ESE 650
Learning in Robotics

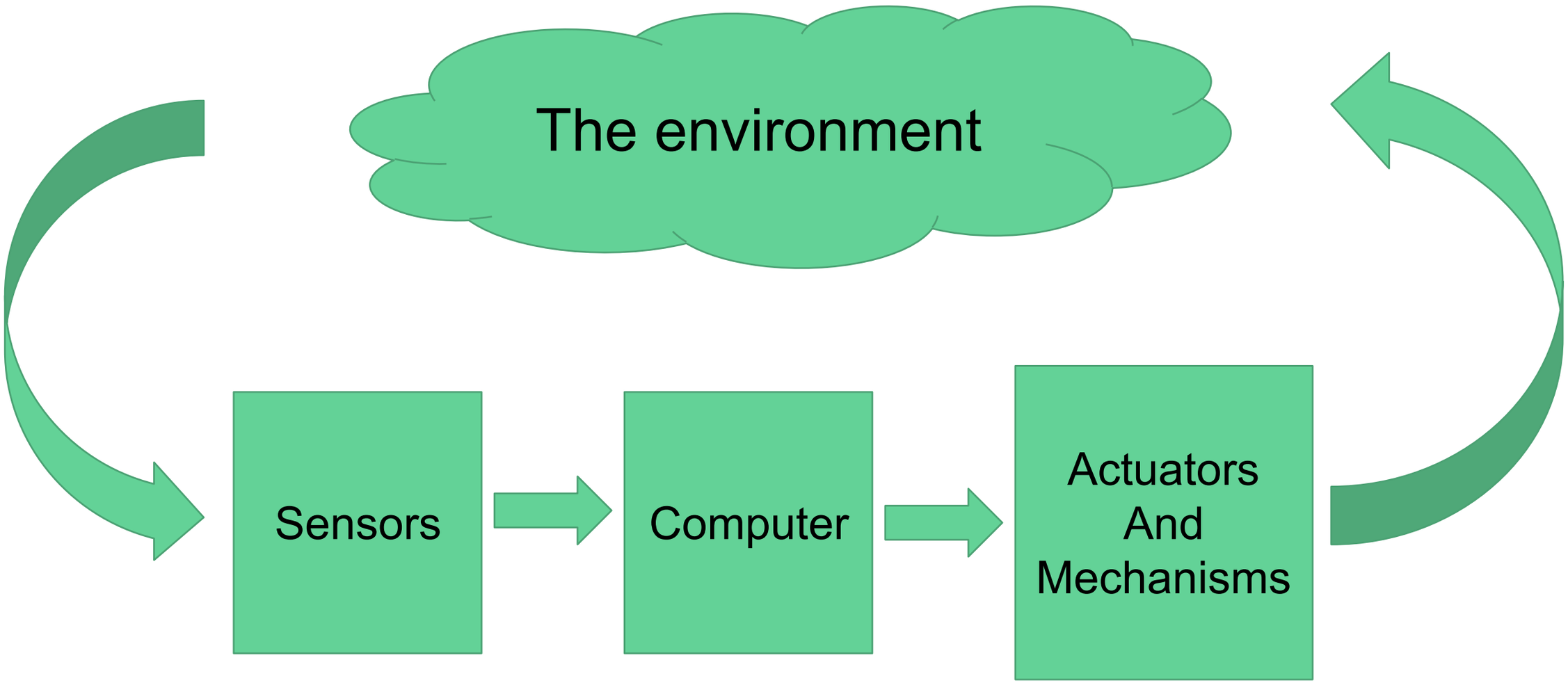
Spring 2019

Instructors

- Faculty
  - Pratik Chaudhari: ESE
  - C.J. Taylor: CIS
- Teaching Assistants
  - Arbaaz Khan
  - Ian Miller
  - Ty Nguyen
  - Zhexian Xie
  - Elijah Lee

Gratuitous pictures of robots

What is a robot



Course Goals

- The goal of the class will be to introduce you to some of the manifold ways that robotic systems learn from data
  - One or more robots gathering data and using it to build a model of their environment
  - Using data collected over multiple trials to find good strategies for achieving a goal

Topics

- Module 1 : State Estimation
- Module 2 : Control and Planning
- Module 3 : Reinforcement Learning
- Module 4 : Other Learning Schemes
    - Transfer learning, meta-learning, multi-task learning, self-supervised learning

Course Texts

- Recommended texts
  - "Probablistic Robotics", Thrun, Burgard and Fox
  - "Introduction to Probability", Blitzstein and Hwang
  - "Introduction to Linear Algebra", Strang

Course Logistics

- There is a Canvas page for the course which will have
  - Assignments
  - Grades
  - Files and handouts
  - Course Syllabus
  - Link to Piazza site for course
  - Link to Gradescope for submitting assignments

Course Logistics

- There will be a Piazza site linked from the Canvas page which can be used to ask questions
- Please make your questions public
- Please use search feature to see if your question has been addressed in an earlier post.

Course Organization

- There will be multiple homework involving both written and programming assignments (Python) - 50% of grade
- There will be a midterm - 20% of grade
- There will be a final project - 30% of grade

Probability Review

Why Probability

- Probability gives us a language to talk about what we believe about what we know and what we don't know.
- Uncertainty is a fact of life in robotics.
  - In sensing
  - In actuation

Discrete Probability

- **Experiment**: any procedure that can be repeated infinitely and has a well-defined set of possible outcomes.
- **Sample space** Ω: the set of possible outcomes of an experiment.
  - $\Omega = \{HH, HT, TH, TT\}$
- **Event** A: a subset of the possible outcomes Ω
  - $A = \{HH\},\ B = \{HT, TH\}$
- **Probability of an event**: $\mathbb{P}(A)$ - A function that maps events to numbers between 0 and 1.

Probability Review

- **Probability Axioms**:
  - $\mathbb{P}(A) \geq 0$
  - $\mathbb{P}(\Omega) = 1$
  - If $\{A_i\}$ are disjoint $(A_i \cap A_j = \emptyset)$, then $\mathbb{P}(\bigcup_i A_i) = \sum_i \mathbb{P}(A_i)$
- **Corollary**
  - $\mathbb{P}(\emptyset) = 0$
  - $\max\{\mathbb{P}(A), \mathbb{P}(B)\} \leq \mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B) \leq \mathbb{P}(A) + \mathbb{P}(B)$
  - $A \subseteq B \Rightarrow \mathbb{P}(A) \leq \mathbb{P}(B)$
  - $P(A^c) = 1 - P(A)$

Probability Review

- **Conditional Probability**: $\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}, \quad \mathbb{P}(B) \neq 0$
- **Total Probability Theorem**: If $\{A_1, \ldots, A_n\}$ is a partition of Ω, i.e., $\Omega = \bigcup_i A_i$ and $A_i \cap A_j = \emptyset, i \neq j$, then:

$$\mathbb{P}(B) = \sum_{i=1}^{n} \mathbb{P}(B \cap A_i)$$

- **Bayes Theorem** If $\{A_1, \ldots, A_n\}$ is a partition of Ω, then:

$$\mathbb{P}(A_i \mid B) = \frac{\mathbb{P}(B \mid A_i)\mathbb{P}(A_i)}{\sum_{j=1}^{n} \mathbb{P}(B \mid A_j)\mathbb{P}(A_j)}$$

- **Independent events**: $\mathbb{P}(\bigcap_i A_i) = \prod_i \mathbb{P}(A_i)$
  - observing one does not give any information about another
  - in contrast, disjoint events never occur together: one occurring tells you that others will not occur and hence, *disjoint events are always dependent*

Probability Review

We can also talk about experiments that produce real number values but this requires more work.

- σ**-algebra**: a collection of subsets of Ω closed under complementation and countable unions.
- **Measurable space**: a tuple $(\Omega, \mathcal{F})$, where Ω is a sample space and $\mathcal{F}$ is a σ-algebra.
- **Measure**: a function $\mu : \mathcal{F} \to \mathbb{R}$ satisfying $\mu(A) \geq \mu(\emptyset) = 0$ for all $A \in \mathcal{F}$ and countable additivity $\mu(\cup_i A_i) = \sum_i \mu(A_i)$ for disjoint A_i.
- A measure μ is σ**-finite** on $(\Omega, \mathcal{F})$, if Ω can be obtained as the countable union $\cup_n A_n$ of sets $A_n \in \mathcal{F}$ of finite measure, $\mu(A_n) < \infty$.

Probability Review

- **Probability measure**: a measure that satisfies $\mu(\Omega) = 1$.
- **Probability space**: a triple $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is a sample space, $\mathcal{F}$ is a σ-algebra, and $\mathbb{P} : \mathcal{F} \to [0, 1]$ is a probability measure.

Random Variables

- **Random variable** X: an $\mathcal{F}$-measurable *function* from $(\Omega, \mathcal{F})$ to $(\mathbb{R}, \mathcal{B})$, i.e., a function $X : \Omega \to \mathbb{R}$ s.t. the preimage of every set in $\mathcal{B}$ is in $\mathcal{F}$.
- **Cumulative Distribution Function (CDF) $F(x)$ of a random variable** X: a function $F(x) := \mathbb{P}(X \leq x)$ that is *non-decreasing*, *right-continuous*, and $\lim_{x\to\infty} F(x) = 1$ and $\lim_{x\to-\infty} F(x) = 0$.
- **Probability Distribution Function (PDF) $f(x)$** such that

$$F(x) = \int_{-\infty}^{x} f(y)dy = \mathbb{P}(X \leq x)$$

- **Density/mass function $f(x)$ of a random variable** X

Continuous RV

$X : (\Omega, \mathcal{F}, \mathbb{P}) \to (\mathbb{R}, \mathcal{B}, \mathbb{P} \circ X^{-1})$

$f(x) \geq 0$

$\int f(y)dy = 1$

$F(x) = \int_{-\infty}^{x} f(y)dy = \mathbb{P}(X \leq x)$

$\mathbb{P}(X = x) = F(x) - F(x^-) = \lim_{\epsilon\to 0} \int_{x-\epsilon}^{x} f(y)dy = 0$

$\mathbb{P}(a < X \leq b) = F(b) - F(a) = \int_a^b f(x)dx$

Discrete RV

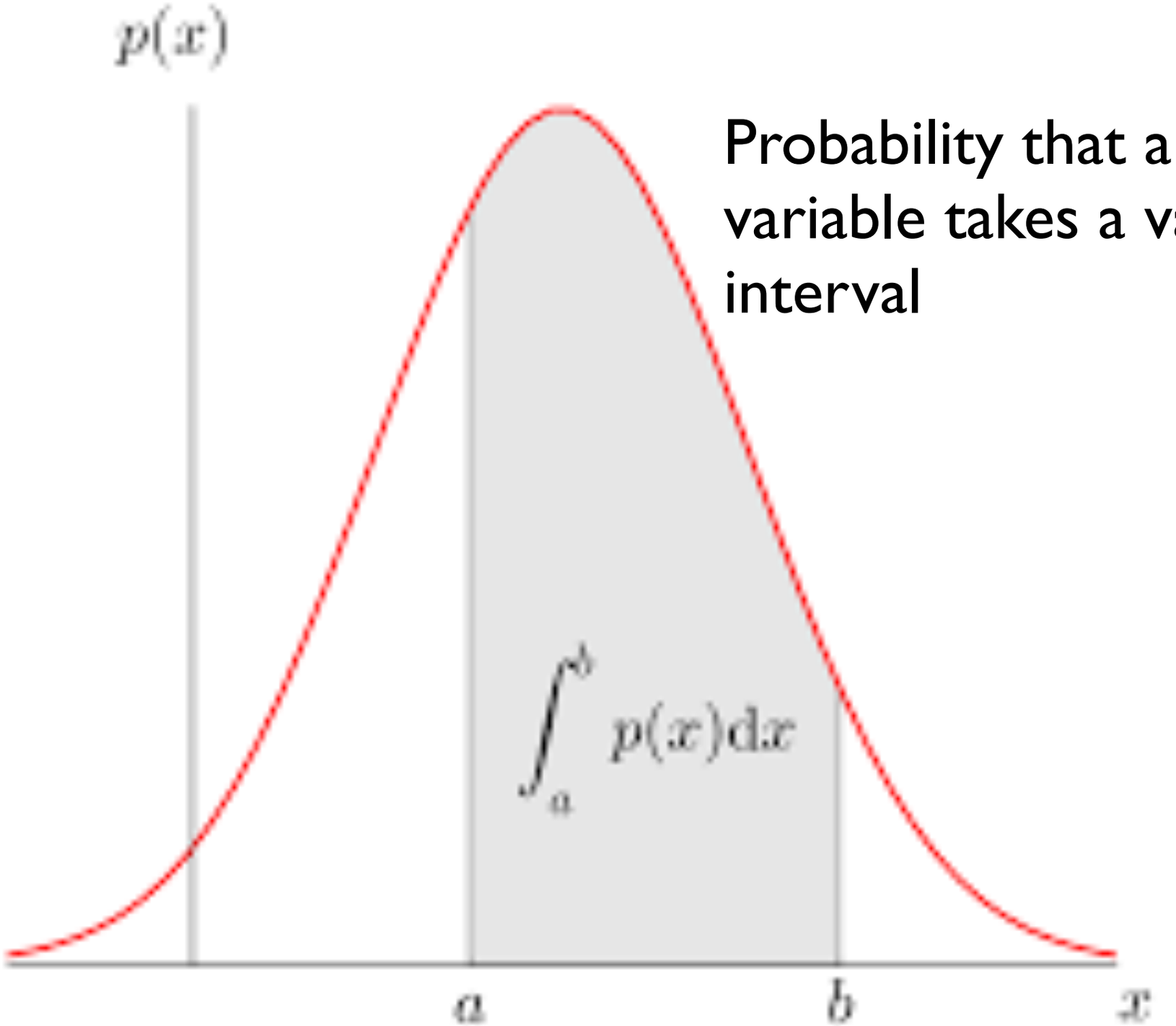
$X : (\Omega, 2^{\Omega}, \mathbb{P}) \to (\mathbb{R}, \mathcal{B}, \mathbb{P} \circ X^{-1})$

$f(x) = \mathbb{P}(X = x) \geq 0$

$\sum_i f(i) = 1$

$F(x) = \sum_{i\in\mathbb{Z}, i\leq x} f(i) = \mathbb{P}(X \leq x)$

Probability Density Function



Probability that a random variable takes a value in this interval

Probability Review

- **Lebesgue Integration**: The integral $\int g d\mu$ of a measurable function g on measurable space $(\Omega, \mathcal{F})$ with a σ-finite measure μ can be defined. In the case that μ has a pdf p, the Lebesgue integral is equivalent to a Riemann integral: $\int g d\mu = \int g(x)p(x)dx$.

- **Expectation**: Given a random variable $X : (\Omega, \mathcal{F}, \mathbb{P}) \to (\mathbb{R}^n, \mathcal{B}^n, \mathbb{P} \circ X^{-1})$ and a measurable function $g : (\mathbb{R}^n, \mathcal{B}^n, \mathbb{P} \circ X^{-1}) \to (\mathbb{R}^m, \mathcal{B}^m, \mathcal{L})$, the expectation of $g(X)$ is defined as follows:

$$\mathbb{E}[g(X)] = \int_\Omega g(X(\omega))d\mathbb{P}(\omega) = \int_{\mathbb{R}^n} g(x)d\mathbb{P}(X^{-1}(x)) = \int_{\mathbb{R}^m} y d\mathcal{L}(y)$$

  When X has a pdf p and g has a pdf l, the above simplifies to:

$$\mathbb{E}[g(X)] = \int_{\mathbb{R}^n} g(x)p(x)dx = \int_{\mathbb{R}^m} y l(y) dy$$

- **Variance of a random variable** X: $Var[X] := \mathbb{E}\big[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^T\big] = \mathbb{E}\big[XX^T\big] - \mathbb{E}[X]\mathbb{E}[X]^T$

Joint Distribution

- The **joint distribution** of random variables $\{X_i\}_{i=1}^n$ on $(\Omega, \mathcal{F}, \mathbb{P})$ defines their simultaneous behavior and is associated with a cumulative distribution function $F(x_1, \ldots, x_n) := \mathbb{P}(X_1 \leq x_1, \ldots, X_n \leq x_n)$. The CDF $F_i(x_i)$ of X_i defines its **marginal distribution**.

- Random variables $\{X_i\}_{i=1}^n$ on $(\Omega, \mathcal{F}, \mathbb{P})$ are **jointly independent** iff for all $\{A_i\}_{i=1}^n \subset \mathcal{F}$, $\mathbb{P}(X_i \in A_i, \forall i) = \prod_{i=1}^n \mathbb{P}(X_i \in A_i)$

- Let X and Y be random variables and suppose $\mathbb{E}X$, $\mathbb{E}Y$, and $\mathbb{E}XY$ exist. Then, X and Y are **uncorrelated** iff $\mathbb{E}XY = \mathbb{E}X\mathbb{E}Y$ or equivalently $Cov(X, Y) = 0$.

- Independence implies uncorrelatedness

- Two random variables X and Y are orthogonal if $\mathbb{E}[X^T Y] = 0$

Joint Distribution

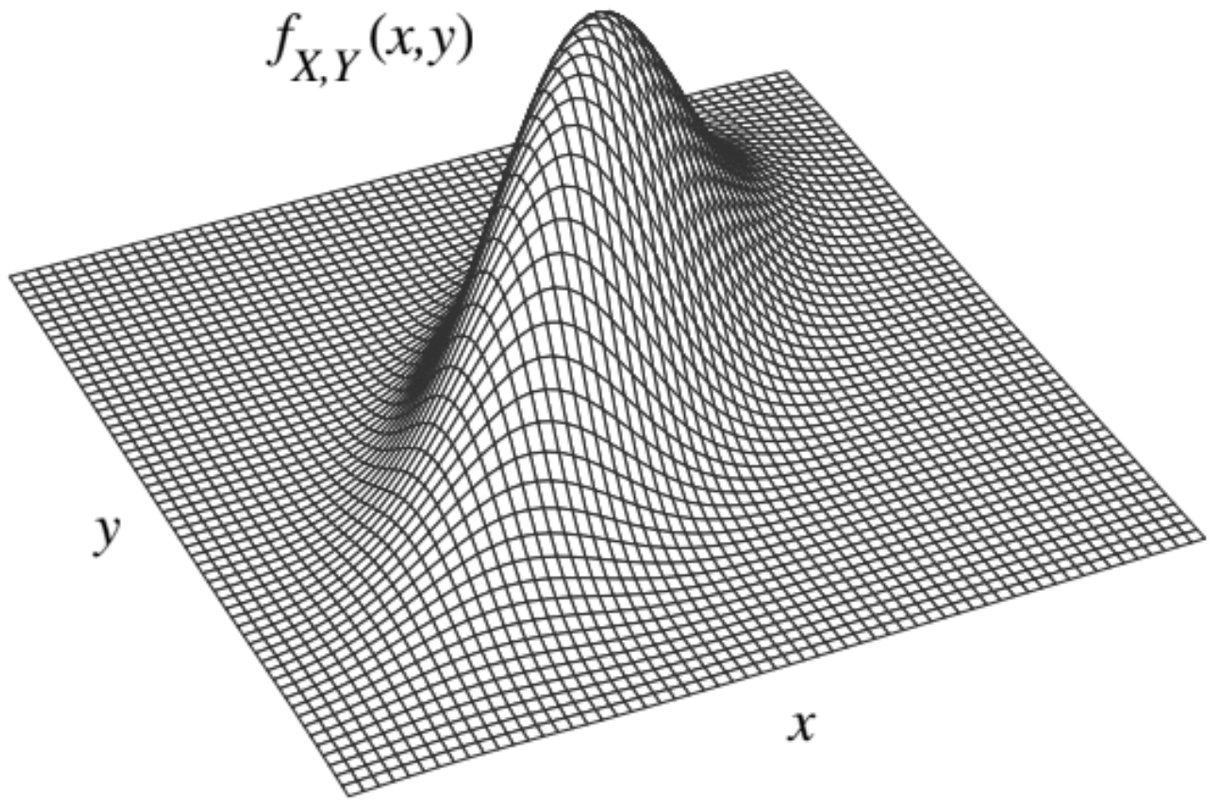


FIGURE 7.4
Joint PDF of continuous r.v.s X and Y.

Conditional PDFs

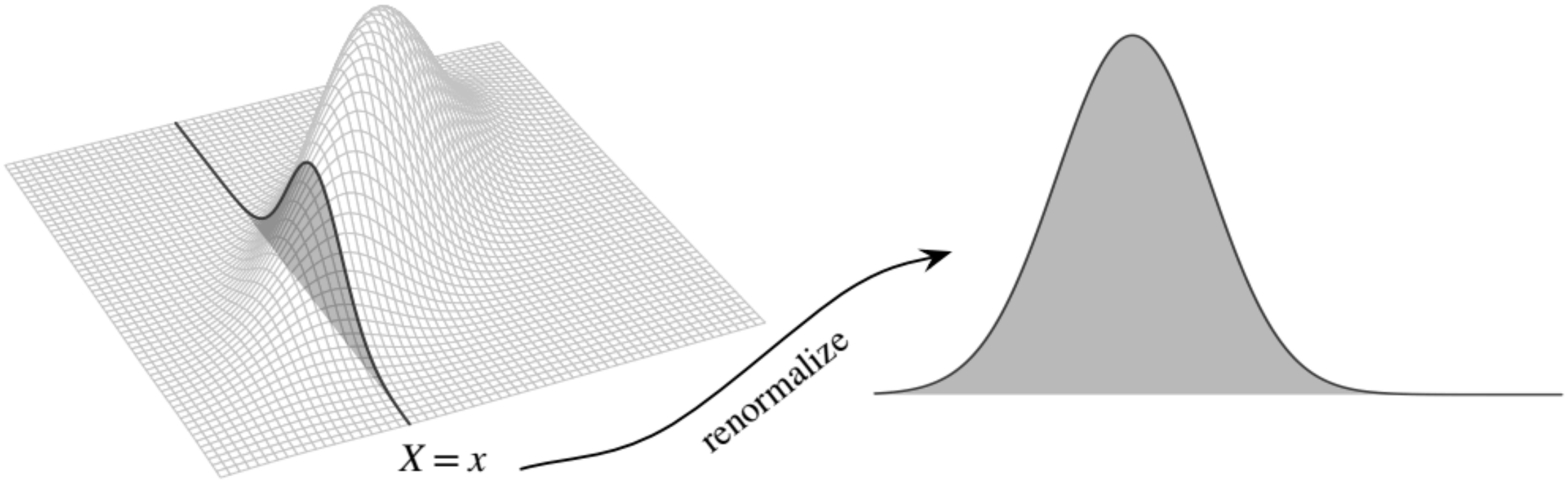


FIGURE 7.5
Conditional PDF of Y given $X = x$. The conditional PDF $f_{Y|X}(y|x)$ is obtained by renormalizing the slice of the joint PDF at the fixed value x.

