbf{A}_{t-1}$ executed at the previous time step.
- and the observations of the environment $\mathbf{Z}_t$ obtained at the present time step.

Putting all variables and observations into a **chain structure**.



Shaded square boxes mean the values of the associated variables are known. Unshaded circles mean unknown/latent variables.

Localisation as state estimation (cont.)

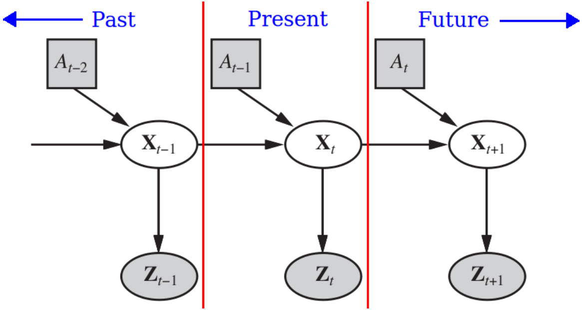
What is a “state”? A state encapsulates what we want to know about the robot, e.g.,

$$\mathbf{X}_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x \text{ coordinate of the robot} \\ y \text{ coordinate of the robot} \\ \text{heading/orientation of the robot} \end{bmatrix}$$

We may also include other properties like velocity, acceleration, battery level, etc.

Filtering as recursive update

State estimation is also called **filtering** since involves estimating the state $\mathbf{X}_t$ from noisy observations $\mathbf{Z}_t$.



The formal goal of filtering is this:
Given that we executed actions $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_{t-1}$ and receive observations $\mathbf{Z}_1, \mathbf{Z}_2, \ldots, \mathbf{Z}_t$, find the current state $\mathbf{X}_t$.

Filtering as recursive update (cont.)

Knowledge about the state $\mathbf{X}_t$ is encoded in a **belief state**

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$

where $\mathbf{Z}_{1:t} = \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\}$ and $\mathbf{A}_{1:t-1} = \{\mathbf{A}_1, \ldots, \mathbf{A}_{t-1}\}$. The belief state is a **probability distribution** over the space of states.

The filtering task is a **recursive update** of the belief state

$$\begin{aligned} &P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1}) \\ &= \alpha P(\mathbf{Z}_t|\mathbf{X}_t) \int P(\mathbf{X}_t|\mathbf{X}_{t-1}, \mathbf{A}_{t-1}) P(\mathbf{X}_{t-1}|\mathbf{Z}_{1:t-1}, \mathbf{A}_{1:t-2}) d\mathbf{X}_{t-1} \end{aligned}$$

α is a normalisation constant.

Filtering as recursive update (cont.)

Knowledge about the state $\mathbf{X}_t$ is encoded in a **belief state**

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$

where $\mathbf{Z}_{1:t} = \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\}$ and $\mathbf{A}_{1:t-1} = \{\mathbf{A}_1, \ldots, \mathbf{A}_{t-1}\}$. The belief state is a **probability distribution** over the space of states.

The filtering task is a **recursive update** of the belief state

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$
$$= \alpha P(\mathbf{Z}_t|\mathbf{X}_t) \underbrace{\int P(\mathbf{X}_t|\mathbf{X}_{t-1}, \mathbf{A}_{t-1}) P(\mathbf{X}_{t-1}|\mathbf{Z}_{1:t-1}, \mathbf{A}_{1:t-2}) d\mathbf{X}_{t-1}}_{\text{Integration over all possible previous states}}$$

Filtering as recursive update (cont.)

Knowledge about the state $\mathbf{X}_t$ is encoded in a **belief state**

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$

where $\mathbf{Z}_{1:t} = \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\}$ and $\mathbf{A}_{1:t-1} = \{\mathbf{A}_1, \ldots, \mathbf{A}_{t-1}\}$. The belief state is a **probability distribution** over the space of states.

The filtering task is a **recursive update** of the belief state

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$
$$= \alpha P(\mathbf{Z}_t|\mathbf{X}_t) \int P(\mathbf{X}_t|\mathbf{X}_{t-1}, \mathbf{A}_{t-1}) \underbrace{P(\mathbf{X}_{t-1}|\mathbf{Z}_{1:t-1}, \mathbf{A}_{1:t-2})}_{\text{Belief state at previous time step}} d\mathbf{X}_{t-1}$$

Filtering as recursive update (cont.)