Conditional PDF

Definition 7.1.14 (Conditional PDF)**.** For continuous r.v.s X and Y with joint PDF $f_{X,Y}$, the *conditional PDF* of Y given $X = x$ is

$$f_{Y|X}(y|x) = \frac{f_{X,Y}(x,y)}{f_X(x)}.$$

This is considered as a function of y for fixed x.

Theorem 7.1.17 (Continuous form of Bayes' rule and LOTP)**.** For continuous r.v.s X and Y,

$$f_{Y|X}(y|x) = \frac{f_{X|Y}(x|y)f_Y(y)}{f_X(x)},$$

$$f_X(x) = \int_{-\infty}^{\infty} f_{X|Y}(x|y)f_Y(y)dy.$$

Proof. By definition of conditional PDFs, we have

$$f_{Y|X}(y|x)f_X(x) = f_{X,Y}(x,y) = f_{X|Y}(x|y)f_Y(y).$$

The continuous version of Bayes' rule follows immediately from dividing by $f_X(x)$. The continuous version of LOTP follows immediately from integrating with respect to y:

$$f_X(x) = \int_{-\infty}^{\infty} f_{X,Y}(x,y)dy = \int_{-\infty}^{\infty} f_{X|Y}(x|y)f_Y(y)dy.$$

■

Out of curiosity, let's see what would have happened if we had plugged in the other expression for $f_{X,Y}(x,y)$ instead in the proof of LOTP:

$$f_X(x) = \int_{-\infty}^{\infty} f_{X,Y}(x,y)dy = \int_{-\infty}^{\infty} f_{Y|X}(y|x)f_X(x)dy = f_X(x)\int_{-\infty}^{\infty} f_{Y|X}(y|x)dy.$$

So this just says that

$$\int_{-\infty}^{\infty} f_{Y|X}(y|x)dy = 1,$$

confirming the fact that conditional PDFs must integrate to 1.

Probabilistic Robotics

Introduction

Probabilities

Bayes rule

Bayes filters

Discrete Random Variables

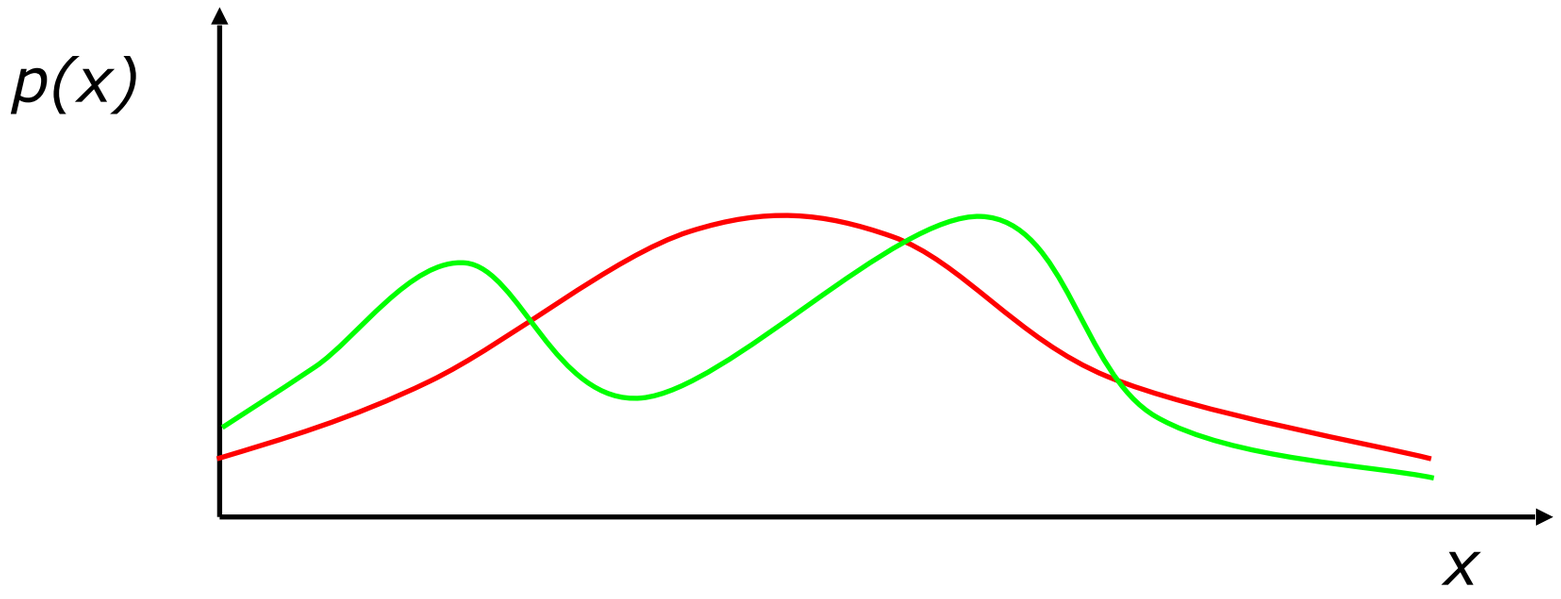
- X denotes a random variable.
- X can take on a countable number of values in $\{x_1, x_2, \ldots, x_n\}$.
- $P(X=x_i)$, or $P(x_i)$, is the probability that the random variable X takes on value x_i.
- $P(\cdot)$ is called probability mass function.
- E.g. $$P(Room) = \langle 0.7, 0.2, 0.08, 0.02 \rangle$$

Continuous Random Variables

- *X* takes on values in the continuum.
- *p(X=x)*, or *p(x)*, is a probability density function.

$$\Pr(x \in (a,b)) = \int_a^b p(x)dx$$

- E.g.



Joint and Conditional Probability

- $P(X=x \text{ and } Y=y) = P(x,y)$

- If X and Y are independent then

$$P(x,y) = P(x)\,P(y)$$

- $P(x \mid y)$ is the probability of x given y

$$P(x \mid y) = P(x,y) / P(y)$$

$$P(x,y) = P(x \mid y)\,P(y)$$

- If X and Y are independent then

$$P(x \mid y) = P(x)$$

Law of Total Probability, Marginals

Discrete case

$$\sum_x P(x) = 1$$

$$P(x) = \sum_y P(x, y)$$

$$P(x) = \sum_y P(x \mid y) P(y)$$

Continuous case

$$\int p(x)\, dx = 1$$

$$p(x) = \int p(x, y)\, dy$$

$$p(x) = \int p(x \mid y)\, p(y)\, dy$$

Bayes Formula

$$P(x, y) = P(x \mid y)P(y) = P(y \mid x)P(x)$$

$$\Rightarrow$$

$$P(x \mid y) = \frac{P(y \mid x)\ P(x)}{P(y)} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}}$$

Normalization

$$P(x|y) = \frac{P(y|x)\ P(x)}{P(y)} = \eta\ P(y|x)\,P(x)$$

$$\eta = P(y)^{-1} = \frac{1}{\sum_x P(y|x)P(x)}$$

Algorithm:

$$\forall x : \text{aux}_{x|y} = P(y|x)\ P(x)$$

$$\eta = \frac{1}{\sum_x \text{aux}_{x|y}}$$

$$\forall x : P(x|y) = \eta\ \text{aux}_{x|y}$$

Bayes Rule with Background Knowledge

$$P(x \mid y,z) = \frac{P(y \mid x,z)\ P(x \mid z)}{P(y \mid z)}$$

Conditioning

- Total probability:

$$P(x) = \int P(x,z)dz$$

$$P(x) = \int P(x \mid z)P(z)dz$$

$$P(x \mid y) = \int P(x \mid y,z)\,P(z)\,dz$$

Conditional Independence

$$P(x, y \mid z) = P(x \mid z)P(y \mid z)$$

equivalent to

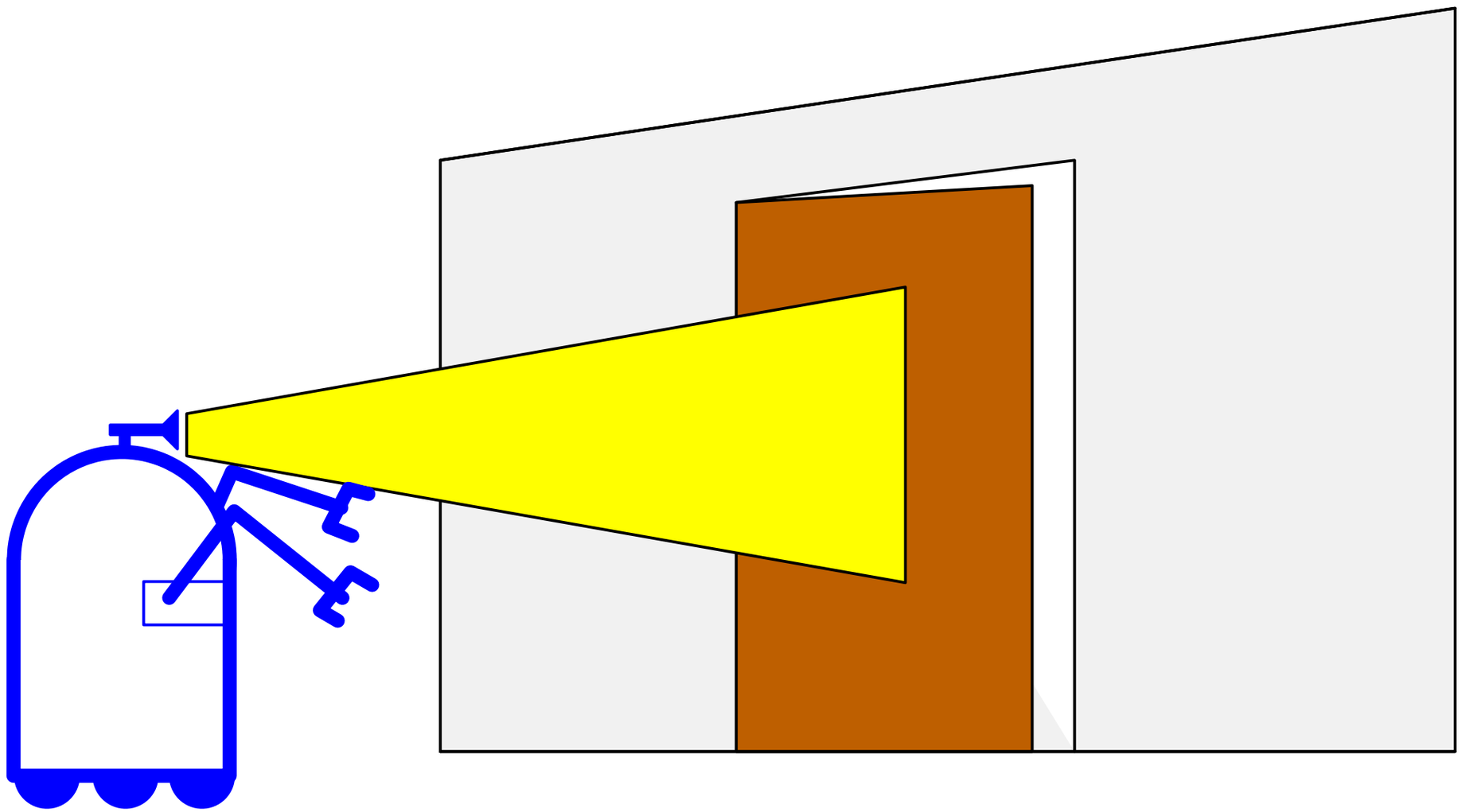
$$P(x \mid z) = P(x \mid z, y)$$

and

$$P(y \mid z) = P(y \mid z, x)$$

Simple Example of State Estimation

- Suppose a robot obtains measurement *z*
- What is *P(open|z)?*



Causal vs. Diagnostic Reasoning

- *P(open|z)* is diagnostic.
- *P(z|open)* is causal.
- Often causal knowledge is easier to obtain.

count frequencies!

- Bayes rule allows us to use causal knowledge:

$$P(open \mid z) = \frac{P(z \mid open)P(open)}{P(z)}$$

Example

- $P(z|open) = 0.6 \qquad P(z|\neg open) = 0.3$
- $P(open) = P(\neg open) = 0.5$

$$P(open \mid z) = \frac{P(z \mid open)P(open)}{P(z \mid open)p(open) + P(z \mid \neg open)p(\neg open)}$$

$$P(open \mid z) = \frac{0.6 \cdot 0.5}{0.6 \cdot 0.5 + 0.3 \cdot 0.5} = \frac{2}{3} = 0.67$$

- z raises the probability that the door is open.

Combining Evidence

- Suppose our robot obtains another observation z_2.

- How can we integrate this new information?

- More generally, how can we estimate $P(x| z_1...z_n)$?

Recursive Bayesian Updating

$$P(x \mid z_1, \ldots, z_n) = \frac{P(z_n \mid x, z_1, \ldots, z_{n-1})\, P(x \mid z_1, \ldots, z_{n-1})}{P(z_n \mid z_1, \ldots, z_{n-1})}$$

Markov assumption: z_n is independent of $z_1, \ldots, z_{n-1}$ if we know x.

$$P(x \mid z_1, \ldots, z_n) = \frac{P(z_n \mid x)\, P(x \mid z_1, \ldots, z_{n-1})}{P(z_n \mid z_1, \ldots, z_{n-1})}$$

$$= \eta\, P(z_n \mid x)\, P(x \mid z_1, \ldots, z_{n-1})$$

$$= \eta_{1 \ldots n} \prod_{i=1 \ldots n} P(z_i \mid x)\, P(x)$$

Example: Second Measurement

- $P(z_2|open) = 0.5$ $\quad\quad$ $P(z_2|\neg open) = 0.6$
- $P(open|z_1)=2/3$

$$P(open \mid z_2, z_1) = \frac{P(z_2 \mid open)\, P(open \mid z_1)}{P(z_2 \mid open)\, P(open \mid z_1) + P(z_2 \mid \neg open)\, P(\neg open \mid z_1)}$$

$$= \frac{\frac{1}{2} \cdot \frac{2}{3}}{\frac{1}{2} \cdot \frac{2}{3} + \frac{3}{5} \cdot \frac{1}{3}} = \frac{5}{8} = 0.625$$

- z_2 lowers the probability that the door is open.

Coherence of Bayes Rule

- Note that in this example it does not matter whether we consider the measurement z_1 or z_2 first. We will end up with the same result. In this sense Bayes rule is coherent since it gives the same result regardless of the order in which the information is accumulated.

- One way to see this is to observe that in the discrete case the update

$$p(x|z) = \frac{p(z|x)p(x)}{p(z)} = p(x)\frac{p(z|x)}{p(z)}$$

can be viewed as a matrix vector multiplication where $p(x)$ and $p(x|z)$ are row vectors denoting our belief about x before and after the measurement respectively and $\frac{p(z|x)}{p(z)}$ is a *diagonal* matrix which captures the measurement model and the required scaling. Since diagonal matrices commute we can apply all relevant measurements in any order and get the same result.

State Estimation

State of a system

- We will use the term state denoted by the symbol x_t to denote the aspects of the world that we want to estimate
- Some examples of state include:
  - Robot position
  - Robot velocity
  - Map of environment
  - Configuration of joints
  - Location and velocity of other cars or pedestrians
  - Air pressure outside vehicle
  - Rotational speed of propellors
- Note state is often vector-valued

Time

- Note that we use the subscript t in x_t to denote the fact that the state of the world typically evolves over time.
- Since we are typically implementing our estimation system on a digital computer time is often discretized

Measurements

- We will use the symbol z_t to denote sensor measurements that the robot acquires over time.
- Some examples of measurement include
  - GPS measurements
  - Image measurements
  - Switch readings
  - LIDAR readings
  - Outputs of a person detection system
- Sensor measurements can be vector valued and can be continuous or discrete

Measurement Model

- In the sequel we will assume that the sensor measurements we receive can be predicted from the state of the robot. More precisely we will assume that we can model the probability of a sensor output conditioned on the state of the robot

$$P(z_t|x_t)$$

- Note that this probabilistic model allows us to account for the fact that our sensor measurements may contain random errors. If there were no such errors we could simply model the measurements as a deterministic function of the state.

Actions

- We will use the symbol u_t to denote actions or control outputs that the robot emits over time
- For instance the robot may move its wheels or extend its arms or open a valve that actuates a pneumatic element

Action Model

- In the sequel we will assume that we can model the evolution of the state of the system over time using another conditional probability distribution.

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

- This model assumes that the state of the system can be predicted given the last state of the system, x_{t-1}, and the current control input, u_t. Once again this model is probabilistic to account for the fact that our model may have inaccuracies or other random aspects.

- Note that this model is Markovian, that is it assumes that the next state can be predicted solely from the current state and control without considering any previous states. This is a non-trivial assumption and you will typically need to define your state carefully to satisfy it.

State Estimation

- The goal of state estimation is to maintain a probability distribution that represents our knowledge, or lack thereof, about the state of the world conditioned on all of the measurements we have seen and all of the actions we have taken.

$$p(x_t|z_1, u_1, z_2, u_2, z_3, u_3, \cdots, z_{t-1}, u_{t-1})$$

- We will sometimes denote this distribution as our *belief* distribution

$$bel(x_t)$$

Example

- The figure to the right depicts a robot moving along a straight corridor. The state we want to estimate is its position which is discretized into a finite set of cells

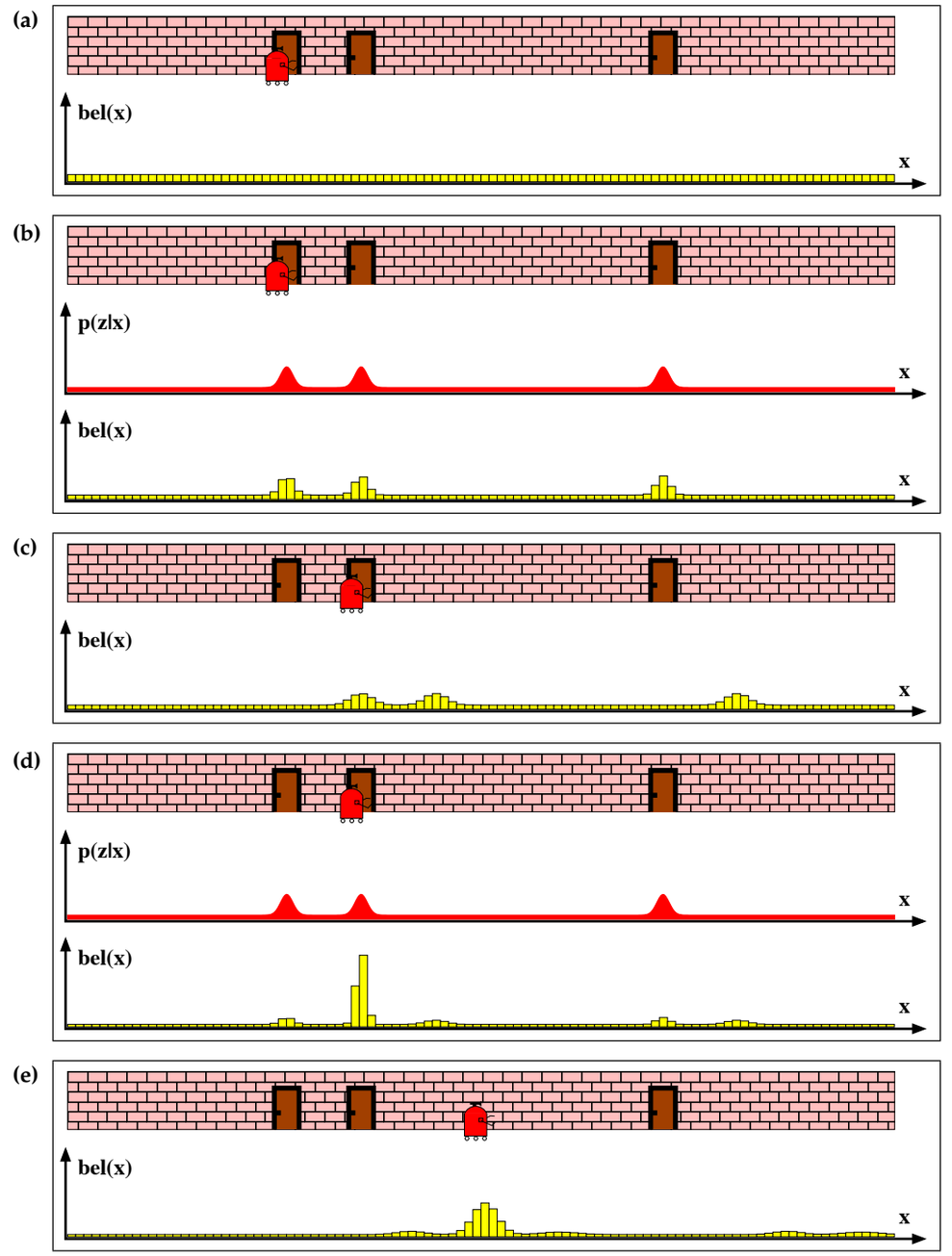


Figure 8.1 Grid localization using a fine-grained metric decomposition. Each picture depicts the position of the robot in the hallway along with its belief $bel(x_t)$, represented by a histogram over a grid.

Sensor Model

- We assume that the robot is equipped with a sensor which it uses to detect doors returning a yes or no answer, z_t. The sensor isn't entirely accurate and we capture this with the following sensor model.

$$p(z_t = 1|\text{ door}) = 0.8$$

$$p(z_t = 1|\text{ no door}) = 0.1$$

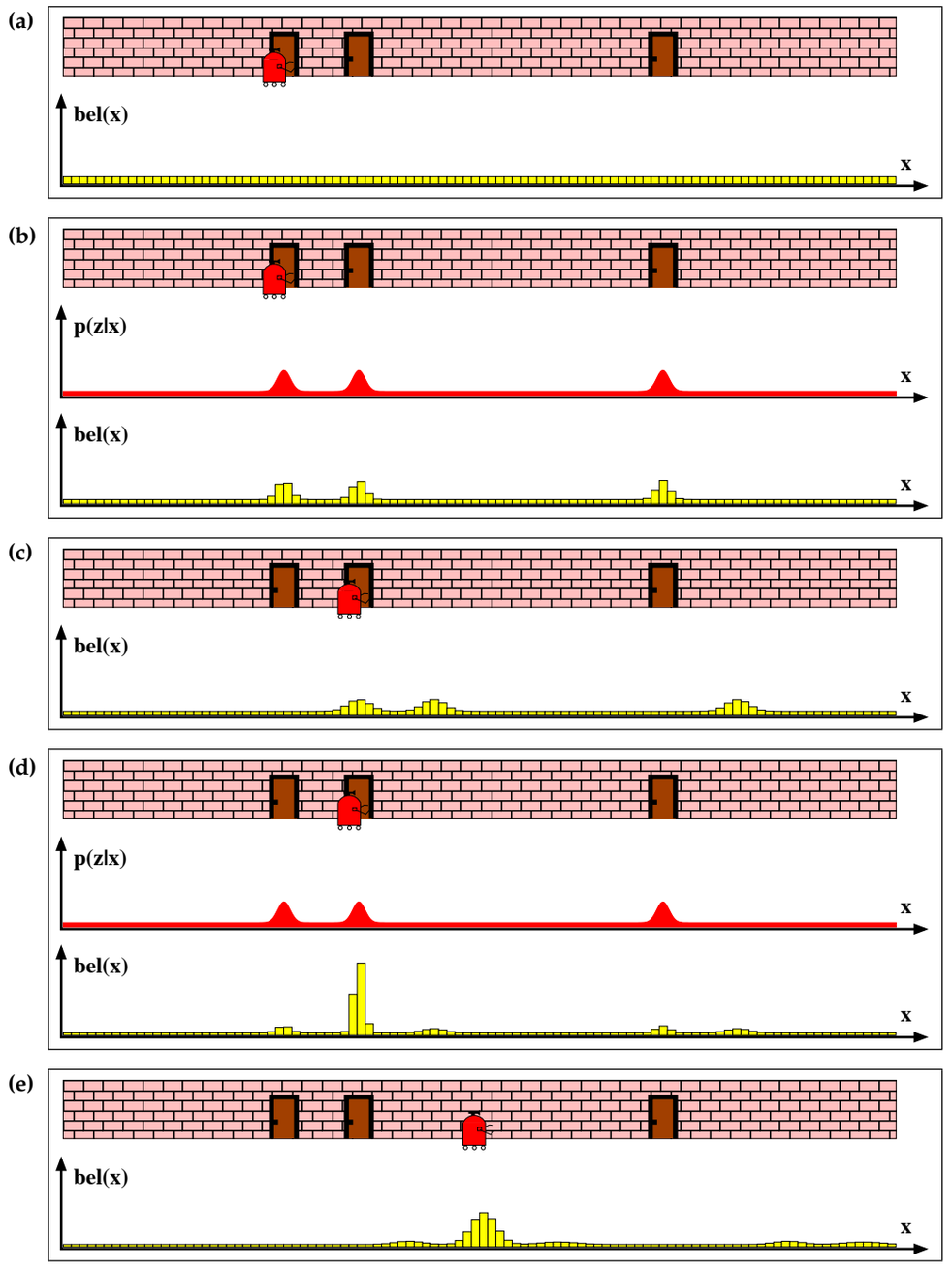


Figure 8.1 Grid localization using a fine-grained metric decomposition. Each picture depicts the position of the robot in the hallway along with its belief $bel(x_t)$, represented by a histogram over a grid.

Action Model

- At any point the robot can move left, move right or stay in the same place.
- If it chooses to move the probability that it actually moves is 0.8 otherwise it fails to move and ends up in the same place.
- In the special case where the robot is at the far left or far right and elects to move left or right respectively it will stay in the same place with probability one. You can assume that this corresponds to hitting a physical wall.

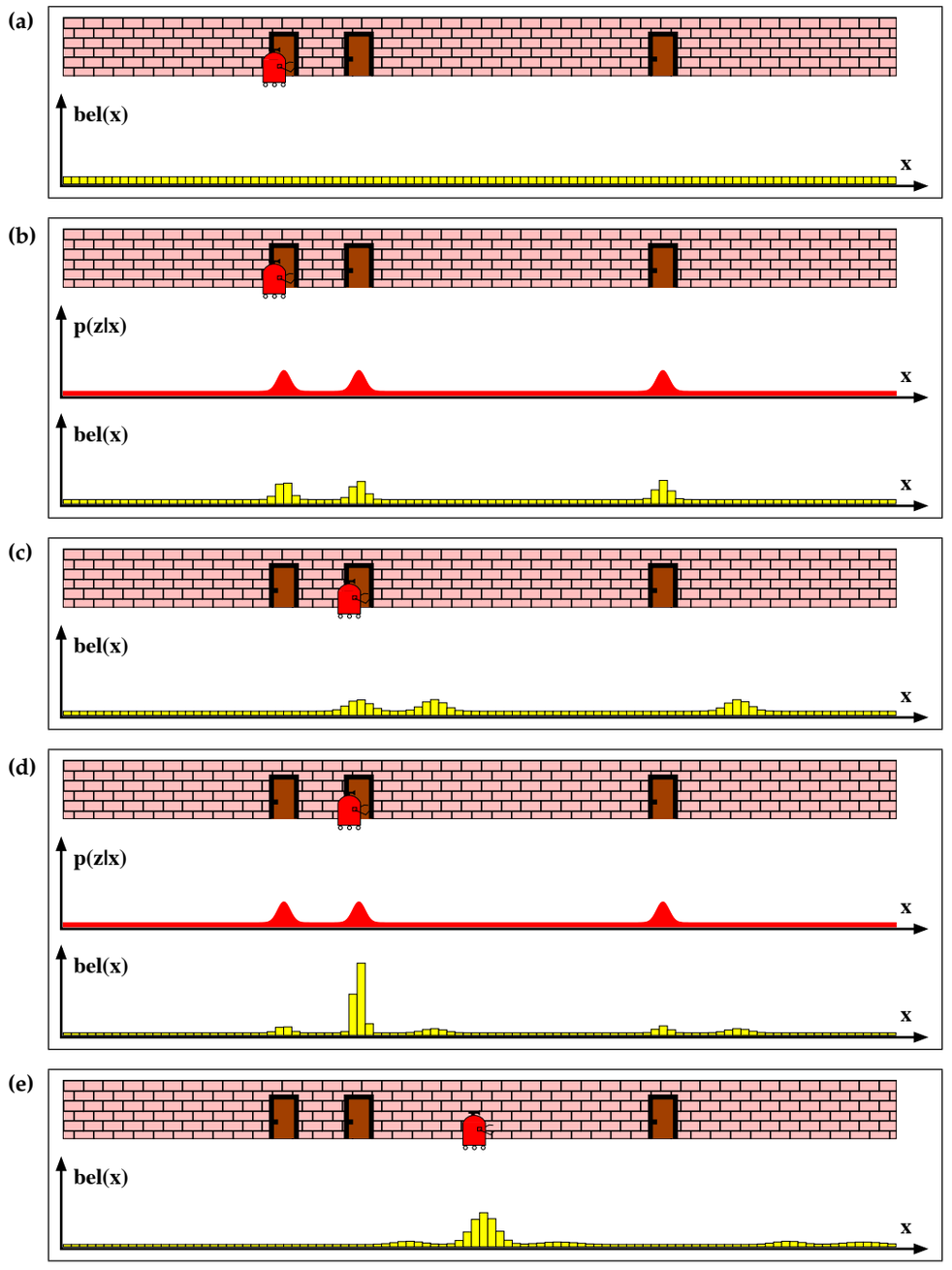


Figure 8.1 Grid localization using a fine-grained metric decomposition. Each picture depicts the position of the robot in the hallway along with its belief $bel(x_t)$, represented by a histogram over a grid.

Action Update

- Let's consider a simple example where the corridor is divided into 5 cells. In the beginning we will assume that we believe that the robot is equally likely to be in any of the 5 cells.

$$bel(x_0) = \begin{bmatrix} 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \end{bmatrix}$$

- At time 1 the robot issues a command to move one step right. We can compute our new belief state as follows:

$$\begin{aligned}
bel(x_1|x_0, u_1) &= 0.2 \times \begin{bmatrix} 0.2 & 0.8 & 0 & 0 & 0 \end{bmatrix} + && (1)\\
&\quad 0.2 \times \begin{bmatrix} 0 & 0.2 & 0.8 & 0 & 0 \end{bmatrix} + && (2)\\
&\quad 0.2 \times \begin{bmatrix} 0 & 0 & 0.2 & 0.8 & 0 \end{bmatrix} + && (3)\\
&\quad 0.2 \times \begin{bmatrix} 0 & 0 & 0 & 0.2 & 0.8 \end{bmatrix} + && (4)\\
&\quad 0.2 \times \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix} && (5)\\
&= \begin{bmatrix} 0.04 & 0.2 & 0.2 & 0.2 & 0.36 \end{bmatrix} && (6)
\end{aligned}$$

- Note this is an application of the Law of Total Probability

Markov Matrix

- Note that this update step can be viewed as a matrix vector multiplication where $bel(x_0)$ is a row vector denoting the current belief state which we multiply by a matrix representing the state transition probabilities of the Markov process, $p(x_1|x_0, u_1)$.

$$bel(x_1) = bel(x_0)p(x_1|x_0, u_1)$$

$$bel(x_1) = bel(x_0)M_{01}$$

$$M_{01} = \begin{pmatrix} 0.2 & 0.8 & 0 & 0 & 0 \\ 0 & 0.2 & 0.8 & 0 & 0 \\ 0 & 0 & 0.2 & 0.8 & 0 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

- The matrix M_{01} is a Markov matrix that captures the transition probabilities between time steps 0 and 1. Note that the sum of the entries along each row is 1.

- We can imagine a sequence of transition matrices that represent the phases of a Discrete Markov Process, $M_{01}, M_{12}, M_{23}, M_{34}...$ and can use these to model the evolution of the belief state as follow:

$$bel(x_4) = bel(x_0)M_{01}M_{12}M_{23}M_{34}$$

Sensor Update

- Let's assume that in this corridor there are doors at the first position and the fourth position.
- Let's consider the case where the belief state at some time t is given by

$$bel(x_t) = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.2 & 0.2 \end{bmatrix}$$

- Let's assume that at time t we take a measurement, z_t, which reports a 1 indicating the possible presence of a door.
- We can use Bayes Rule to update our belief given this new measurement.

$$p(x|z) = \frac{p(z|x)p(x)}{p(z)}$$

$$p(x|z) \propto p(z|x)p(x)$$

- Given the measurement we update our belief state as follows

$$\begin{aligned} \bar{bel}(x_t|z_t) &\propto \begin{bmatrix} 0.8 \times 0.1 & 0.1 \times 0.2 & 0.1 \times 0.3 & 0.8 \times 0.2 & 0.1 \times 0.2 \end{bmatrix} \\ &\propto \begin{bmatrix} 0.08 & 0.02 & 0.03 & 0.16 & 0.02 \end{bmatrix} \end{aligned}$$

- As a final step we can normalize the array so that the entries sum to 1.

Coherence of Bayes Rule

- Another way to look at this is to observe that in the discrete case the update

$$p(x|z) = \frac{p(z|x)p(x)}{p(z)} = p(x)\frac{p(z|x)}{p(z)}$$

can be viewed as a matrix vector multiplication where $p(x)$ and $p(x|z)$ are row vectors denoting our belief about x before and after the measurement respectively and $\frac{p(z|x)}{p(z)}$ is a *diagonal* matrix which captures the measurement model and the required scaling. Since diagonal matrices commute we can apply all relevant measurements in any order and get the same result. This explains the coherence of Bayes rule.

Bayes Filters: Framework

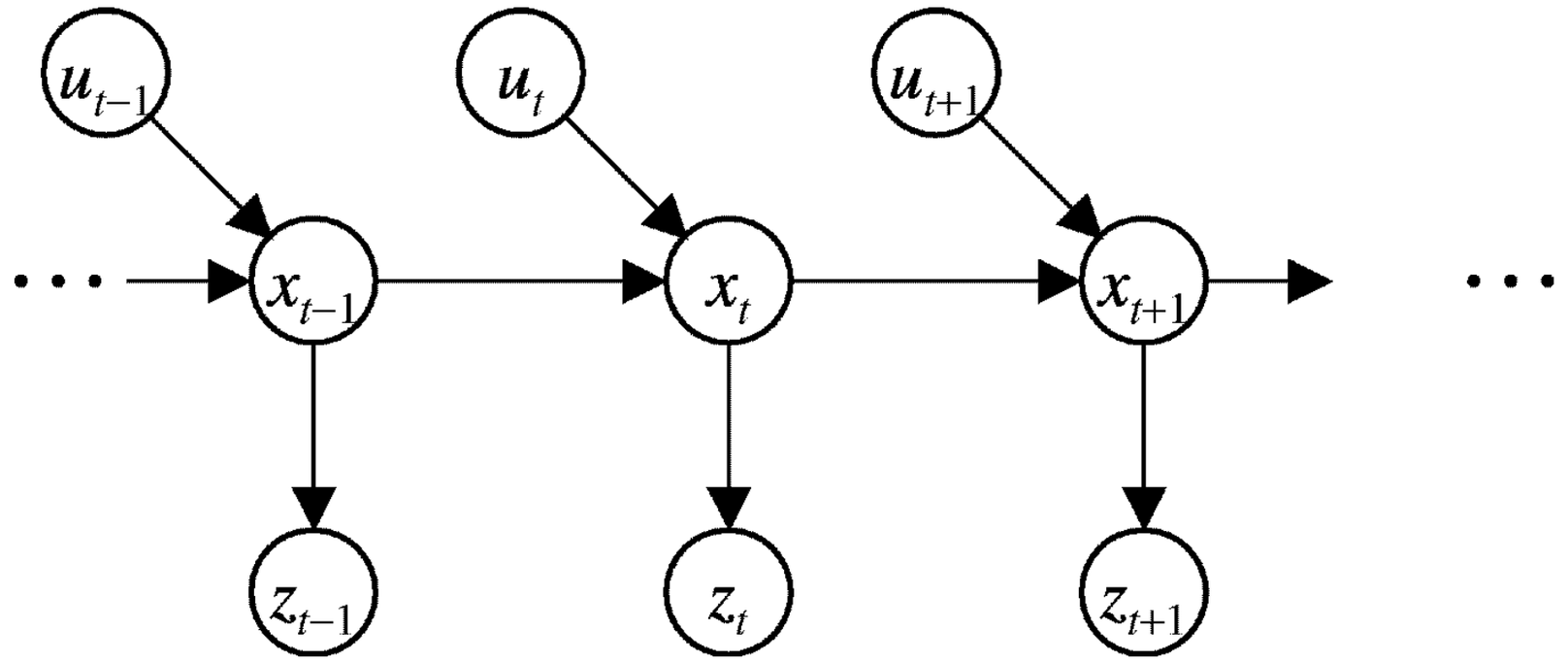
- **Given:**
  - Stream of observations *z* and action data *u:*

$$d_t = \{u_1, z_1 \ldots, u_t, z_t\}$$

  - Sensor model *P(z|x).*
  - Action model *P(x|u,x')*.
  - Prior probability of the system state *P(x).*
- **Wanted:**
  - Estimate of the state *X* of a dynamical system.
  - The posterior of the state is also called **Belief**:

$$Bel(x_t) = P(x_t \mid u_1, z_1 \ldots, u_t, z_t)$$

Markov Assumption



$$p(z_t \mid x_{0:t}, z_{1:t}, u_{1:t}) = p(z_t \mid x_t)$$
$$p(x_t \mid x_{1:t-1}, z_{1:t}, u_{1:t}) = p(x_t \mid x_{t-1}, u_t)$$

Underlying Assumptions

- Static world
- Independent noise
- Perfect model, no approximation errors

$$Bel(x_t) = \eta\ P(z_t \mid x_t) \int P(x_t \mid u_t, x_{t-1})\ Bel(x_{t-1})\ dx_{t-1}$$

1. Algorithm **Bayes_filter**(*Bel(x),d*):
2. $\eta=0$
3. If *d* is a perceptual data item *z* then
4. For all *x* do
5. $Bel'(x) = P(z \mid x)Bel(x)$
6. $\eta = \eta + Bel'(x)$
7. For all *x* do
8. $Bel'(x) = \eta^{-1} Bel'(x)$
9. Else if *d* is an action data item *u* then
10. For all *x* do
11. $Bel'(x) = \int P(x \mid u, x')\ Bel(x')\ dx'$
12. Return *Bel'(x)*

Summary

- Bayes rule allows us to compute probabilities that are hard to assess otherwise.
- Under the Markov assumption, recursive Bayesian updating can be used to efficiently combine evidence.
- Bayes filters are a probabilistic tool for estimating the state of dynamic systems.

Comments

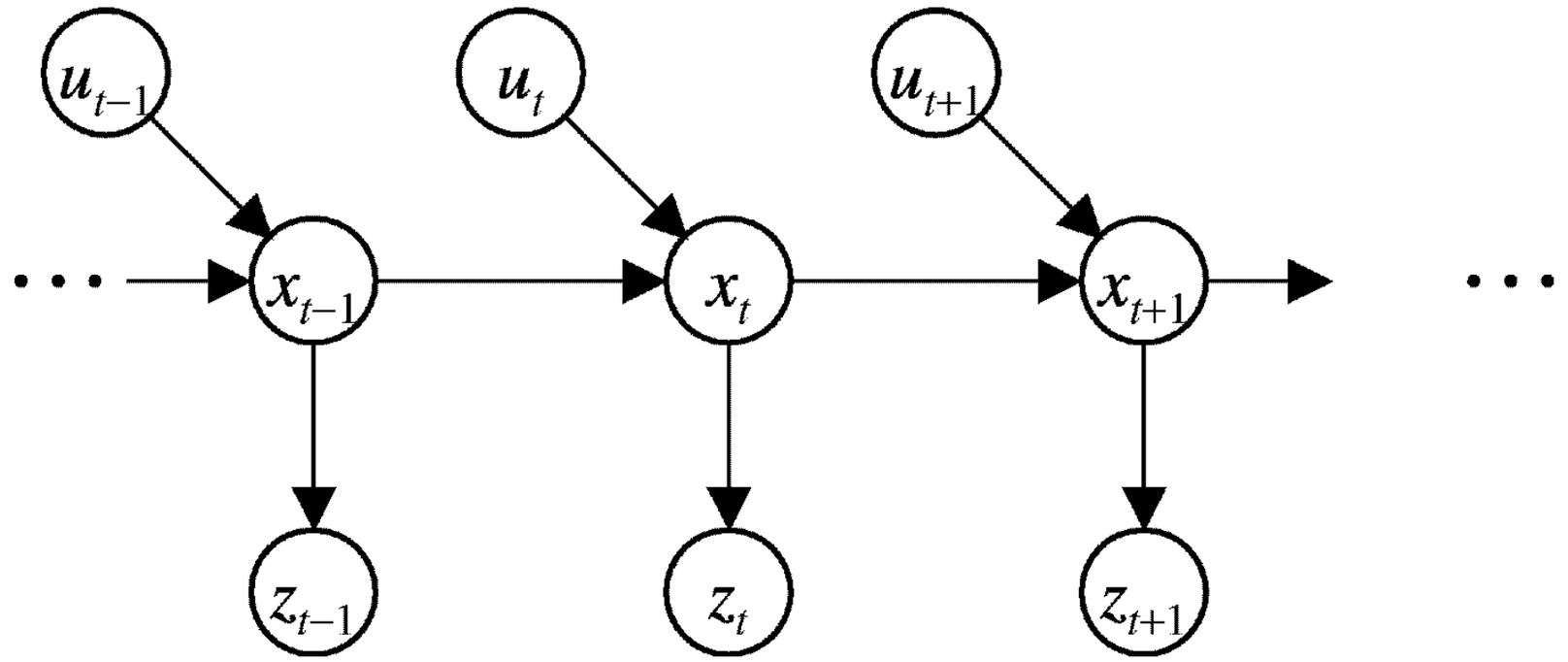
- This implementation of a Bayes filter represents the probability distribution over the state with a histogram
- This scales badly as we increase the number of dimensions in the state space
- Other implementations of the Bayes Filter idea that we will consider will use different representations of the PDF with different strengths and weaknesses

Bayes Filters are Familiar!

$$Bel(x_t) = \eta\ P(z_t \mid x_t) \int P(x_t \mid u_t, x_{t-1})\ Bel(x_{t-1})\ dx_{t-1}$$

- Kalman filters
- Particle filters
- Hidden Markov models
- Dynamic Bayesian networks
- Partially Observable Markov Decision Processes (POMDPs)

Hidden Markov Model



$$p(z_t \mid x_{0:t}, z_{1:t}, u_{1:t}) = p(z_t \mid x_t)$$
$$p(x_t \mid x_{1:t-1}, z_{1:t}, u_{1:t}) = p(x_t \mid x_{t-1}, u_t)$$

- This generic model captures the idea of a hidden state, x, indirectly observed through a set of output measurements, z.
- State evolves in a Markovian fashion

Hidden Markov Models

- Given a hidden markov model consider the problem of finding a sequence of states that are in best agreement with a set of observations.

$$\max_{x_1,x_2,x_3} p(x_1, x_2, x_3 | z_1, z_2, z_3)$$

- Note that because of Bayes rule and the structure of the Markov Model we can rewrite as follows

$$p(x_1, x_2, x_3 | z_1, z_2, z_3) = \frac{p(z_1|x_1)p(z_2|x_2)p(z_3|x_3)p(x_1, x_2, x_3)}{p(z_1, z_2, z_3)}$$

- Further

$$p(x_1, x_2, x_3) = p(x_0)p(x_1|x_0)p(x_2|x_1)p(x_3|x_2)$$

- Putting it all together

$$p(x_1, x_2, x_3 | z_1, z_2, z_3) = \frac{p(z_1|x_1)p(z_2|x_2)p(z_3|x_3)p(x_0)p(x_1|x_0)p(x_2|x_1)p(x_3|x_2)}{p(z_1, z_2, z_3)}$$

Example

- Consider a markov model of the weather where the world has three states, sunny, cloudy and rainy. Suppose that the following matrix captures the transitions between those three states:

$$\begin{pmatrix} 0.5 & 0.2 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.3 & 0.6 & 0.1 \end{pmatrix}$$

  egs. if the current day is sunny the probability that the next day will be cloudy is 0.2 according to this table.

- Suppose we have an indirect measurement of the weather in terms of elevated umbrella sales. The probability of elevated umbrella sales given the that the weather is sunny, cloudy or rainy are 0.1, 0.4 and 0.8 respectively.

- Given a prior distribution of the weather on day 0, (0.3, 0.2, 0.5), and a sequence of umbrella readings (elevated, normal, elevated), what is the most likely sequence of weather states over the three days?

Naive Approach

- One naive approach to solving this problem would be to enumerate all possible sequences of weather conditions for the three days, egs. (sunny, rainy, cloudy) and then compute the associated probability, $p(x_1, x_2, x_3|z_1, z_2, z_3)$. Then we could determine which sequence is most probable.

- The problem with this approach is that it scales poorly. If there are N possible states and we are considering T time steps we would have to compute N^T probabilities.