Knowledge about the state $\mathbf{X}_t$ is encoded in a **belief state**

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$

where $\mathbf{Z}_{1:t} = \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\}$ and $\mathbf{A}_{1:t-1} = \{\mathbf{A}_1, \ldots, \mathbf{A}_{t-1}\}$. The belief state is a **probability distribution** over the space of states.

The filtering task is a **recursive update** of the belief state

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$
$$= \alpha P(\mathbf{Z}_t|\mathbf{X}_t) \int \underbrace{P(\mathbf{X}_t|\mathbf{X}_{t-1}, \mathbf{A}_{t-1})}_{\text{Motion model}} P(\mathbf{X}_{t-1}|\mathbf{Z}_{1:t-1}, \mathbf{A}_{1:t-2}) d\mathbf{X}_{t-1}$$

Filtering as recursive update (cont.)

Knowledge about the state $\mathbf{X}_t$ is encoded in a **belief state**

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t},\mathbf{A}_{1:t-1})$$

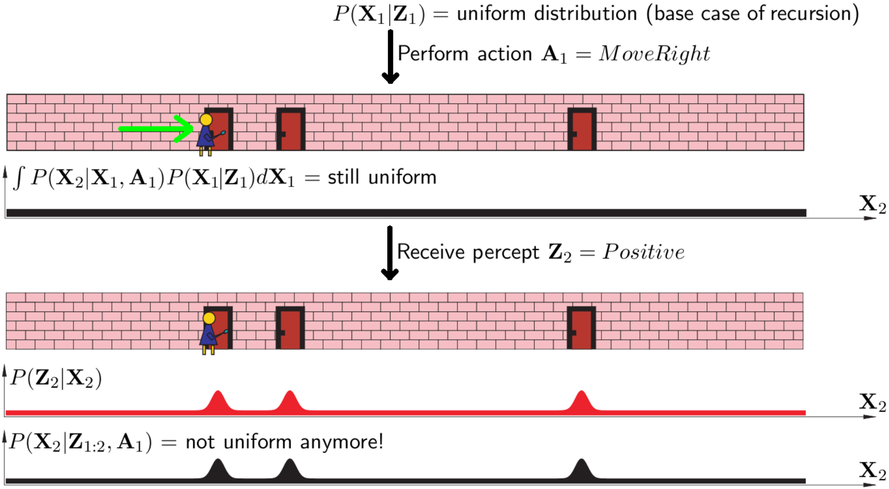
where $\mathbf{Z}_{1:t} = \{\mathbf{Z}_1,\dots,\mathbf{Z}_t\}$ and $\mathbf{A}_{1:t-1} = \{\mathbf{A}_1,\dots,\mathbf{A}_{t-1}\}$. The belief state is a **probability distribution** over the space of states.

The filtering task is a **recursive update** of the belief state

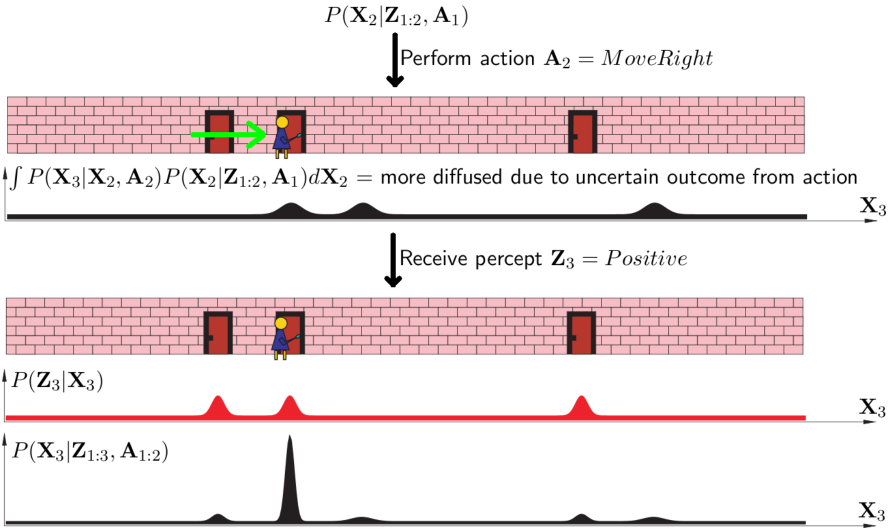
$$P(\mathbf{X}_t|\mathbf{Z}_{1:t},\mathbf{A}_{1:t-1})$$
$$= \alpha \underbrace{P(\mathbf{Z}_t|\mathbf{X}_t)}_{\text{Sensor model}} \int P(\mathbf{X}_t|\mathbf{X}_{t-1},\mathbf{A}_{t-1}) P(\mathbf{X}_{t-1}|\mathbf{Z}_{1:t-1},\mathbf{A}_{1:t-2}) d\mathbf{X}_{t-1}$$