Viterbi Algorithm

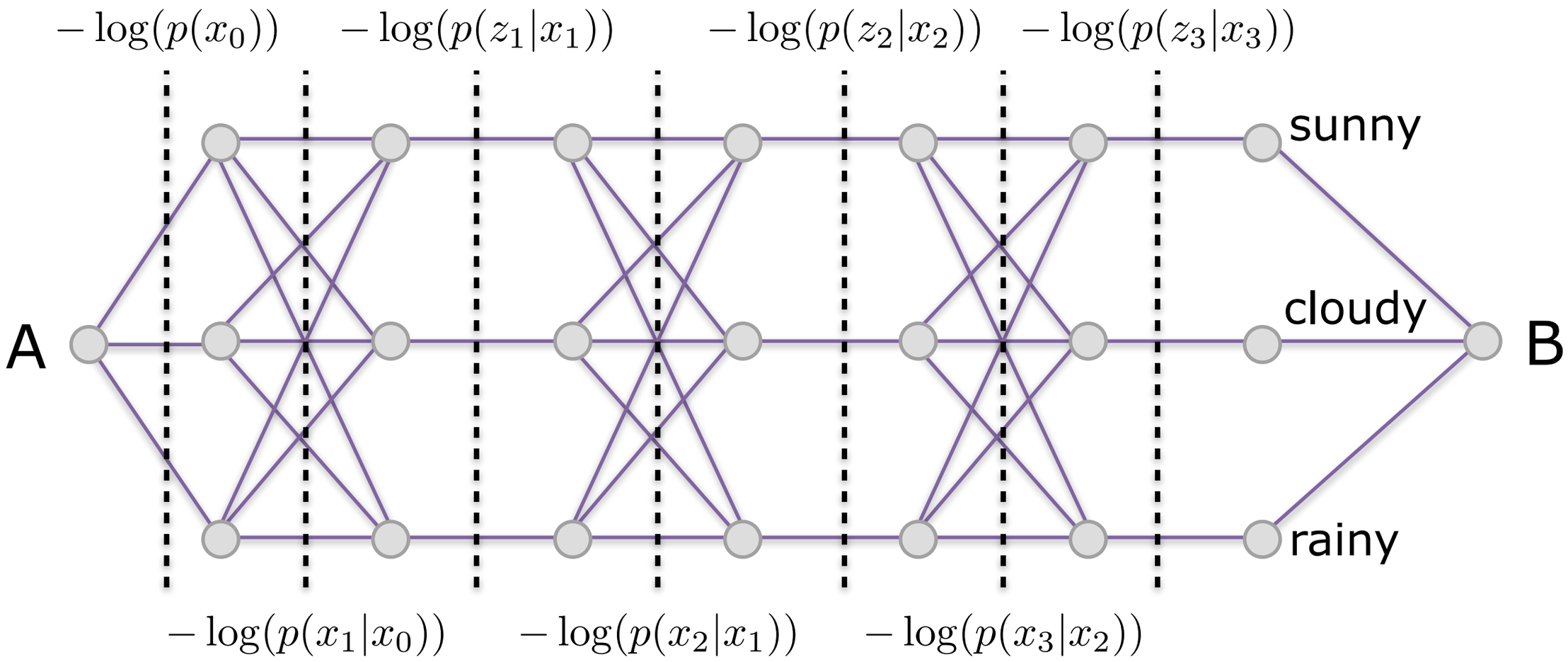
- There is a more efficient way to tackle this problem which leverages the structure of the HMM.
- One way to see this is by taking the log of the probability we are trying to maximize:

$$p(x_1, x_2, x_3|z_1, z_2, z_3) = \frac{p(z_1|x_1)p(z_2|x_2)p(z_3|x_3)p(x_0)p(x_1|x_0)p(x_2|x_1)p(x_3|x_2)}{p(z_1, z_2, z_3)}$$

$$\begin{aligned} \log p(x_1, x_2, x_3|z_1, z_2, z_3) &= \log p(z_1|x_1) + \log p(z_2|x_2) + \log p(z_3|x_3) + \log p(x_0) + \\ &\quad \log p(x_1|x_0) + \log p(x_2|x_1) + \log p(x_3|x_2) \\ &\quad - \log p(z_1, z_2, z_3) \end{aligned} \tag{1}$$

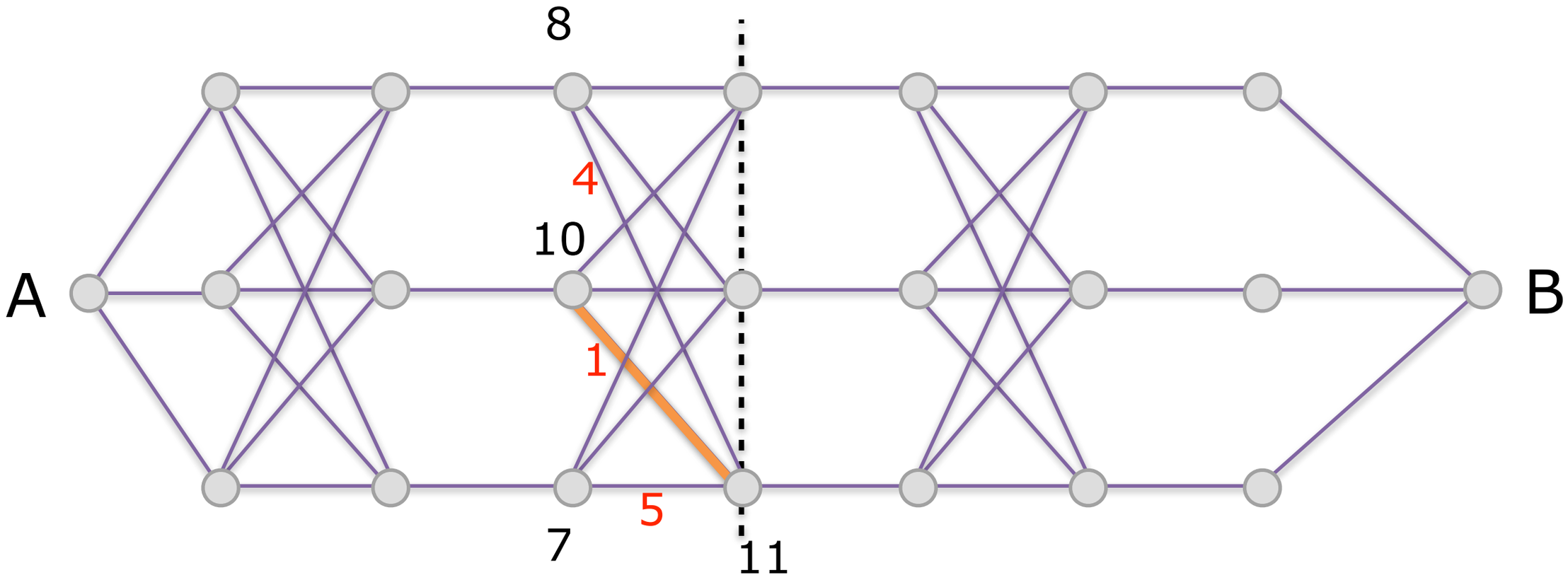
- To maximize the probability we can *minimize* $-\log p(x_1, x_2, x_3|z_1, z_2, z_3)$. Since the probabilities are all between 0 and 1 the negative logs will be non-negative values.
- Note that the term $\log p(z_1, z_2, z_3)$ is a constant given the observed measurements so it doesn't affect the optimization

Viterbi Algorithm



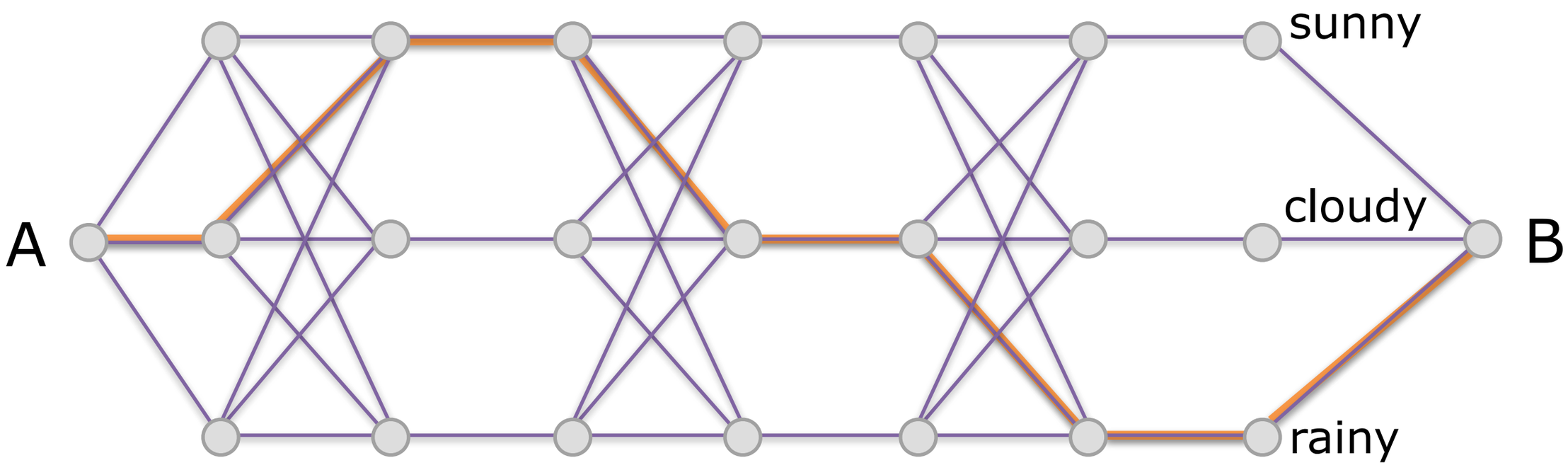
- This diagram shows how we can recast the optimization problem as one of finding the shortest path between 2 points, A and B, in a trellis graph.
- At each stage of the trellis graph the edge weights model one aspect of the sum being minimized, $-\log(p(x_0))$, $-\log p(x_1|x_0)$, $-\log p(z_1|x_1)$ etc.
- In this diagram the edge weights in the last phase linking to node B are all 0.

Viterbi Algorithm



- The optimization problem can be solved efficiently by considering each stage in the trellis in turn from left to right. At each stage each node computes the shortest distance to the start node by considering the shortest distances in the previous stage. We need only record the shortest path distance for each node and its parent in the previous stage.
- When the algorithm reaches the final stage we can trace back through the parent pointers to recover the optimal path/state assignment.
- This is an example of *Dynamic Programming* where our optimization is split over a number of stages and the partial results from one stage of the optimization are used in the next.

Viterbi Algorithm



- When the algorithm reaches the final stage we can trace back through the parent pointers to recover the optimal path/state assignment.
- If there are N possible states and T time steps the overall computational complexity is N^2T since each of the T stages considers N^2 possibilities.

ESE 650
Learning in Robotics

Spring 2019

Kalman Filter

- Goal to introduce the Kalman Filter for state estimation
- Much like the discrete Bayes filter from the previous lecture the goal here is to maintain an estimate for the state of a system as we perform actions and take measurements
- The representation of probability will be different

Linearity of Expectation

- If X and Y are random variables then

$$\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

  This is actually more surprising than it looks because it is true regardless of the structure of the joint probability distribution function $P(X,Y)$.

- If $Z = X + Y$ then:

$$\begin{aligned}
\mathbb{E}[(Z - \mathbb{E}(Z))^2] &= \mathbb{E}[((X+Y) - (\mathbb{E}(X) + \mathbb{E}(Y)))^2] \\
&= \mathbb{E}[((X - \mathbb{E}(X)) + (Y - \mathbb{E}(Y)))^2] \\
&= \mathbb{E}[(X - \mathbb{E}(X))^2] + \mathbb{E}[(Y - \mathbb{E}(Y))^2] + 2\mathbb{E}[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y))] \\
&= \sigma_x^2 + \sigma_y^2 + 2\sigma_{xy}
\end{aligned}$$

Linearity of Expectation

- Let X and Y denote random variables with joint probability distribution function f_{xy}. The probability distribution function of the sum $Z = X + Y$ is given by.

$$f_z(z) = f_z(z) = \int_{-\infty}^{\infty} f_{xy}(z-t,t)dt$$

- In the special case that X and Y are independent we have $f_{xy}(x,y) = f_x(x)f_y(y)$ where f_x and f_y are the marginal distributions. This yields the convolution result shown below:

$$f_z(z) = \int_{-\infty}^{\infty} f_x(z-t)f_y(t)dt$$

Relevant Inequalities

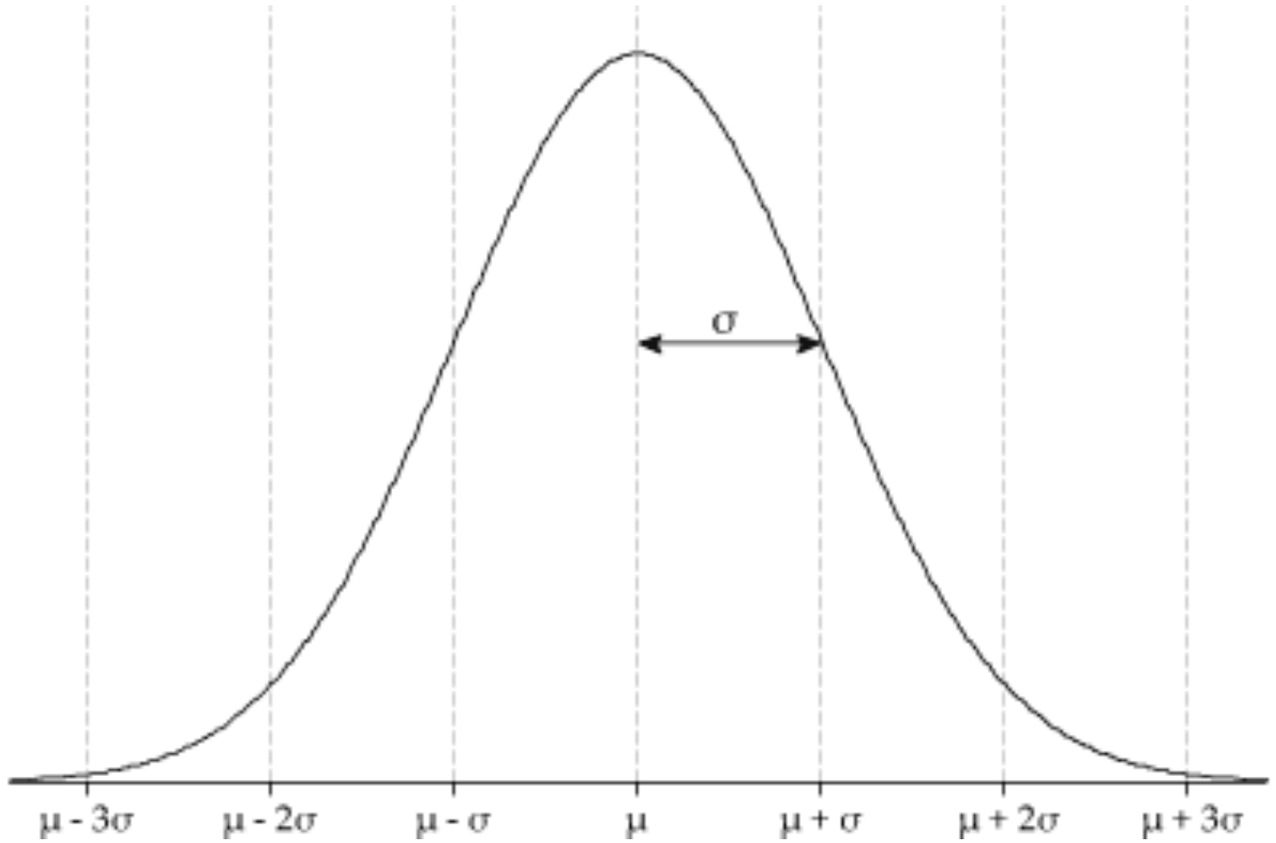
- Chebyshev's inequality

$$P(|(X - \mu)| \geq a) \leq \frac{\sigma^2}{a^2}$$

This gives us a bound on how likely the value of a random variable is to stray from the mean.

Normal Distribution 1D

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



Summing Normal RVs

- If X and Y are normally distributed random variables then their sum $Z = X + Y$ will also be normally distributed with mean

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

and variance

$$\sigma_z^2 = \sigma_x^2 + \sigma_y^2 + 2\sigma_{xy}$$

Covariance Matrices

- Given a vector valued random variable $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$. The covariance matrix associated with the distribution is defined as follows.

$$\Sigma = \mathbb{E}[(\mathbf{x} - \mu)(\mathbf{x} - \mu)^{\mathbf{T}}]$$

  where $\mu = \mathbb{E}(\mathbf{x})$.

- Note that by construction Σ is a symmetric, positive semi definite matrix which means it can be factored as follows:

$$\Sigma = \theta \Lambda \theta^T$$

  Where $\theta \in \mathbb{R}^{n \times n}$ is an orthonormal matrix of eigenvectors $\theta^T \theta = I$ and Λ is a diagonal matrix with non-negative entries.

- Note that the trace of the matrix $\text{tr}(\Sigma)$ represents the sum of the eigenvalues and reflects the total amount of uncertainty in the distribution

Matrix Trace

- The trace of a square matrix, $A \in \mathbb{R}^{n \times n}$, $\mathrm{tr}\,(A)$, is simply the sum of the diagonal entries.
- The trace of a square matrix computes the sum of the eigenvectors of the matrix
- If A and B are matrices then

$$\mathrm{tr}\,(AB) = \mathrm{tr}\,(BA)$$

  Note that A and B don't need to be square, only their product does, and in general $AB \neq BA$.

- If $P, Q \in \mathbb{R}^{m \times n}$ then

$$\mathrm{tr}(P^T Q) = \mathrm{tr}(Q^T P) = \sum_{i=1}^{m} \sum_{j=1}^{n} Q_{ij} P_{ij}$$

  so this operation can be thought of as taking the inner product of two compatible matrices.

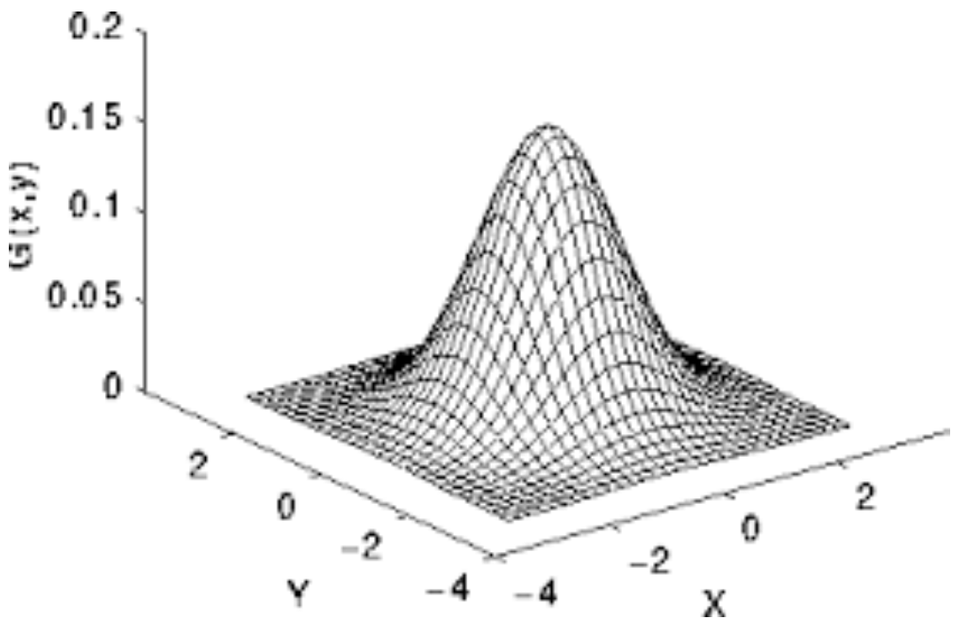
Normal Distribution

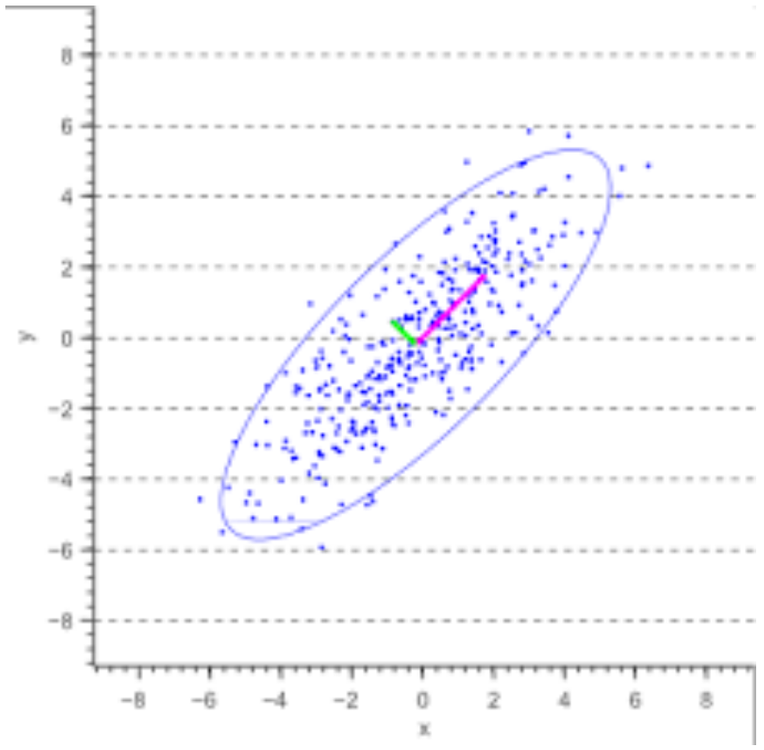
- The multivariate normal distribution is defined by

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

  where μ and Σ denote the mean and covariance respectively

- The isocontours of the PDF correspond to ellipses where the center corresponds to μ and the axes and dimensions of the ellipse are defined by the eigenvalues and eigenvectors of Σ





Summing Normal RVs

- If X and Y are normally distributed random variables then their sum $Z = X + Y$ will also be normally distributed with mean

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

  and variance

$$\sigma_z^2 = \sigma_x^2 + \sigma_y^2 + 2\sigma_{xy}$$

- if X and Y are vector valued then the covariance matrix of Z is computed as follows:

$$\Sigma_z = \Sigma_x + \Sigma_y + \Sigma_{xy} + \Sigma_{yx}$$

- In the special case where X and Y are independent and, hence, uncorrelated the variances simplify as follows:

$$\sigma_z^2 = \sigma_x^2 + \sigma_y^2$$

$$\Sigma_z = \Sigma_x + \Sigma_y$$

Linear functions of RVs

- If X is a vector valued, normally distributed random variable in $\mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$ is a constant matrix then the random variable $Y = AX$ is also a normally distributed random variable in $\mathbb{R}^m$ with mean as follows:

$$\mu_y = \mathbb{E}(Y) = \mathbb{E}(AX) = A\mathbb{E}(X) = A\mu_x$$

- The covariance of Y can be computed as follows:

$$\begin{aligned}
\Sigma_y &= \mathbb{E}[(Y - \mu_y)(Y - \mu_y)^T] \\
&= \mathbb{E}[(AX - A\mu_x)(AX - A\mu_x)^T] \\
&= \mathbb{E}[(A(X - \mu_X))(A(X - \mu_X))^T] \\
&= \mathbb{E}[A(X - \mu_X)(X - \mu_X)^T A^T] \\
&= A\mathbb{E}[(X - \mu_X)(X - \mu_X)^T]A^T \\
&= A\Sigma_x A^T
\end{aligned}$$

Estimators

- Let $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$ denote the true state of a system.
- Let $\hat{\mathbf{x}} \in \mathbb{R}^n$ denote a vector valued random variable which we term an estimator for the state.
- We say that the estimator is unbiased if $\mathbb{E}[\hat{\mathbf{x}}] = \mathbf{x}$ which is what we usually want.
- $\tilde{\mathbf{x}} = \hat{\mathbf{x}} - \mathbf{x}$ denotes the error in the estimator.
- $\Sigma_{\hat{\mathbf{x}}}$ denotes the variance of the estimator

Minimum Variance Linear Estimator

- Let's imagine we have two scalar estimators, $\hat{x}_1$ and $\hat{x}_2$, for the same state variable x. Let's further assume that $\hat{x}_1$ and $\hat{x}_2$ are conditionally independent given x

- Our goal is to form a new unbiased estimator as a linear combination of the original 2 that has the minimum variance.

$$\hat{x}_3 = k_1\hat{x}_1 + k_2\hat{x}_2$$

$$\mathbb{E}[k_1\hat{x}_1 + k_2\hat{x}_2] = x \Rightarrow k_1 + k_2 = 1$$

- Computing the variance of $\hat{x}_3$

$$\sigma_3^2 = k_1^2\sigma_1^2 + k_2^2\sigma_2^2$$

$$\sigma_3^2 = k_1^2\sigma_1^2 + (1-k_1)^2\sigma_2^2$$

- Minimizing variance with respect to k_1 yields the following optimal combination

$$\hat{x}_3 = \frac{\sigma_2^2}{\sigma_1^2+\sigma_2^2}\hat{x}_1 + \frac{\sigma_1^2}{\sigma_1^2+\sigma_2^2}\hat{x}_2$$

$$\sigma_3^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2+\sigma_2^2}$$

- Note that the variance of the new estimator is less than the variance of the original two estimators.