Example

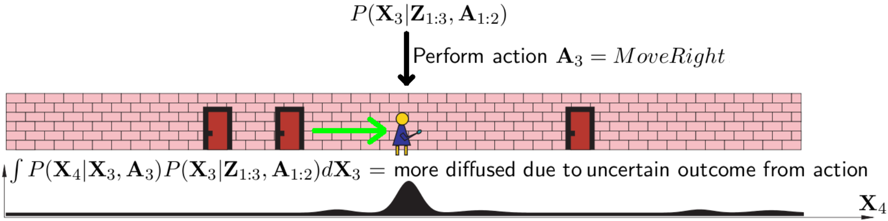
Robot in 1D map. Sensor detects whether robot is near a door.



Example (cont.)



Example (cont.)



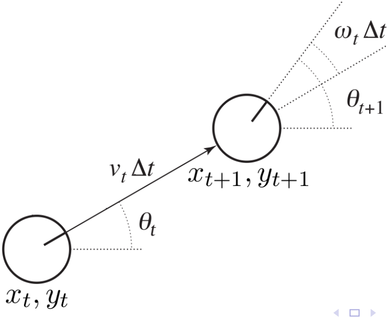
Motion model

The **motion model** $P(\mathbf{X}_{t+1}|\mathbf{X}_t, \mathbf{A}_t)$ is obviously tied to the definition of the state, i.e., what each value in $\mathbf{X}_t$ means.

For $\mathbf{X}_t = [x_t \ y_t \ \theta_t]^T$, a **deterministic** motion model is

$$\mathbf{X}_t \xrightarrow[\mathbf{A}_t=\{v_t,\omega_t\}]{\text{Conduct action}} \hat{\mathbf{X}}_{t+1} = \mathbf{X}_t + \begin{bmatrix} v_t \cdot \Delta t \cdot \cos\theta_t \\ v_t \cdot \Delta t \cdot \sin\theta_t \\ \omega_t \cdot \Delta t \end{bmatrix}$$

v_t is the **translational velocity** and ω_t the **angular velocity**.



Motion model (cont.)

Of course, the actual motion undertaken is **not deterministic**. This may be modelled by a Gaussian distribution

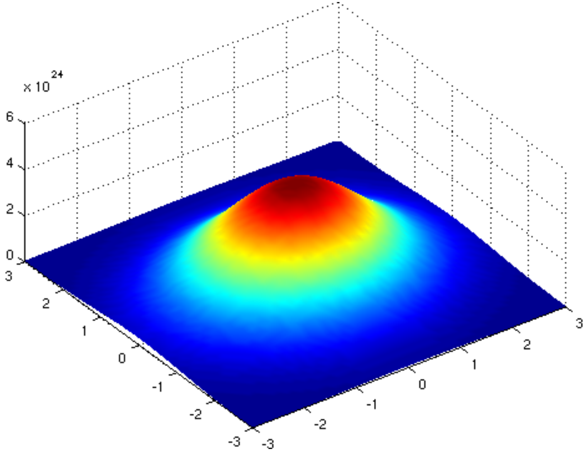
$$
\begin{aligned}
P(\mathbf{X}_{t+1}|\mathbf{X}_t, \mathbf{A}_t) &= \mathcal{N}(\mathbf{X}_{t+1}|\hat{\mathbf{X}}_{t+1}, \boldsymbol{\Sigma}) \\
&= \frac{1}{\sqrt{(2\pi)^d|\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\Delta\mathbf{X})^T \boldsymbol{\Sigma}^{-1} \Delta\mathbf{X}\right)
\end{aligned}
$$

where

- d is the length of vector $\mathbf{X}$.
- $\Delta\mathbf{X} = \mathbf{X}_{t+1} - \hat{\mathbf{X}}_{t+1}$
- $\boldsymbol{\Sigma}$ is a $d \times d$ matrix called the **covariance matrix** of the Gaussian distribution.