Minimum Variance Linear Estimator

- Let's imagine we have two vector estimators, $\hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}_2$, for the same state variable $\mathbf{x}$. Let's further assume that $\hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}_2$ are conditionally independent given $\mathbf{x}$
- Our goal is to form a new unbiased estimator as a linear combination of the original 2 that has the minimum variance.

$$\hat{\mathbf{x}}_3 = K_1\hat{\mathbf{x}}_1 + K_2\hat{\mathbf{x}}_2$$

$$\mathbb{E}[\hat{\mathbf{x}}_3] = K_1\hat{\mathbf{x}}_1 + K_2\hat{\mathbf{x}}_2 = \mathbf{x} \Rightarrow \mathbf{K_1} + \mathbf{K_2} = \mathbf{I}$$

- Computing the variance of $\hat{\mathbf{x}}_3$

$$\Sigma_3 = K_1\Sigma_1K_1^T + K_2\Sigma_2K_2^T$$

$$\Sigma_3 = K_1\Sigma_1K_1^T + (I - K_1)\Sigma_2(I - K_1)^T$$

Minimum Variance Linear Estimator

- Consider the scalar valued function $\mathrm{tr}\,(ABA^T)$ where $B^T = B$. Taking derivatives with respect to the matrix A yields.

$$\frac{\partial}{\partial A}\,\mathrm{tr}\,(ABA^T) = 2AB$$

- Computing the variance of $\hat{\mathbf{x}}_3$

$$\Sigma_3 = K_1\Sigma_1 K_1^T + K_2\Sigma_2 K_2^T$$

$$\Sigma_3 = K_1\Sigma_1 K_1^T + (I - K_1)\Sigma_2 (I - K_1)^T$$

- Minimizing $\mathrm{tr}(\Sigma_3)$ with respect to by setting $\frac{\partial}{\partial K_1}\Sigma_3 = 0$ yields

$$K_1\Sigma_1 - (I - K_1)\Sigma_2 = 0$$

$$K_1 = \Sigma_2(\Sigma_1 + \Sigma_2)^{-1} \quad K_2 = \Sigma_1(\Sigma_1 + \Sigma_2)^{-1}$$

- So the optimal linear estimator is.

$$\mathbf{x}_3 = \Sigma_2(\Sigma_1 + \Sigma_2)^{-1}\mathbf{x}_1 + \Sigma_1(\Sigma_1 + \Sigma_2)^{-1}\mathbf{x}_2$$

Measurement System

- Now we will consider the case where we have a measurement $\mathbf{z}$ that is related to the state of the system $\mathbf{x}$ as follows:

$$\mathbf{z} = C\mathbf{x} + \delta$$

  where $\delta : \mathcal{N}(0, Q)$, that is it is normally distributed with zero mean and a covariance of Q.

- Note that in general the measurement vector $\mathbf{z}$ is not the same dimension as the state vector $\mathbf{x}$

Combining Information

- Given that we have an existing estimate for the state of the system $\hat{\mathbf{x}}'$ with mean $\mathbf{x}$ and covariance Σ' and a measurement vector, $\mathbf{z} = C\mathbf{x} + \delta$, what is the best way to combine them linearly to make a new estimator $\hat{\mathbf{x}}$?

- Again we will assume a linear combination

$$\hat{\mathbf{x}} = K'\hat{\mathbf{x}}' + K\mathbf{z}$$

- We want the estimator to be unbiased so we want $\mathbb{E}(\hat{\mathbf{x}}) = \mathbf{x}$.

$$\begin{aligned}
\mathbb{E}(\hat{\mathbf{x}}) &= \mathbf{x} \\
\mathbb{E}(K'\hat{\mathbf{x}}' + K(C\mathbf{x} + \delta)) &= \mathbf{x} \\
(K' + KC)\mathbf{x} &= \mathbf{x} \\
(K' + KC) &= I \\
K' &= I - KC
\end{aligned} \tag{1}$$

Combining Information

- Our best linear estimator has the form

$$\hat{\mathbf{x}} = (I - KC)\hat{\mathbf{x}}' + K\mathbf{z} = \hat{\mathbf{x}}' + K(z - C\hat{\mathbf{x}}')$$

  We now want to find the optimal value of K which is the value that minimizes the trace of the covariance of the resulting estimator

- In computing the covariance of $\hat{\mathbf{x}}$ we will assume that $\hat{\mathbf{x}}'$ and $\mathbf{z}$ are independent. This means that the covariance of the sum is just the sum of the covariances.

$$\Sigma = (I - KC)\Sigma'(I - KC)^T + KQK^T$$

- Optimizing the trace of Σ with respect to K

$$\begin{aligned}
\frac{\partial}{\partial K} tr(\Sigma) &= 0 \\
-2(I - KC)\Sigma' C^T + 2KQ &= 0 \\
K(C\Sigma' C^T + Q) &= \Sigma' C^T \\
K &= \Sigma' C^T (C\Sigma' C^T + Q)^{-1}
\end{aligned}$$

Kalman Gain Matrix

- This expression yields the Kalman gain matrix which is the optimal form of this linear filter.

$$K = \Sigma' C^T (C\Sigma' C^T + Q)^{-1}$$

$$\hat{\mathbf{x}} = \hat{\mathbf{x}}' + K(z - C\hat{\mathbf{x}}')$$

- This expression gives us the optimal way to combine a measurement with our existing state estimate when the measurement is a linear function of the state and the error in the measurement is independent of the error in the state.

State Update

- Let us consider the case where we have a system where the state of the system evolves over time according to the following expression.

$$\mathbf{x}_t = A_t \mathbf{x}_{t-1} + B_t \mathbf{u}_t + \epsilon_t$$

  Where $\mathbf{u}_t$ denotes the control input and $\epsilon_t : \mathcal{N}(0, R_t)$ represents random additive zero mean error with covariance R_t reflecting the fact that there is error in the system.

- At time $t-1$ we have an estimate for the state of the system $\hat{\mathbf{x}}_{t-1}$ with mean $\mathbf{x}_t$ and covariance Σ_{t-1}. We want to construct an estimate for what the state will be at time t. We can do this directly from the state update expression assuming that $\hat{\mathbf{x}}_{t-1}$ and the error ϵ_t are uncorrelated.

$$\hat{\mathbf{x}}_t = A_t \hat{\mathbf{x}}_{t-1} + B_t \mathbf{u}_t$$

$$\Sigma_t = A_t \Sigma_{t-1} A_t^T + R_t$$

Kalman Filter

- Taken together these two updates rules define the Kalman filter which seeks to maintain an estimate for the state in the form a vector $\hat{\mathbf{x}}_t$ and an associated covariance matrix Σ_t.

- Whenever the system applies a control input $\mathbf{u}_t$ we update the state estimate according to the following expressions where R_t is the covariance matrix for the additive noise in the update:

$$\hat{\mathbf{x}}_t = A_t \hat{\mathbf{x}}_{t-1} + B_t \mathbf{u}_t$$

$$\Sigma_t = A_t \Sigma_{t-1} A_t^T + R_t$$

- And when we have a measurement $\mathbf{z}_t = C_t \mathbf{x}_t + \delta_t$, we update the state estimate according to the measurement rule where Q_t is the covariance matrix for the additive error in the measurement. Where $\hat{\mathbf{x}}'_t$ and Σ'_t denote the mean and covariance of the estimate before the update.

$$\begin{aligned} K_t &= \Sigma_t C_t^T (C_t \Sigma_t C_T^T + Q^T)^{-1} && (1) \\ \hat{\mathbf{x}}_t &= \hat{\mathbf{x}}'_t + K_t (z_t - C_t \hat{\mathbf{x}}'_t) && (2) \\ \Sigma_t &= (I - K_t C_t) \Sigma'_t && (3) \end{aligned}$$

Kalman Filter Notes

- The attraction of the Kalman filter is that it is easy and efficient to implement.
- Our belief about the state is represented with a normal distribution characterized by a mean and a variance. Note that this means that the distribution cannot be multimodal.
- The Kalman filter is optimal when the following conditions hold:
  - The state of the system evolves linearly in the following sense:

$$\mathbf{x}_t = A_t \mathbf{x}_{t-1} + B_t \mathbf{u}_t + \epsilon_t$$

  - The additive errors on the state evolution normally distributed and independent of each other and of the state $\epsilon_t : \mathcal{N}(0, R_t)$
  - The measurements are a linear function of the state

$$\mathbf{z}_t = C_t \mathbf{x}_t + \delta_t$$

  - The additive errors in the measurements are normally distributed and independent of each other and of the state. $\delta_t : \mathcal{N}(0, Q_t)$

Linear functions of RVs

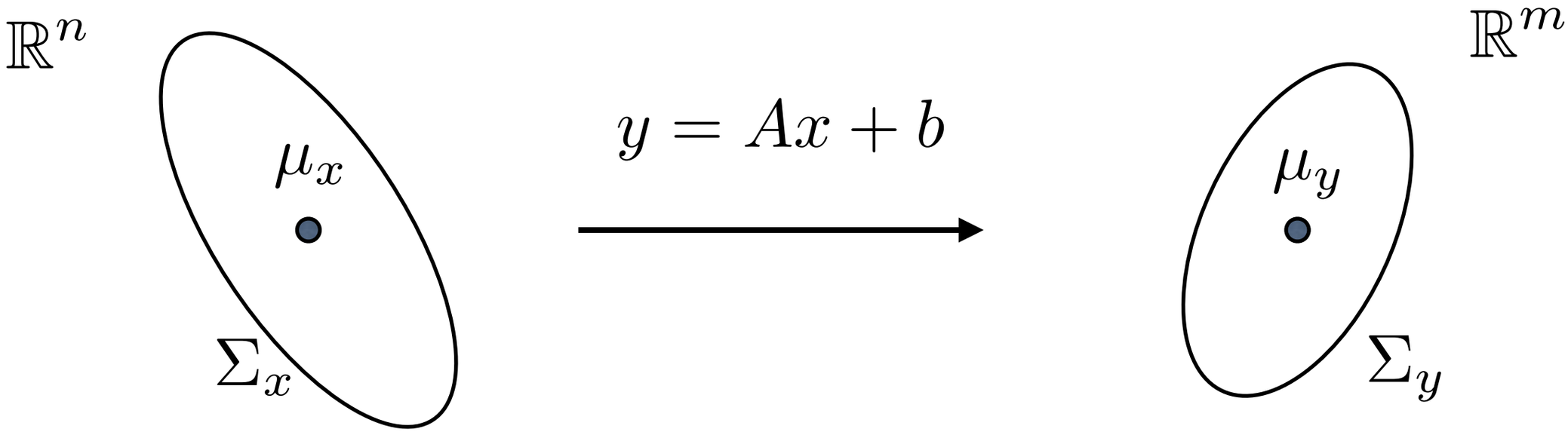
- If X is a vector valued, normally distributed random variable in $\mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$ is a constant matrix then the random variable $Y = AX$ is also a normally distributed random variable in $\mathbb{R}^m$ with mean as follows:

$$\mu_y = \mathbb{E}(Y) = \mathbb{E}(AX) = A\mathbb{E}(X) = A\mu_x$$

- The covariance of Y can be computed as follows:

$$\begin{aligned}
\Sigma_y &= \mathbb{E}[(Y - \mu_y)(Y - \mu_y)^T] \\
&= \mathbb{E}[(AX - A\mu_x)(AX - A\mu_x)^T] \\
&= \mathbb{E}[(A(X - \mu_X))(A(X - \mu_X))^T] \\
&= \mathbb{E}[A(X - \mu_X)(X - \mu_X)^T A^T] \\
&= A\mathbb{E}[(X - \mu_X)(X - \mu_X)^T]A^T \\
&= A\Sigma_x A^T
\end{aligned}$$

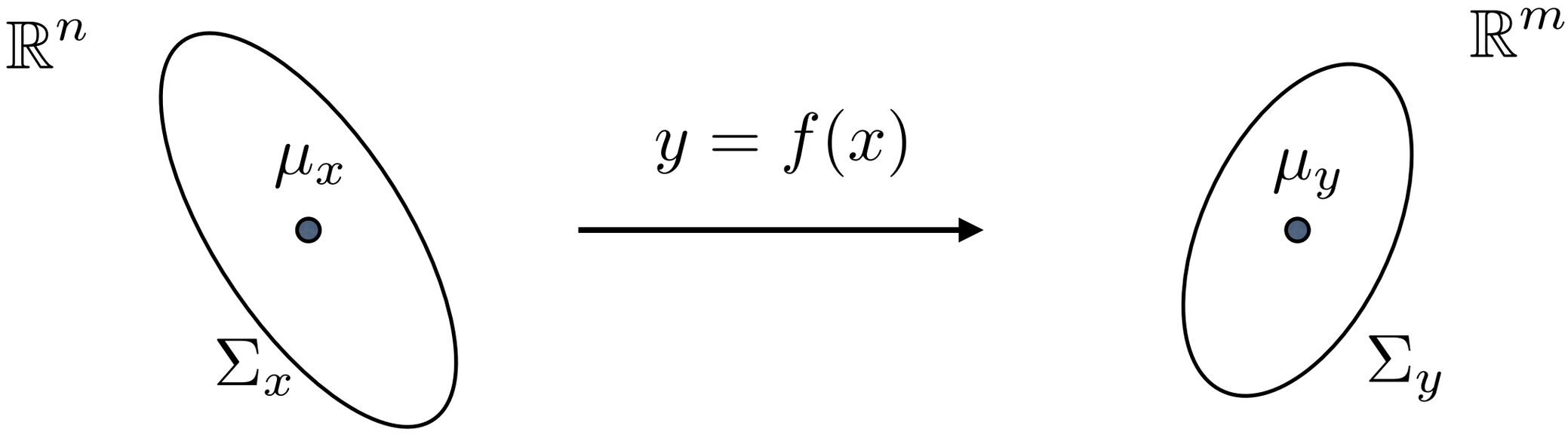
Affine Transformations



$$\mathbb{E}(X) = \mu_x \qquad \mathbb{E}(Y) = A\mu_x + b$$

$$\mathbb{E}[(X - \mu_X)(X - \mu_X)^T] = \Sigma_x \qquad \mathbb{E}[(Y - \mu_Y)(Y - \mu_Y)^T] = A\Sigma_x A^T$$

Nonlinear Transformations



$$y = f(x) \approx f(\mu_x) + J(x - \mu_x) = Jx + (f(\mu_x) - J\mu_x)$$

$$\mathbb{E}(X) = \mu_x$$
$$\mathbb{E}[(X - \mu_X)(X - \mu_X)^T] = \Sigma_x$$

$$\mu_y \approx f(\mu_x)$$
$$\mathbb{E}[(Y - \mu_Y)(Y - \mu_Y)^T] = \Sigma_y \approx J\Sigma_x J^T$$
$$J = \nabla f(x)|_{\mu_x}$$

Nonlinear Example

- Consider the following example:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = f \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1^2 + x_2 x_3 \\ \sin x_2 + \cos x_3 \end{pmatrix}$$

$$\nabla f(x) = \begin{pmatrix} 2x_1 & x_3 & x_2 \\ 0 & \cos x_2 & -\sin x_3 \end{pmatrix}$$

-

$$\mathbb{E}(x) = \mu_x = \begin{pmatrix} \mu_1^x \\ \mu_2^x \\ \mu_3^x \end{pmatrix}, \; Var(x) = \Sigma_x$$

$$J = \nabla f(x)|_{\mu_x} = \begin{pmatrix} 2\mu_1^x & \mu_3^x & \mu_2^x \\ 0 & \cos \mu_2^x & -\sin \mu_3^x \end{pmatrix}$$

$$\Sigma_y \approx J \Sigma_x J^T$$

Kalman Filter In Practice

- In practice it is often difficult to properly characterize the errors in our state evolution model and our measurements.
- We assume that the error distributions are Gaussian but don't verify properly.
- We often make simplifying assumptions about the form of the covariance matrices, for example assuming that they are diagonal.
- We often apply Kalman filtering techniques to situations where the state evolution model and the measurement model are non-linear using linearization (EKF).

Extended Kalman Filter

- There are many circumstances where the state of our system and/or the measurements are not linearly related to the state.
- We may choose to model state evolution as follows.

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

where g is a non-linear function and $\epsilon_t : \mathcal{N}(0, R_t)$

- And our measurement process as follows:

$$z_t = h(x_t) + \delta_t$$

where h is a non-linear function and $\delta_t : \mathcal{N}(0, Q_t)$

- Note that we continue to assume Gaussian additive noise

Extended Kalman Filter

- We can approximate the state update function by an affine function by linearizing about our current state estimate $\hat{\mathbf{x}}_{t-1}$

$$\mathbf{x}_t = g(\mathbf{x}_{t-1}, \mathbf{u}_t) + \epsilon_t$$

Linearizing about $\hat{\mathbf{x}}_{t-1}$

$$\mathbf{x}_t \approx G_{t-1}(\mathbf{x}_{t-1} - \hat{\mathbf{x}}_{t-1}) + g(\hat{\mathbf{x}}_{t-1}, \mathbf{u}_t) + \epsilon_t$$

Where

$$G_{t-1} = \left.\frac{\partial g}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{t-1}}$$

- With this approximation we get

$$\mathbb{E}(\hat{\mathbf{x}}_t) = g(\hat{\mathbf{x}}_{t-1}, \mathbf{u}_t)$$

and the covariance propagates as follows:

$$\Sigma_t = G_{t-1}\Sigma_{t-1}G_{t-1}^T + R_t$$

Extended Kalman Filter

- Similarly we can approximate the measurement function by an affine function by linearizing about our current state estimate before the sensor update, $\hat{\mathbf{x}}'_t$.

$$\mathbf{z}_t = h(\mathbf{x}_t) + \delta_t$$

Linearizing about $\hat{\mathbf{x}}'_t$

$$\mathbf{z}_t \approx H_t(\mathbf{x}_t - \hat{\mathbf{x}}'_t) + h(\hat{\mathbf{x}}'_t) + \delta_t$$

Where

$$H_t = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}'_t}$$

- We can use this approximation to transform our measurement into a more Kalman filter friendly form as follows:

$$\mathbf{z}'_t = \mathbf{z}_t - h(\hat{\mathbf{x}}'_t) + H_t\hat{\mathbf{x}}'_t \approx H_t\mathbf{x}_t$$

Extended Kalman Filter

- Given this approximation, $\mathbf{z}'_t$, we can apply the standard Kalman update to get.

$$\hat{\mathbf{x}}_t = \hat{\mathbf{x}}'_t + K_t(\mathbf{z}'_t - H_t\hat{\mathbf{x}}'_t)$$

Where

$$K_t = \Sigma'_t H_t(H_t\Sigma'_t H_t^T + Q_t)^{-1}$$

- Then we notice that

$$\mathbf{z}'_t - H_t\hat{\mathbf{x}}'_t = \mathbf{z}_t - h(\hat{\mathbf{x}}'_t)$$

- Which yields

$$\hat{\mathbf{x}}_t = \hat{\mathbf{x}}'_t + K_t(\mathbf{z}_t - h(\hat{\mathbf{x}}'_t))$$

and

$$\Sigma_t = (I - K_t H_t)\Sigma'_t$$

Extended Kalman Filter

- Putting it all together, whenever the system applies a control input $\mathbf{u}_t$ we update the state estimate according to the following expressions where R_t is the covariance matrix for the additive noise in the update:

$$\hat{\mathbf{x}}_t = g(\hat{\mathbf{x}}_{t-1}, \mathbf{u}_t)$$

$$\Sigma_t = G_{t-1}\Sigma_{t-1}G_{t-1}^T + R_t$$

- And when we have a measurement $\mathbf{z}_t = h(\mathbf{x}_t) + \delta_t$, we update the state estimate according to the measurement rule where Q_t is the covariance matrix for the additive error in the measurement.

$$\begin{aligned} K_t &= \Sigma_t H_t^T (H_t \Sigma_t' H_t^T + Q^T)^{-1} && (1) \\ \hat{\mathbf{x}}_t &= \hat{\mathbf{x}}_t' + K_t(z_t - h(\hat{\mathbf{x}}_t')) && (2) \\ \Sigma_t &= (I - K_t H_t)\Sigma_t' && (3) \end{aligned}$$

Robot Mapping

Unscented Kalman Filter

Cyrill Stachniss

AIS Autonomous Intelligent Systems

KF, EKF and UKF

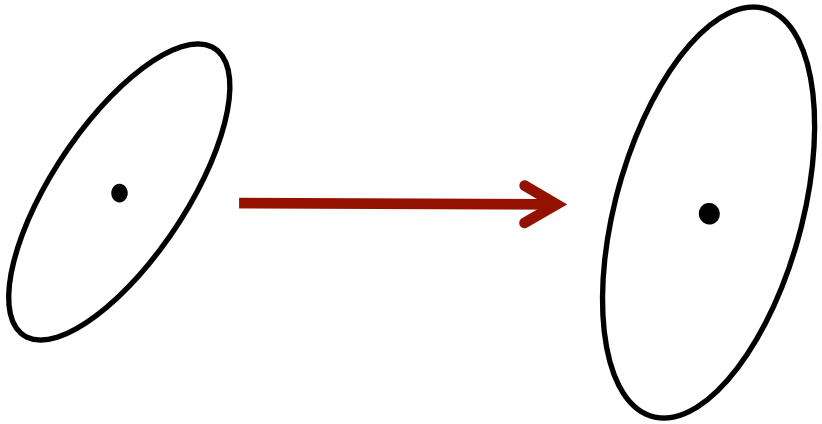
- Kalman filter requires linear models
- EKF linearizes via Taylor expansion

Is there a better way to linearize?