Motion model (cont.)

Here's what a 2D Gaussian distribution $\mathcal{N}(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ looks like:

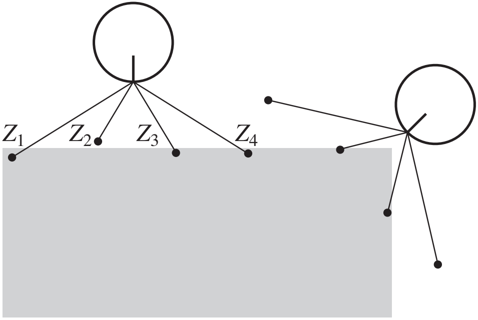


The mean vector $\boldsymbol{\mu}$ corresponds to the location of the mode (peak) of the distribution, while covariance $\boldsymbol{\Sigma}$ controls the spread.

Sensor model

Of course, the **sensor model** $P(\mathbf{Z}_t|\mathbf{X}_t)$ depends on the type of sensor used.

For example, the **range-scan** sensor model (laser, sonar, etc) which has a fixed bearing relative to the robot, and produces M range measurements $\mathbf{Z}_t = \{Z_1, Z_2, \ldots, Z_M\}$.



(This example has 4 range measurements $M = 4$ per scan.)

Sensor model (cont.)

If we know $\mathbf{X}_t$, which entails knowing the position and orientation of the robot, we can analytically work out the expected measurements $\hat{\mathbf{Z}}_t = \{\hat{Z}_1, \hat{Z}_2, \ldots, \hat{Z}_M\}$ from $\mathbf{X}_t$ using the map.

We can then compare the actual measurements $\mathbf{Z}_t$ with the expected measurements $\hat{\mathbf{Z}}_t$

$$P(\mathbf{Z}_t|\mathbf{X}_t) = \alpha \prod_{j=1}^{M} \exp\left(-\frac{(Z_j - \hat{Z}_j)^2}{2\sigma^2}\right)$$

where α is a normalisation constant, and σ^2 is the common variance for all range measurements.

Kalman filter

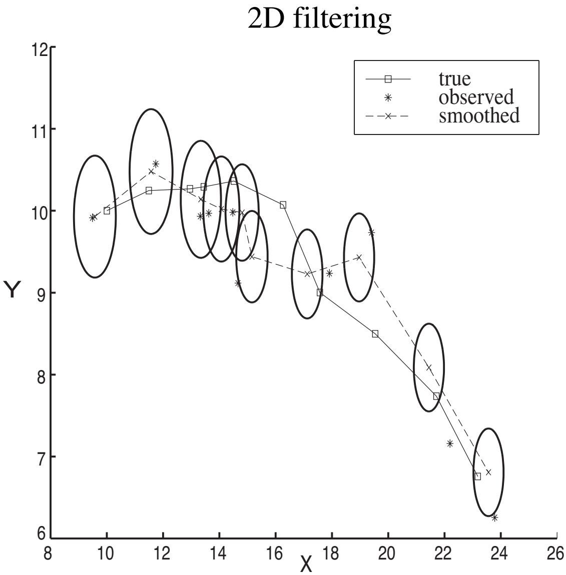
The Kalman filter is one of the most widely used methods to perform the recursive state update. The idea is to represent the belief state $P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$ using a **Gaussian distribution**.

If the motion model and sensor model are **also Gaussian** (the examples given are), we are guaranteed that the updated belief state **remains Gaussian**:

$$P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$$
$$= \alpha \underbrace{P(\mathbf{Z}_t|\mathbf{X}_t)}_{Gaussian} \int \underbrace{P(\mathbf{X}_t|\mathbf{X}_{t-1}, \mathbf{A}_{t-1})}_{Gaussian} \underbrace{P(\mathbf{X}_{t-1}|\mathbf{Z}_{1:t-1}, \mathbf{A}_{1:t-2})}_{Gaussian} d\mathbf{X}_{t-1}$$

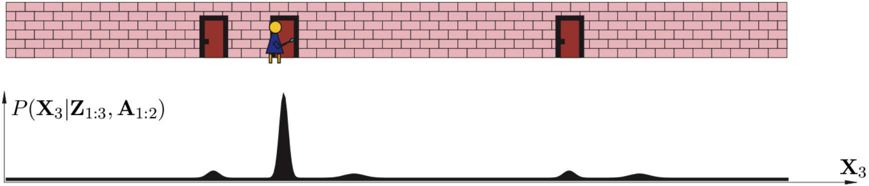
So all we need to do is to **propagate and update the mean vector and covariance matrix** of the belief state — the equations are messy but the idea is simple.