Unscented Transform

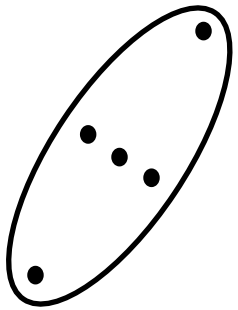
Unscented Kalman Filter (UKF)

Taylor Approximation (EKF)



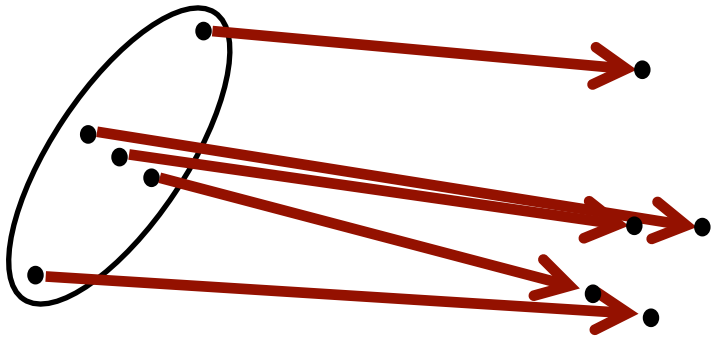
Linearization of the non-linear function through Taylor expansion

Unscented Transform



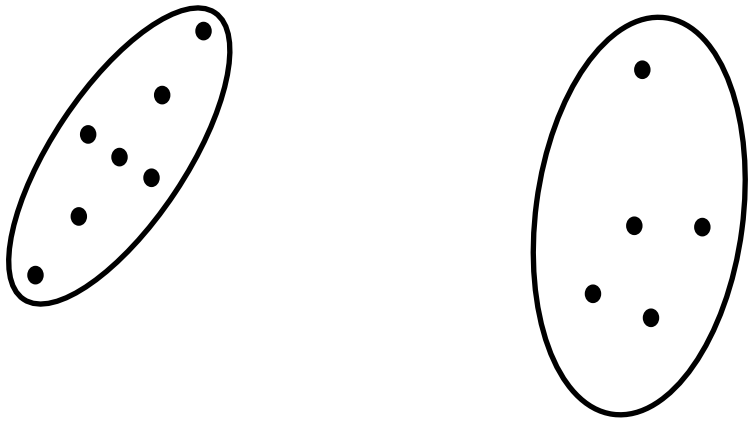
Compute a set of (so-called) sigma points

Unscented Transform



Transform each sigma point
through the non-linear function

Unscented Transform



Compute Gaussian from the transformed and weighted points

Unscented Transform Overview

- Compute a set of sigma points
- Each sigma points has a weight
- Transform the point through the non-linear function
- Compute a Gaussian from weighted points

- **Avoids to linearize around the mean as the EKF does**

Sigma Points

- How to choose the sigma points?
- How to set the weights?

Sigma Points Properties

- How to choose the sigma points?
- How to set the weights?
- Select $\mathcal{X}^{[i]}, w^{[i]}$ so that:

$$\sum_i w^{[i]} = 1$$

$$\mu = \sum_i w^{[i]} \mathcal{X}^{[i]}$$

$$\Sigma = \sum_i w^{[i]} (\mathcal{X}^{[i]} - \mu)(\mathcal{X}^{[i]} - \mu)^T$$

- There is no unique solution for $\mathcal{X}^{[i]}, w^{[i]}$

Sigma Points

- Choosing the sigma points

$$\mathcal{X}^{[0]} = \mu$$

First sigma point is the mean

Sigma Points

- Choosing the sigma points

$$
\begin{aligned}
\mathcal{X}^{[0]} &= \mu \\
\mathcal{X}^{[i]} &= \mu + \left(\sqrt{(n+\lambda)\,\Sigma}\right)_i \qquad \text{for } i = 1, \ldots, n \\
\mathcal{X}^{[i]} &= \mu - \left(\sqrt{(n+\lambda)\,\Sigma}\right)_{i-n} \qquad \text{for } i = n+1, \ldots, 2n
\end{aligned}
$$

matrix square root

dimensionality

scaling parameter

column vectors

Matrix Square Root

- Defined as S with $\Sigma = SS$
- Computed via diagonalization

$$
\begin{aligned}
\Sigma &= VDV^{-1} \\
&= V \begin{pmatrix} d_{11} & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & d_{nn} \end{pmatrix} V^{-1} \\
&= V \begin{pmatrix} \sqrt{d_{11}} & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & \sqrt{d_{nn}} \end{pmatrix}^2 V^{-1}
\end{aligned}
$$

Matrix Square Root

- Thus, we can define

$$S = V \underbrace{\begin{pmatrix} \sqrt{d_{11}} & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & \sqrt{d_{nn}} \end{pmatrix}}_{D^{1/2}} V^{-1}$$

- so that

$$SS = (VD^{1/2}V^{-1})(VD^{1/2}V^{-1}) = VDV^{-1} = \Sigma$$

Cholesky Matrix Square Root

- Alternative definition of the matrix square root

$$L \text{ with } \Sigma = LL^T$$

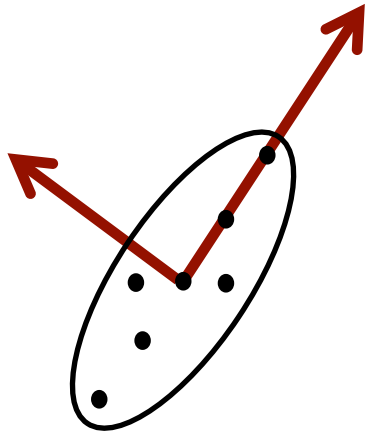
- Result of the Cholesky decomposition
- Numerically stable solution
- Often used in UKF implementations
- L and Σ have the same Eigenvectors

Sigma Points and Eigenvectors

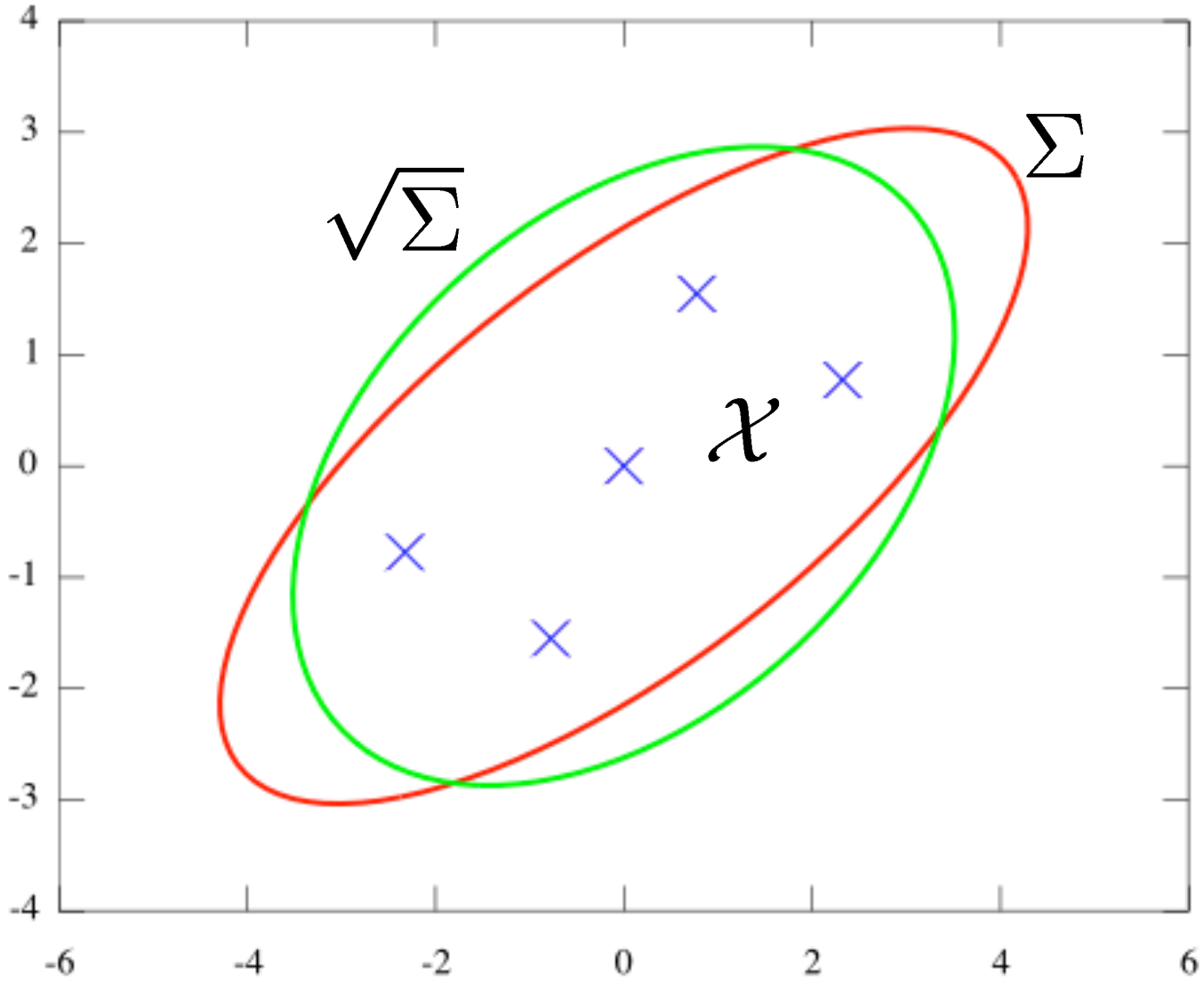
- Sigma point **can** but **do not have to** lie on the main axes of Σ

$$\mathcal{X}^{[i]} = \mu + \left(\sqrt{(n+\lambda)\,\Sigma}\right)_i \quad \text{for } i = 1, \ldots, n$$

$$\mathcal{X}^{[i]} = \mu - \left(\sqrt{(n+\lambda)\,\Sigma}\right)_{i-n} \quad \text{for } i = n+1, \ldots, 2n$$

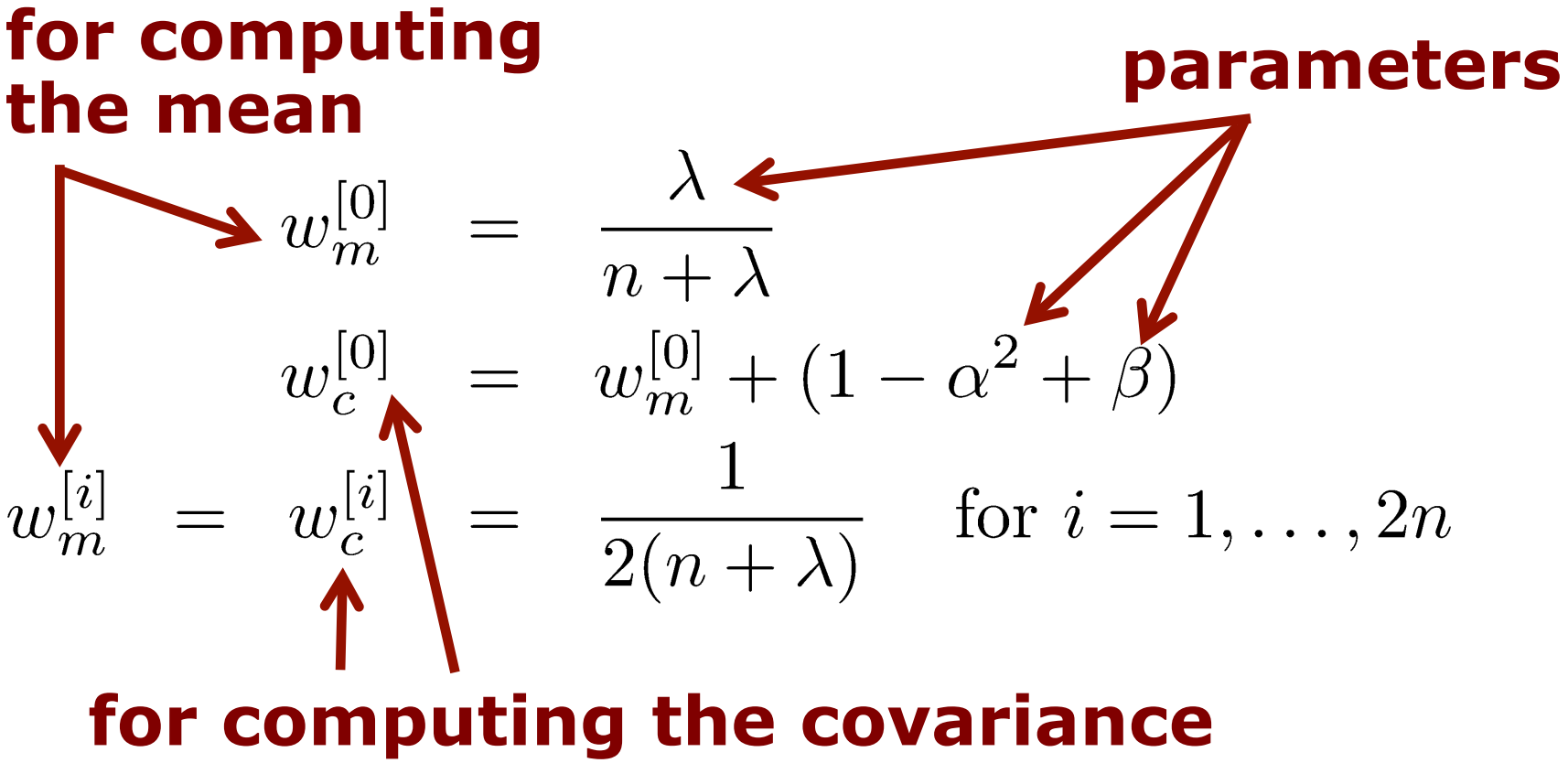


Sigma Points Example



Sigma Point Weights

- Weight sigma points



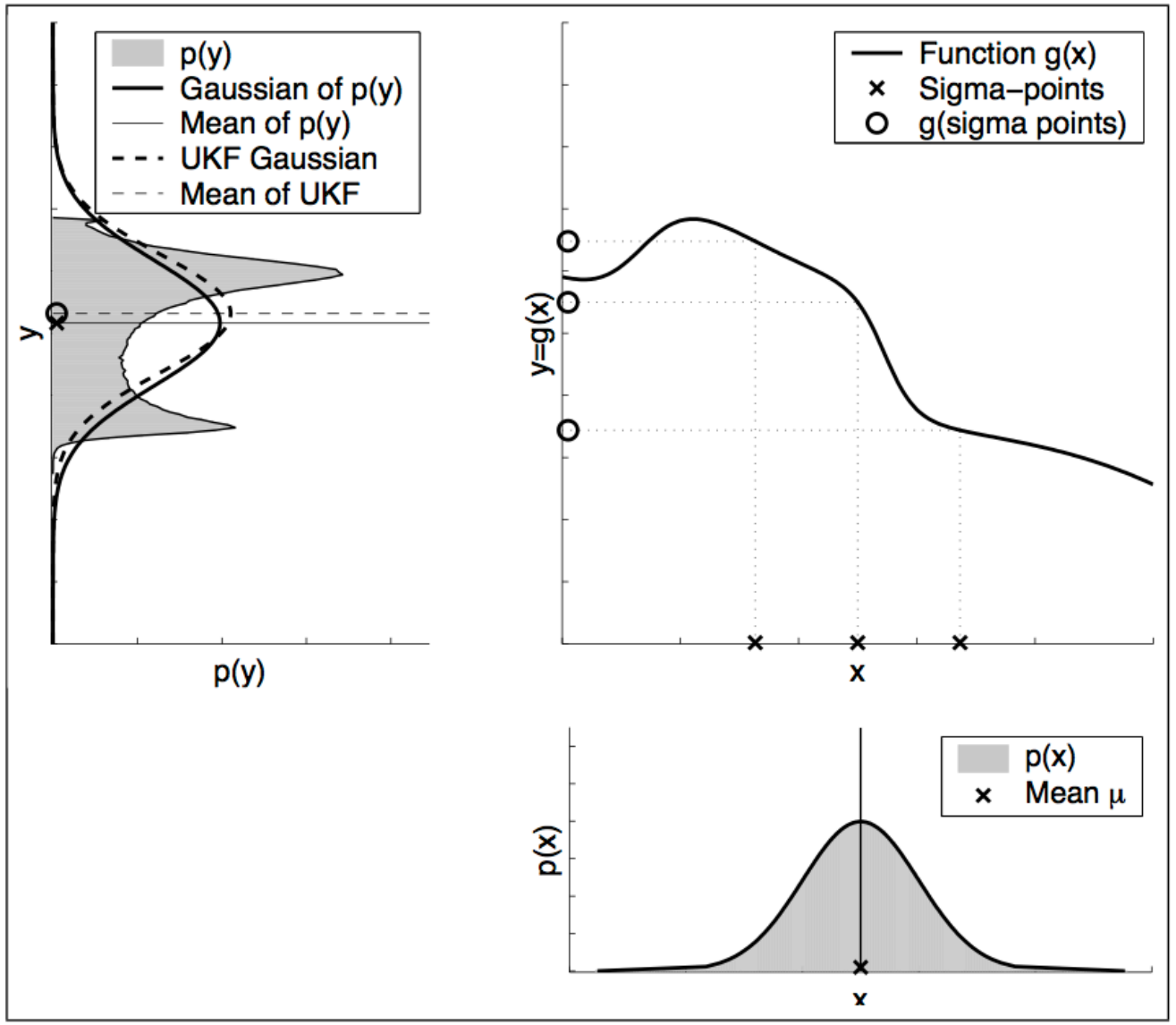
Recover the Gaussian

- Compute Gaussian from weighted and transformed points

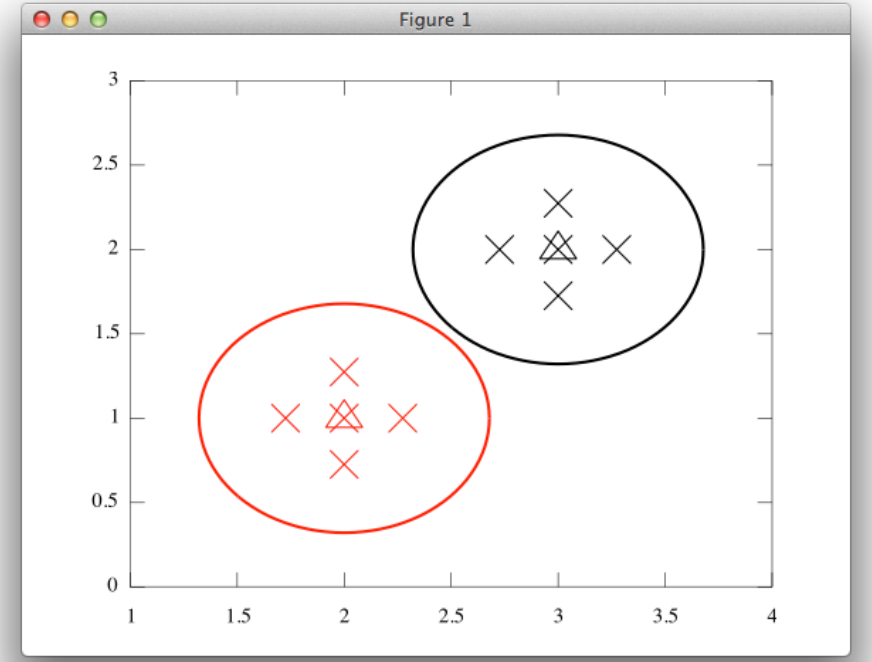
$$\mu' = \sum_{i=0}^{2n} w_m^{[i]} \, g(\mathcal{X}^{[i]})$$

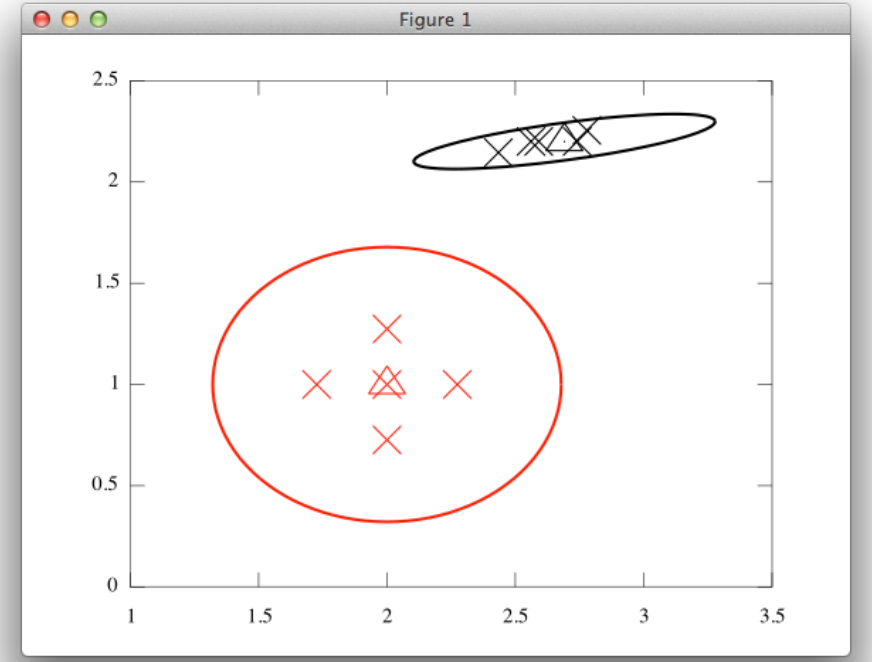
$$\Sigma' = \sum_{i=0}^{2n} w_c^{[i]} \, (g(\mathcal{X}^{[i]}) - \mu')(g(\mathcal{X}^{[i]}) - \mu')^T$$

Example


p(y)
Gaussian of p(y)
Mean of p(y)
UKF Gaussian
Mean of UKF
y
p(y)
Function g(x)
Sigma–points
g(sigma points)
y=g(x)
x
p(x)
Mean μ
p(x)
x

Examples





$$g((x,y)^T) = \begin{pmatrix} x+1 \\ y+1 \end{pmatrix}^T$$

$$g((x,y)^T) = \begin{pmatrix} 1+x+\sin(2x)+\cos(y) \\ 2+0.2y \end{pmatrix}^T$$

Unscented Transform Summary

- Sigma points

$$\begin{aligned}
\mathcal{X}^{[0]} &= \mu \\
\mathcal{X}^{[i]} &= \mu + \left(\sqrt{(n+\lambda)\,\Sigma}\right)_i \qquad \text{for } i = 1, \ldots, n \\
\mathcal{X}^{[i]} &= \mu - \left(\sqrt{(n+\lambda)\,\Sigma}\right)_{i-n} \qquad \text{for } i = n+1, \ldots, 2n
\end{aligned}$$