Kalman filter (cont.)



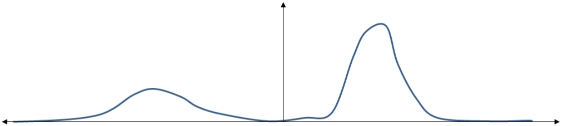
Kalman filter (cont.)

While Gaussian assumptions are valid in some applications (the Kalman filter was used on the Apollo navigation computer), it is highly inadequate in others:



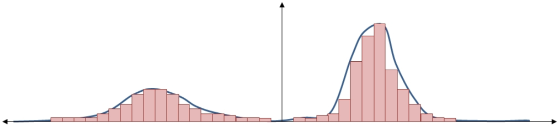
Non Gaussian distributions

A major problem with the Gaussian is that is is unimodal.
What if we want/need to represent a distribution with multiple peaks



Non Gaussian distributions (cont.)

We could use a histogram approximation

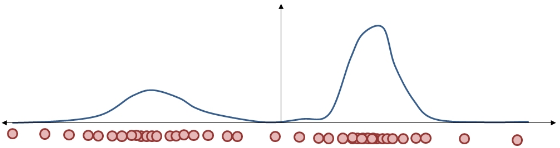


But the memory cost of doing so is huge, even for one dimensional distributions (exponential in number of dimensions)

Particle filtering

The **particle filter** is conceptually very simple and does not impose Gaussian assumptions.

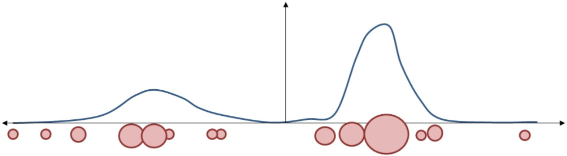
The basic idea is to represent the belief state $P(\mathbf{X}_t|\mathbf{Z}_{1:t}, \mathbf{A}_{1:t-1})$ using a set of **samples** or **particles** (much like approximate inference methods for Bayesian networks).



The local density of particles is then an approximation for the probability density.
Note that we can prepresent a distribution to arbitrary accuracy if we have sufficient particles.

Particle filtering (cont.)

There are different ways we can represent the same distribution:



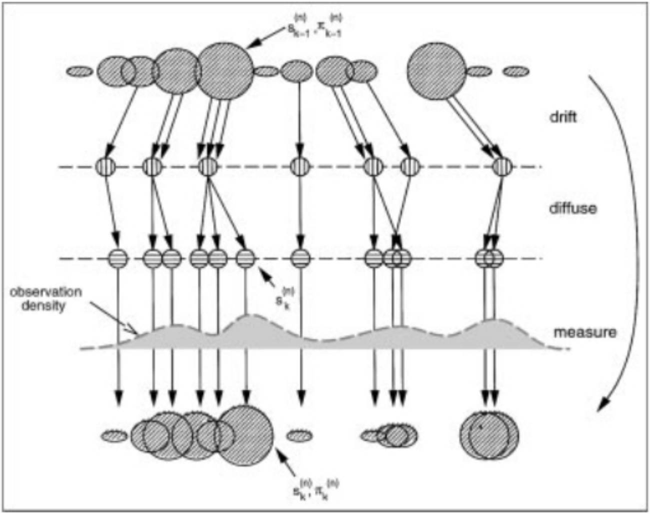
Here each particle has a weight representing local probability mass.

Particle filtering (cont.)