- Weights

$$\begin{aligned}
w_m^{[0]} &= \frac{\lambda}{n+\lambda} \\
w_c^{[0]} &= w_m^{[0]} + (1 - \alpha^2 + \beta) \\
w_m^{[i]} = w_c^{[i]} &= \frac{1}{2(n+\lambda)} \qquad \text{for } i = 1, \ldots, 2n
\end{aligned}$$

UT Parameters

- Free parameters as there is no unique solution
- Scales Unscented Transform uses

$$\kappa \geq 0$$

$$\alpha \in (0, 1]$$

Influence how far the sigma points are away from the mean

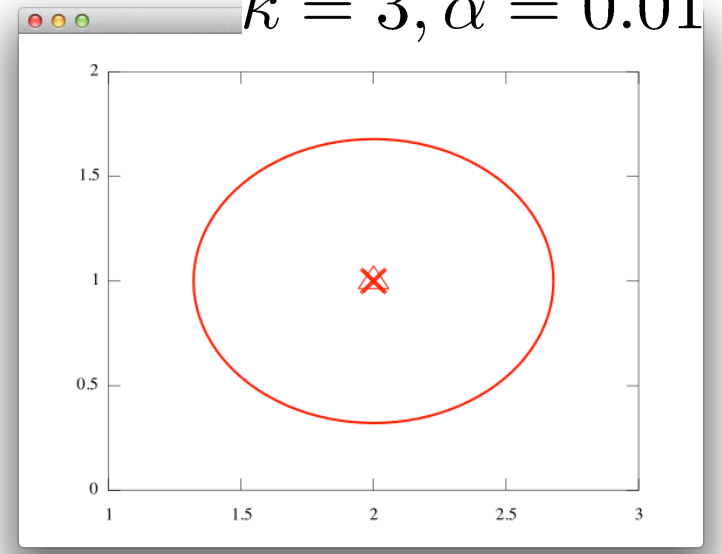
$$\lambda = \alpha^2(n + \kappa) - n$$

$$\beta = 2$$

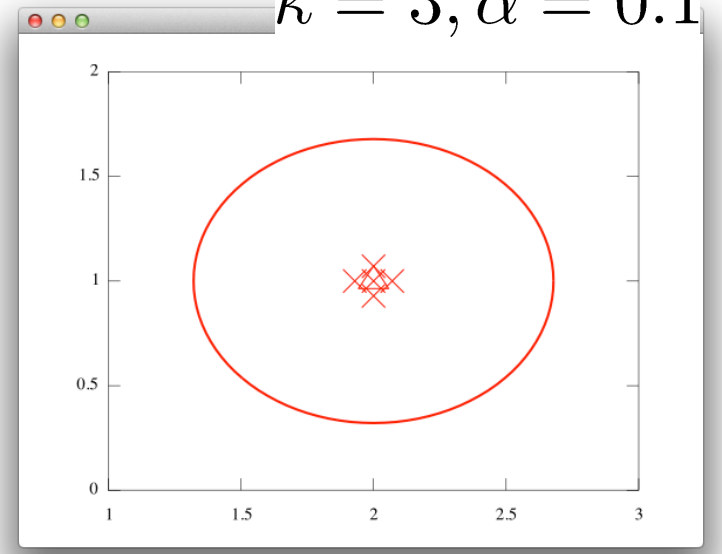
Optimal choice for Gaussians

Examples

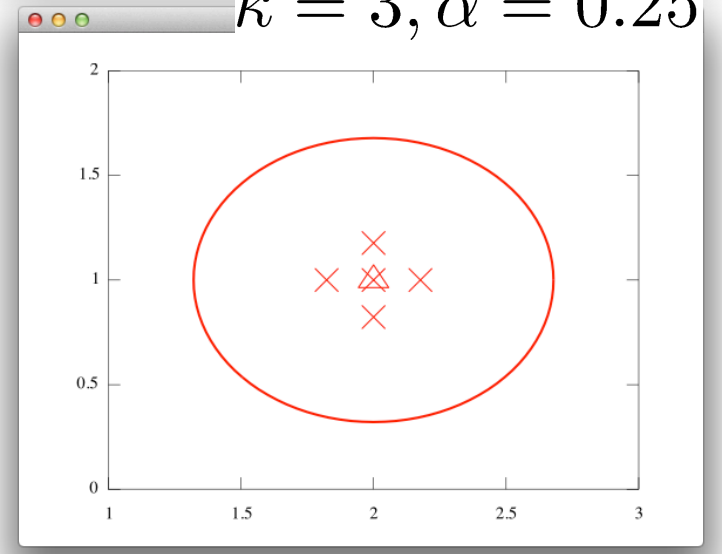
$\kappa = 3, \alpha = 0.01$



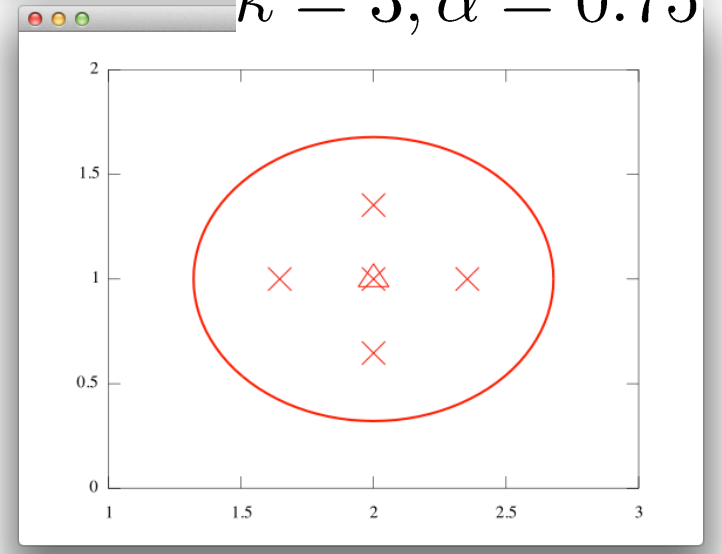
$\kappa = 3, \alpha = 0.1$



$\kappa = 3, \alpha = 0.25$

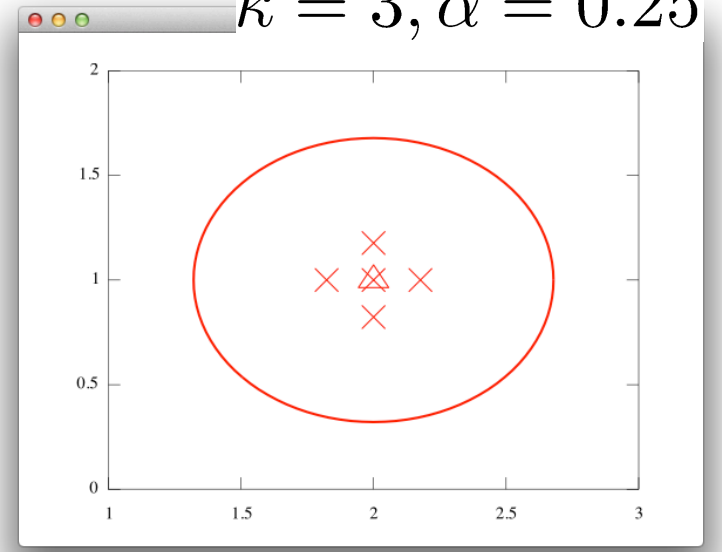


$\kappa = 3, \alpha = 0.75$

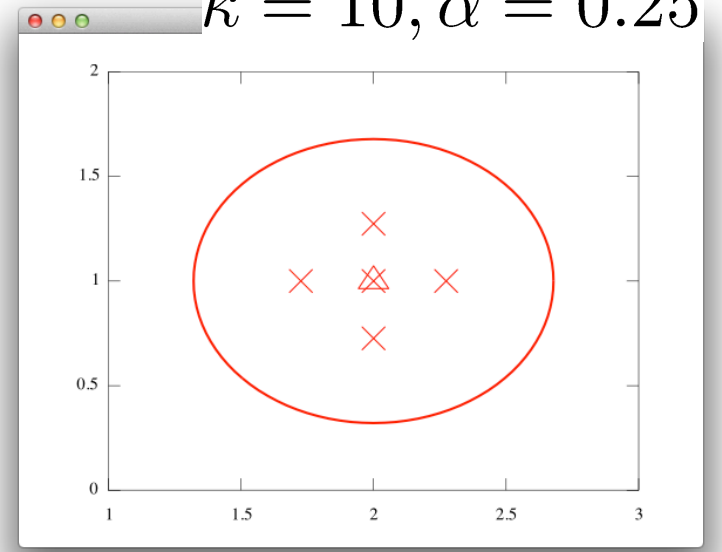


Examples

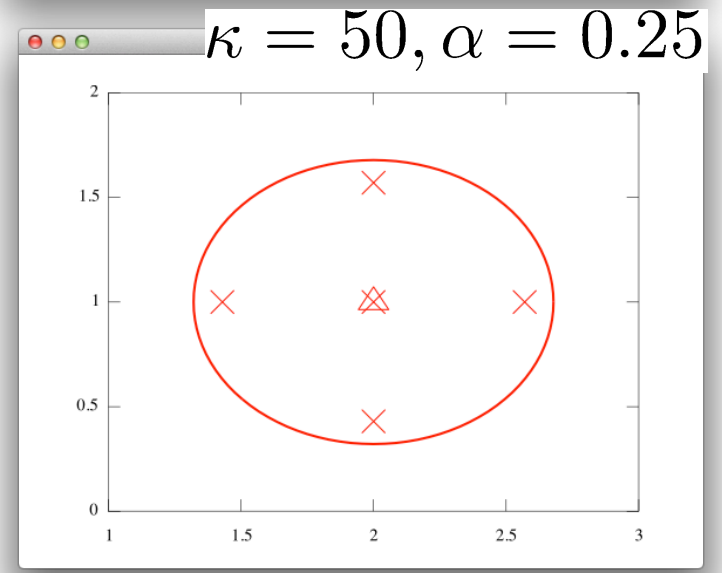
$\kappa = 3, \alpha = 0.25$



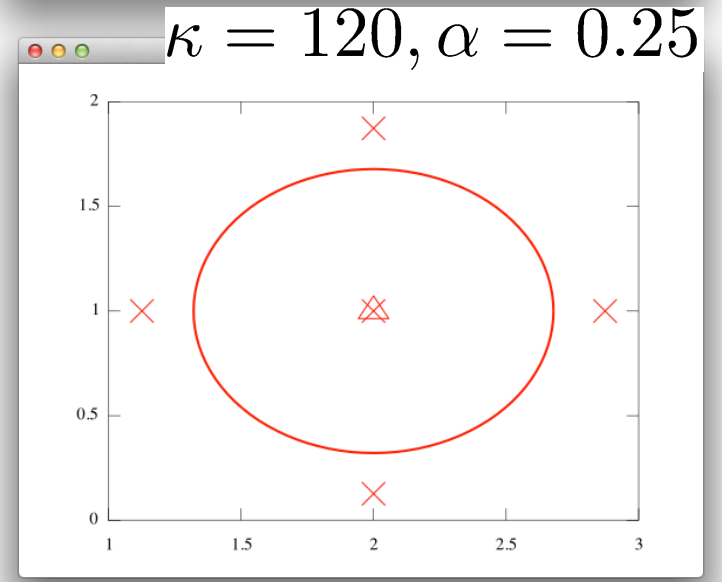
$\kappa = 10, \alpha = 0.25$



$\kappa = 50, \alpha = 0.25$



$\kappa = 120, \alpha = 0.25$



EKF Algorithm

1: **Extended_Kalman_filter**$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$**:**

2: $\bar{\mu}_t = g(u_t, \mu_{t-1})$

3: $\bar{\Sigma}_t = G_t \, \Sigma_{t-1} \, G_t^T + R_t$

4: $K_t = \bar{\Sigma}_t \, H_t^T (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$

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

6: $\Sigma_t = (I - K_t \, H_t) \, \bar{\Sigma}_t$

7: *return* μ_t, Σ_t

EKF to UKF – Prediction

Unscented

1: **~~Extended~~_Kalman_filter**$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$**:**

2: $\bar{\mu}_t =$ replace this by sigma point

3: $\bar{\Sigma}_t =$ propagation of the motion

4: $K_t = \bar{\Sigma}_t \, H_t^T (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$

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

6: $\Sigma_t = (I - K_t \, H_t) \, \bar{\Sigma}_t$

7: *return* μ_t, Σ_t

UKF Algorithm – Prediction

1: **Unscented_Kalman_filter**$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$**:**

2: $\mathcal{X}_{t-1} = (\mu_{t-1} \quad \mu_{t-1} + \gamma\sqrt{\Sigma_{t-1}} \quad \mu_{t-1} - \gamma\sqrt{\Sigma_{t-1}})$

3: $\bar{\mathcal{X}}_t^* = g(u_t, \mathcal{X}_{t-1})$

4: $\bar{\mu}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\mathcal{X}}_t^{*[i]}$

5: $\bar{\Sigma}_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{X}}_t^{*[i]} - \bar{\mu}_t)(\bar{\mathcal{X}}_t^{*[i]} - \bar{\mu}_t)^T + R_t$

EKF to UKF – Correction

Unscented

1: **~~Extended~~_Kalman_filter**$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$**:**

2: $\bar{\mu}_t =$ replace this by sigma point

3: $\bar{\Sigma}_t =$ propagation of the motion

use sigma point propagation for the expected observation and Kalman gain

5: $\mu_t = \bar{\mu}_t + K_t(z_t - \hat{z}_t)$

6: $\Sigma_t = \bar{\Sigma}_t - K_t\, S_t\, K_t^T$

7: $return\ \mu_t, \Sigma_t$

UKF Algorithm – Correction (1)

6: $\bar{\mathcal{X}}_t = (\bar{\mu}_t \quad \bar{\mu}_t + \gamma\sqrt{\bar{\Sigma}_t} \quad \bar{\mu}_t - \gamma\sqrt{\bar{\Sigma}_t})$

7: $\bar{\mathcal{Z}}_t = h(\bar{\mathcal{X}}_t)$

8: $\hat{z}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\mathcal{Z}}_t^{[i]}$

9: $S_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T + Q_t$

10: $\bar{\Sigma}_t^{x,z} = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{X}}_t^{[i]} - \bar{\mu}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T$

11: $K_t = \bar{\Sigma}_t^{x,z} \; S_t^{-1}$

UKF Algorithm – Correction (1)

6: $\bar{\mathcal{X}}_t = (\bar{\mu}_t \quad \bar{\mu}_t + \gamma\sqrt{\bar{\Sigma}_t} \quad \bar{\mu}_t - \gamma\sqrt{\bar{\Sigma}_t})$

7: $\bar{\mathcal{Z}}_t = h(\bar{\mathcal{X}}_t)$

8: $\hat{z}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\mathcal{Z}}_t^{[i]}$

9: $S_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T + Q_t$

10: $\bar{\Sigma}_t^{x,z} = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{X}}_t^{[i]} - \bar{\mu}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T$

11: $K_t = \bar{\Sigma}_t^{x,z} \, S_t^{-1}$

$$K_t = \overbrace{\bar{\Sigma}_t \, H_t^T}^{\bar{\Sigma}_t^{x,z}} \overbrace{(H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}}^{S_t}$$

(from EKF)

UKF Algorithm – Correction (2)

6: $\bar{\mathcal{X}}_t = (\bar{\mu}_t \quad \bar{\mu}_t + \gamma\sqrt{\bar{\Sigma}_t} \quad \bar{\mu}_t - \gamma\sqrt{\bar{\Sigma}_t})$

7: $\bar{\mathcal{Z}}_t = h(\bar{\mathcal{X}}_t)$

8: $\hat{z}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\mathcal{Z}}_t^{[i]}$

9: $S_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T + Q_t$

10: $\bar{\Sigma}_t^{x,z} = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{X}}_t^{[i]} - \bar{\mu}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T$

11: $K_t = \bar{\Sigma}_t^{x,z} \; S_t^{-1}$

12: $\mu_t = \bar{\mu}_t + K_t(z_t - \hat{z}_t)$

13: $\Sigma_t = \bar{\Sigma}_t - K_t \; S_t \; K_t^T$

14: *return* μ_t, Σ_t

UKF Algorithm – Correction (2)

6: $\bar{\mathcal{X}}_t = (\bar{\mu}_t \quad \bar{\mu}_t + \gamma\sqrt{\bar{\Sigma}_t} \quad \bar{\mu}_t - \gamma\sqrt{\bar{\Sigma}_t})$

7: $\bar{\mathcal{Z}}_t = h(\bar{\mathcal{X}}_t)$

8: $\hat{z}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\mathcal{Z}}_t^{[i]}$

9: $S_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T + Q_t$

10: $\bar{\Sigma}_t^{x,z} = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{X}}_t^{[i]} - \bar{\mu}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T$

11: $K_t = \bar{\Sigma}_t^{x,z} \; S_t^{-1}$

12: $\mu_t = \bar{\mu}_t + K_t(z_t - \hat{z}_t)$

13: $\Sigma_t = \bar{\Sigma}_t - K_t \; S_t \; K_t^T$

14: *return* μ_t, Σ_t

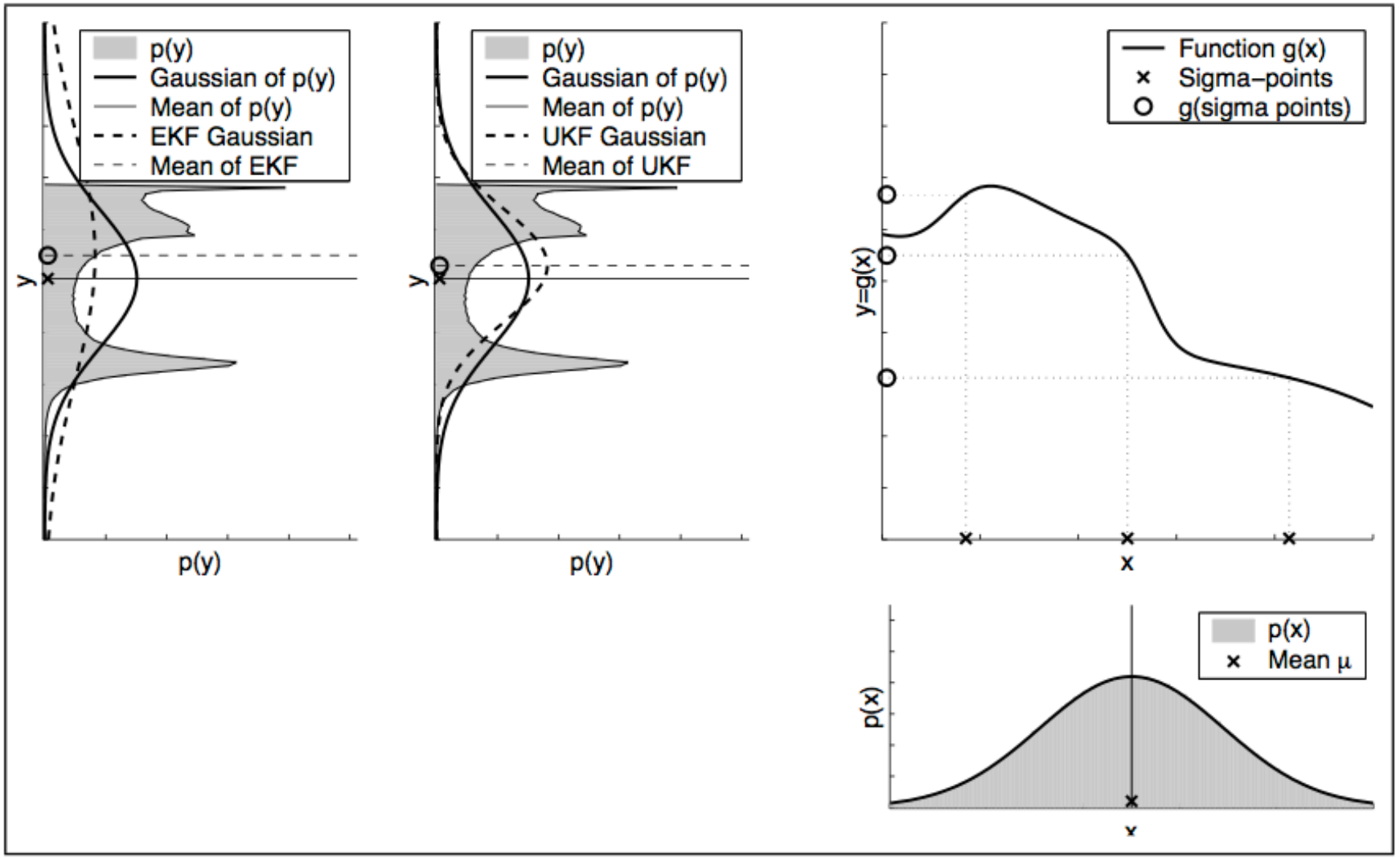
$$
\begin{aligned}
\Sigma_t &= (I - K_t H_t)\bar{\Sigma}_t \\
&= \bar{\Sigma}_t - K_t H_t \bar{\Sigma}_t \\
&= \bar{\Sigma}_t - K_t \left(\Sigma^{x,z}\right)^T \\
&= \bar{\Sigma}_t - K_t \left(\Sigma^{x,z} s_t^{-1} S_t\right)^T \\
&= \bar{\Sigma}_t - K_t \left(K_t S_t\right)^T \\
&= \bar{\Sigma}_t - K_t S_t^T K_t^T \\
&= \bar{\Sigma}_t - K_t S_t K_t^T
\end{aligned}
$$

(see next slide)