The main steps of the particle filter are then as follows:

- Initialise N particles $\{\mathbf{X}_1^i\}_{i=1}^N$ by sampling from $P(\mathbf{X}_1|\mathbf{Z}_1)$ (the base case).
- Each particle $\mathbf{X}_t^i$ is propagated forward by sampling the next state value $\mathbf{X}_{t+1}^i$ using the motion model $P(\mathbf{X}_{t+1}^i|\mathbf{X}_t^i, \mathbf{A}_t)$.
- Each particle $\mathbf{X}_{t+1}^i$ is weighted by the likelihood it assigns to the new evidence $\mathbf{Z}_{t+1}$ using the sensor model $P(\mathbf{Z}_{t+1}|\mathbf{X}_{t+1}^i)$.
- The population of particles is **resampled** based on the weights to generate N new particles.
- Repeat from Step 2.

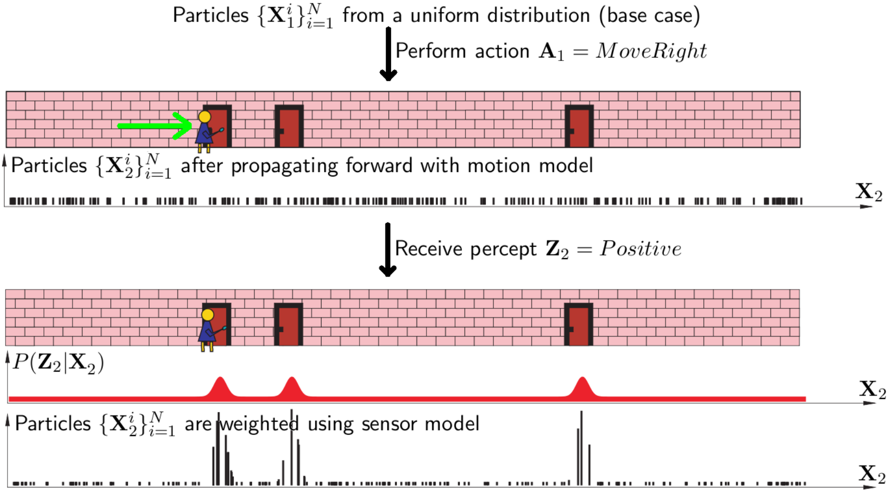
Visually



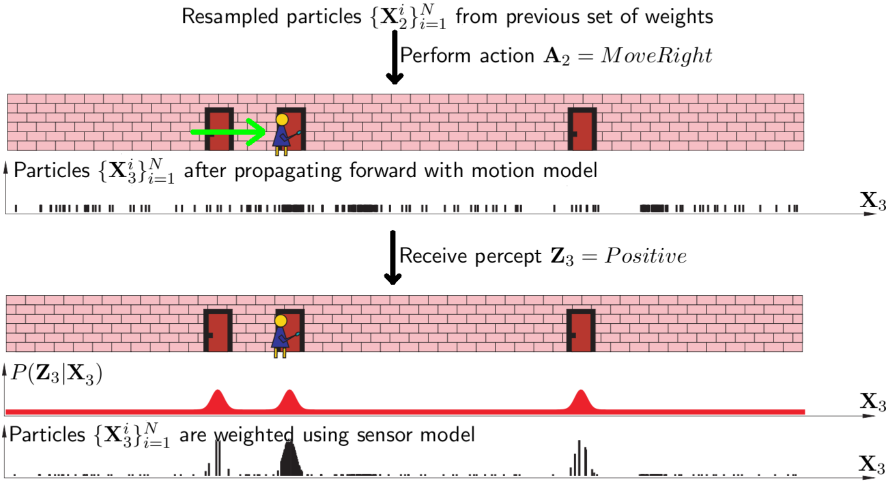
[Figure from Isard and Blake, ECCV 1996]

Example

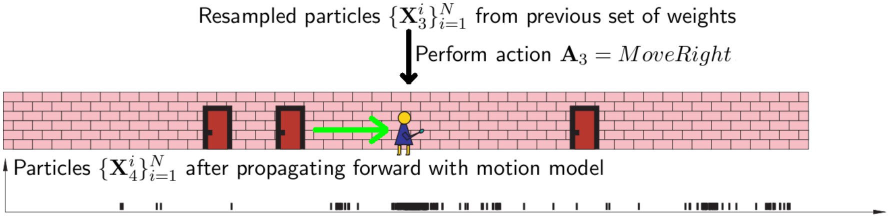
Robot in 1D map. Sensor detects whether robot is near a door.



Example (cont.)



Example (cont.)



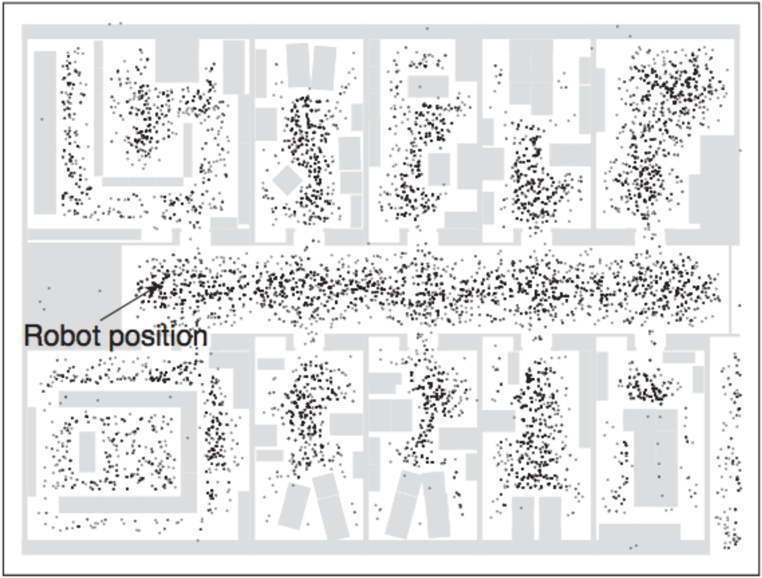
Particle filter for robot localisation

function MONTE-CARLO-LOCALIZATION(a, z, N, $P(X'|X,\ v,\ \omega)$, $P(z|z^*)$, m) **returns** a set of samples for the next time step
  inputs: a, robot velocities v and ω
    z, range scan $z_1, \ldots, z_M$
    $P(X'|X,\ v,\ \omega)$, motion model
    $P(z|z^*)$, range sensor noise model
    m, 2D map of the environment
  persistent: S, a vector of samples of size N
  local variables: W, a vector of weights of size N
    S', a temporary vector of particles of size N
    W', a vector of weights of size N

```
if S is empty then        /* initialization phase */
    for i = 1 to N do
        S[i] ← sample from P(X_0)
    for i = 1 to N do     /* update cycle */
        S'[i] ← sample from P(X'|X = S[i], v, ω)
        W'[i] ← 1
        for j = 1 to M do
            z* ← RAYCAST(j, X = S'[i], m)
            W'[i] ← W'[i] · P(z_j | z*)
    S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S', W')
return S
```

Example

Black dots are particles:

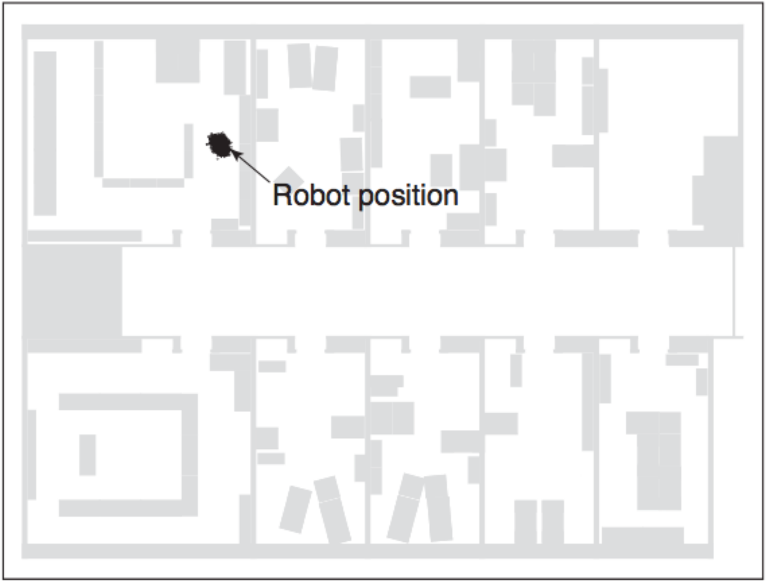


Example (cont.)

Particles after scanning the corridor:

Example (cont.)

Particles after entering the room:



SLAM

What if we don't have the map? The robot will have to build the map on the fly.

This is another **chicken-and-egg** problem: The robot has to localise itself within an unknown map, and build the map while it doesn't know its actual location.

The task is called **simultaneous localisation and mapping (SLAM)**.