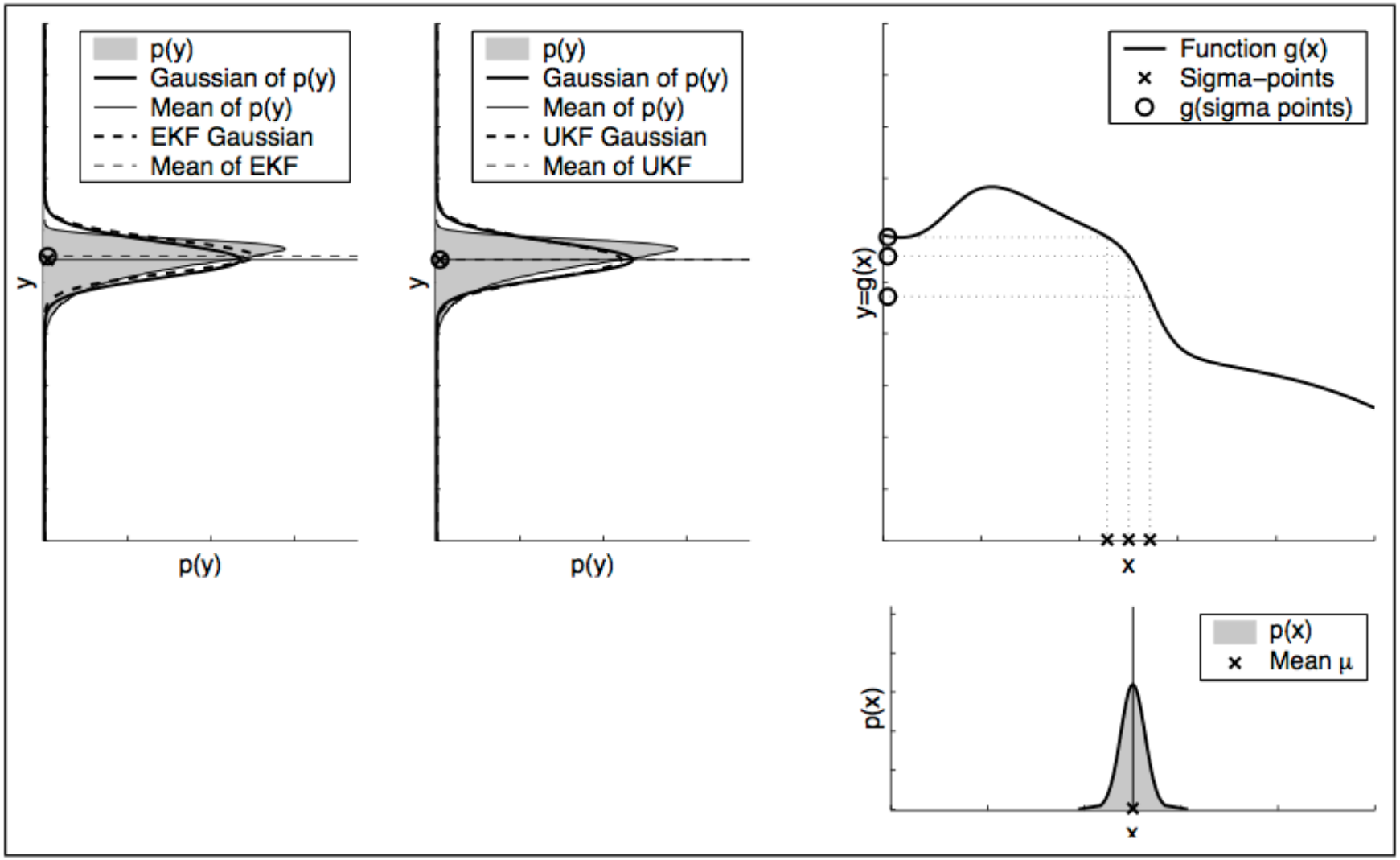
From EKF to UKF – Computing the Covariance

$$
\begin{aligned}
\Sigma_t &= (I - K_t H_t)\bar{\Sigma}_t \\
&= \bar{\Sigma}_t - K_t H_t \bar{\Sigma}_t \\
&= \bar{\Sigma}_t - K_t \left(\bar{\Sigma}^{x,z}\right)^T \\
&= \bar{\Sigma}_t - K_t \left(\bar{\Sigma}^{x,z} S_t^{-1} S_t\right)^T \\
&= \bar{\Sigma}_t - K_t \left(K_t S_t\right)^T \\
&= \bar{\Sigma}_t - K_t S_t^T K_t^T \\
&= \bar{\Sigma}_t - K_t S_t K_t^T
\end{aligned}
$$

UKF vs. EKF

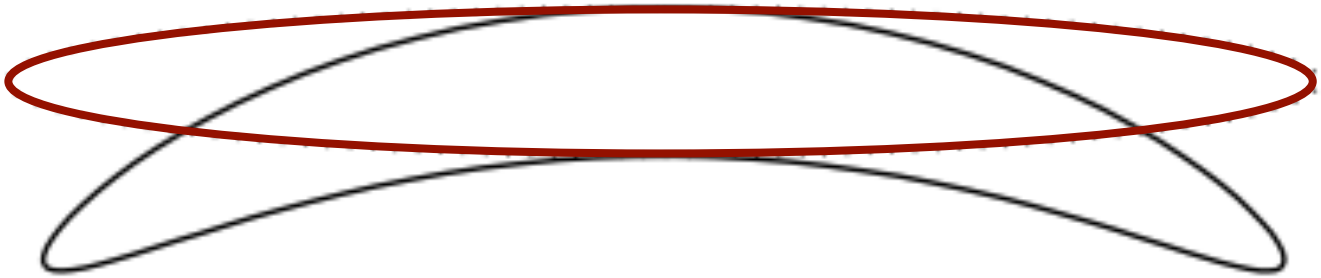

p(y)
Gaussian of p(y)
Mean of p(y)
EKF Gaussian
Mean of EKF
y
p(y)
p(y)
Gaussian of p(y)
Mean of p(y)
UKF Gaussian
Mean of UKF
y
p(y)
Function g(x)
Sigma−points
g(sigma points)
y=g(x)
x
p(x)
Mean μ
p(x)
x

UKF vs. EKF (Small Covariance)

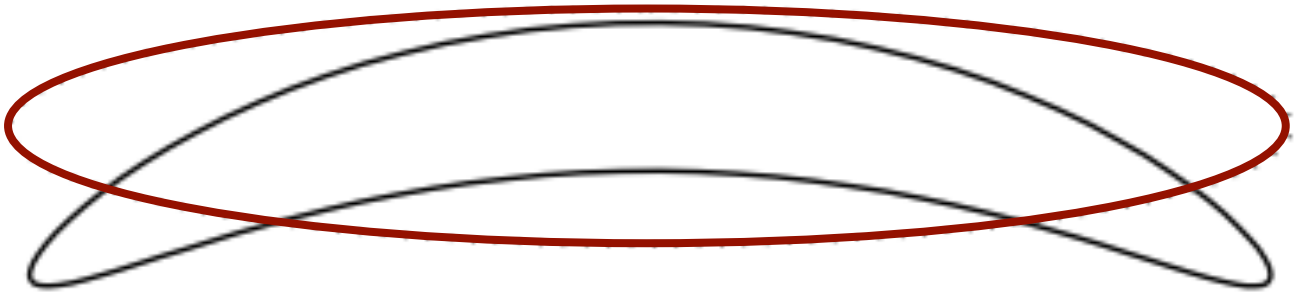


UKF vs. EKF – Banana Shape

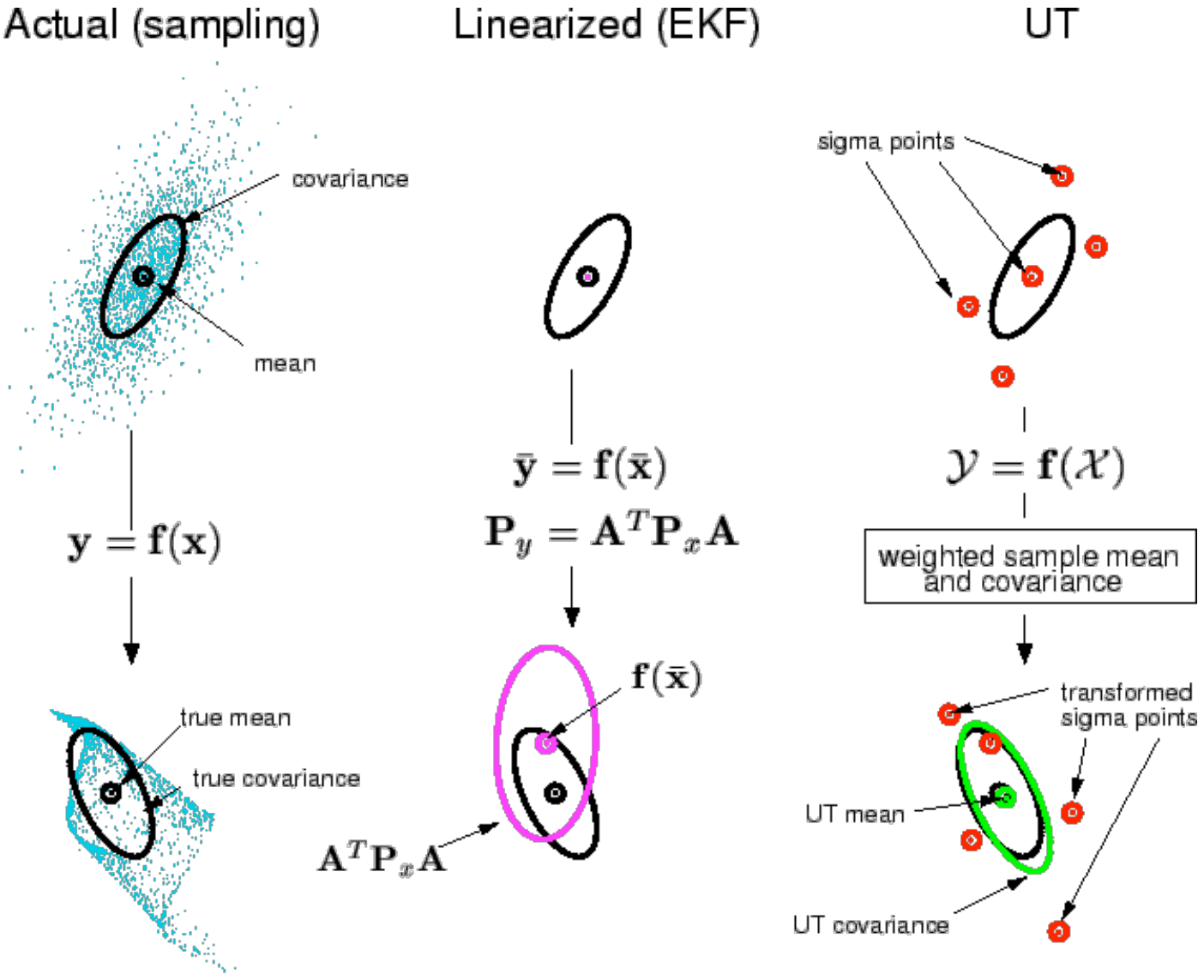
EKF approximation



UKF approximation



UKF vs. EKF



Courtesy: E.A. Wan and R. van der Merwe

UT/UKF Summary

- Unscented transforms as an alternative to linearization
- UT is a better approximation than Taylor expansion
- UT uses sigma point propagation
- Free parameters in UT
- UKF uses the UT in the prediction and correction step

UKF vs. EKF

- Same results as EKF for linear models
- Better approximation than EKF for non-linear models
- Differences often “somewhat small”
- No Jacobians needed for the UKF
- Same complexity class
- Slightly slower than the EKF
- Still requires Gaussian distributions

Literature

Unscented Transform and UKF

- Thrun et al.: “Probabilistic Robotics”, Chapter 3.4
- “A New Extension of the Kalman Filter to Nonlinear Systems” by Julier and Uhlmann, 1995
- “Dynamische Zustandsschätzung” by Fränken, 2006, pages 31-34

Robot Mapping

Short Introduction to Particle Filters and Monte Carlo Localization

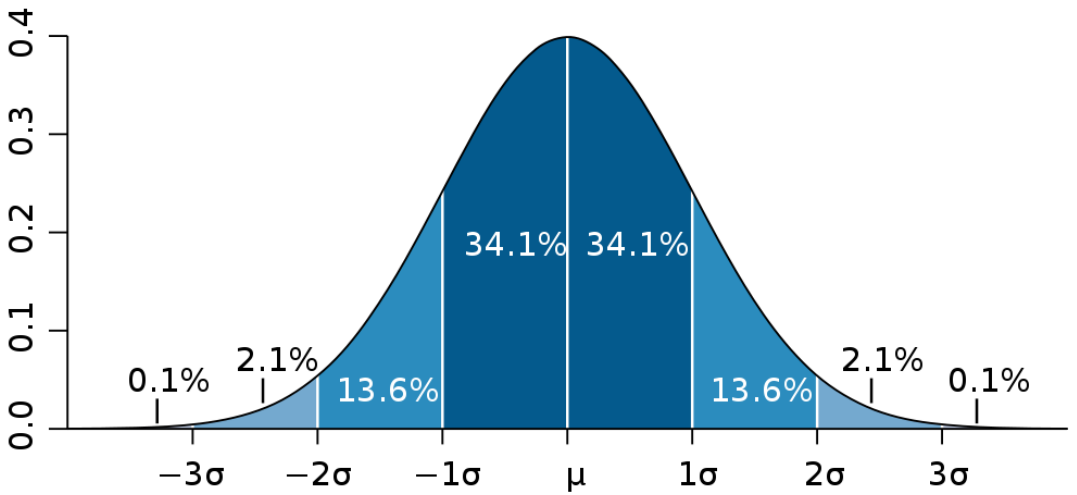
Cyrill Stachniss

AIS Autonomous Intelligent Systems

Gaussian Filters

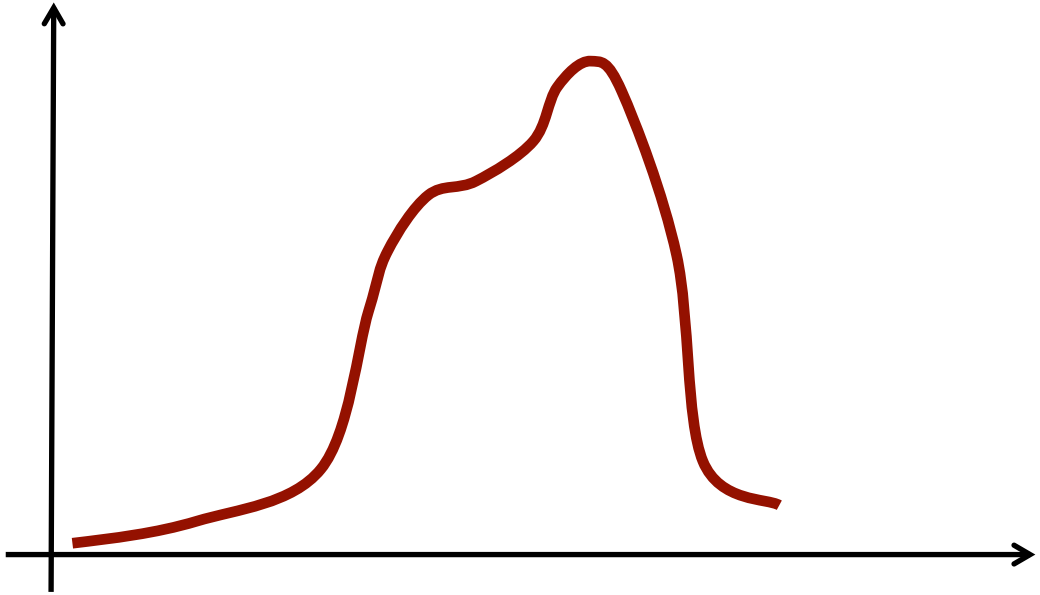
- The Kalman filter and its variants can only model **Gaussian distributions**

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



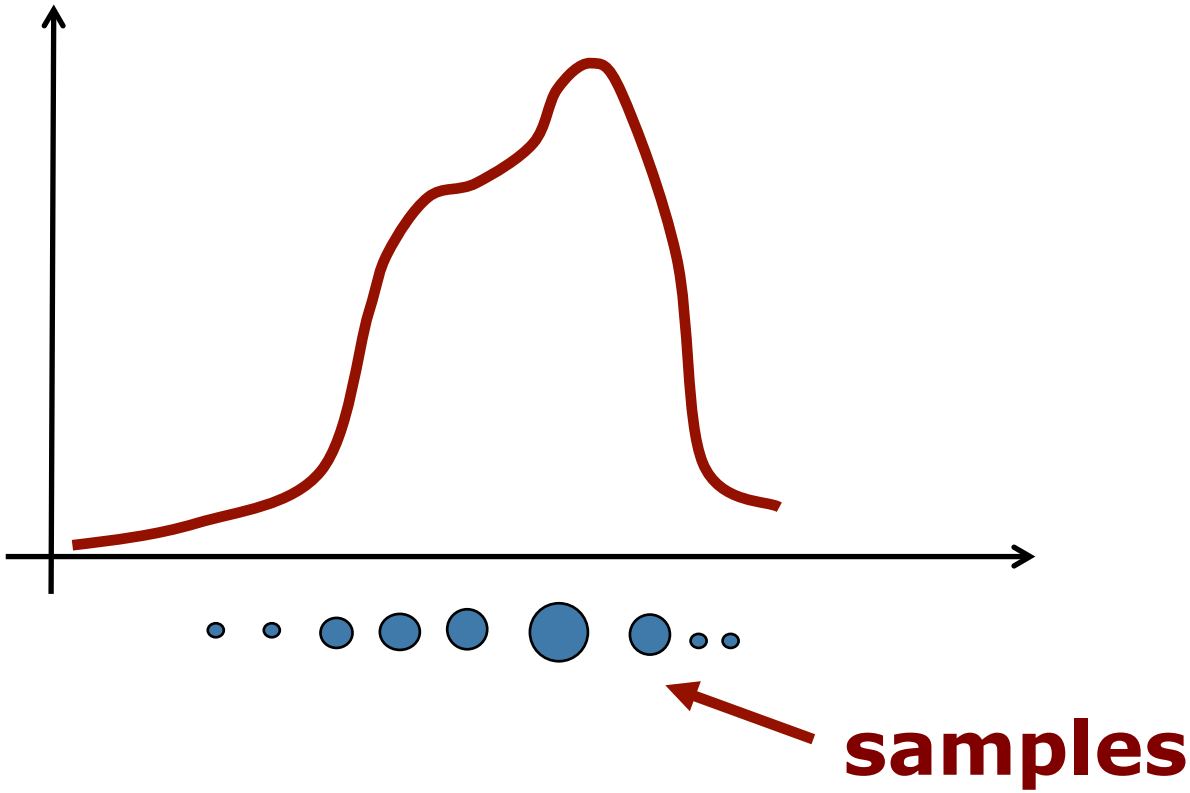
Motivation

- Goal: approach for dealing with **arbitrary distributions**



Key Idea: Samples

- Use **multiple samples** to represent arbitrary distributions



Particle Set

- Set of weighted samples

$$\mathcal{X} = \left\{ \left\langle x^{[i]}, w^{[i]} \right\rangle \right\}_{i=1,\ldots,N}$$

state hypothesis

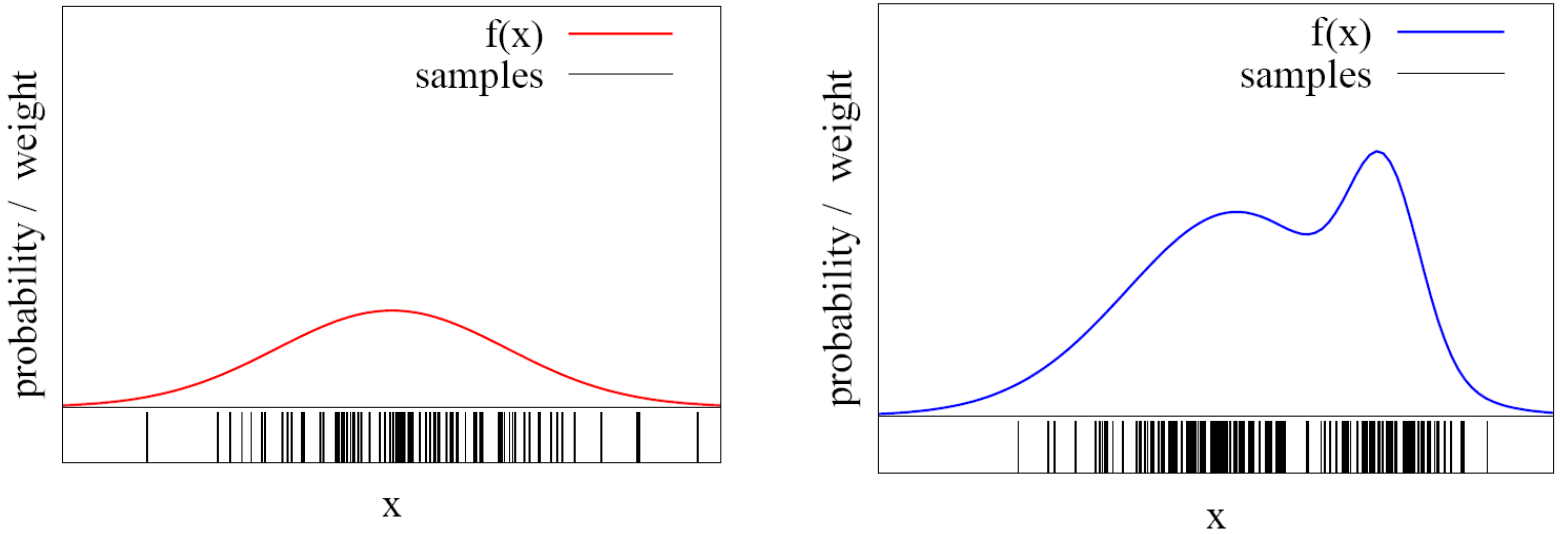
importance weight

- The samples represent the posterior

$$p(x) = \sum_{i=1}^{N} w^{[i]} \, \delta_{x^{[i]}}(x)$$

Particles for Approximation

- Particles for function approximation

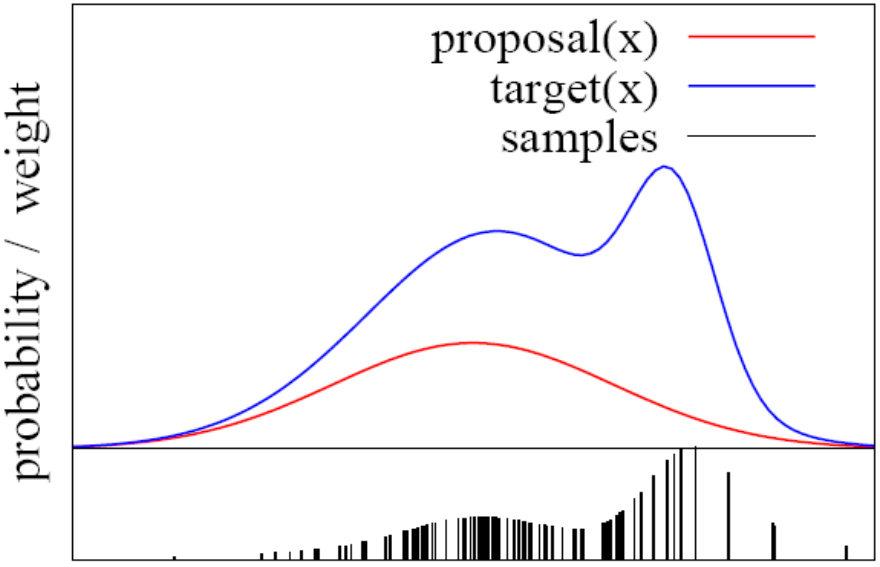


- The more particles fall into an interval, the higher its probability density

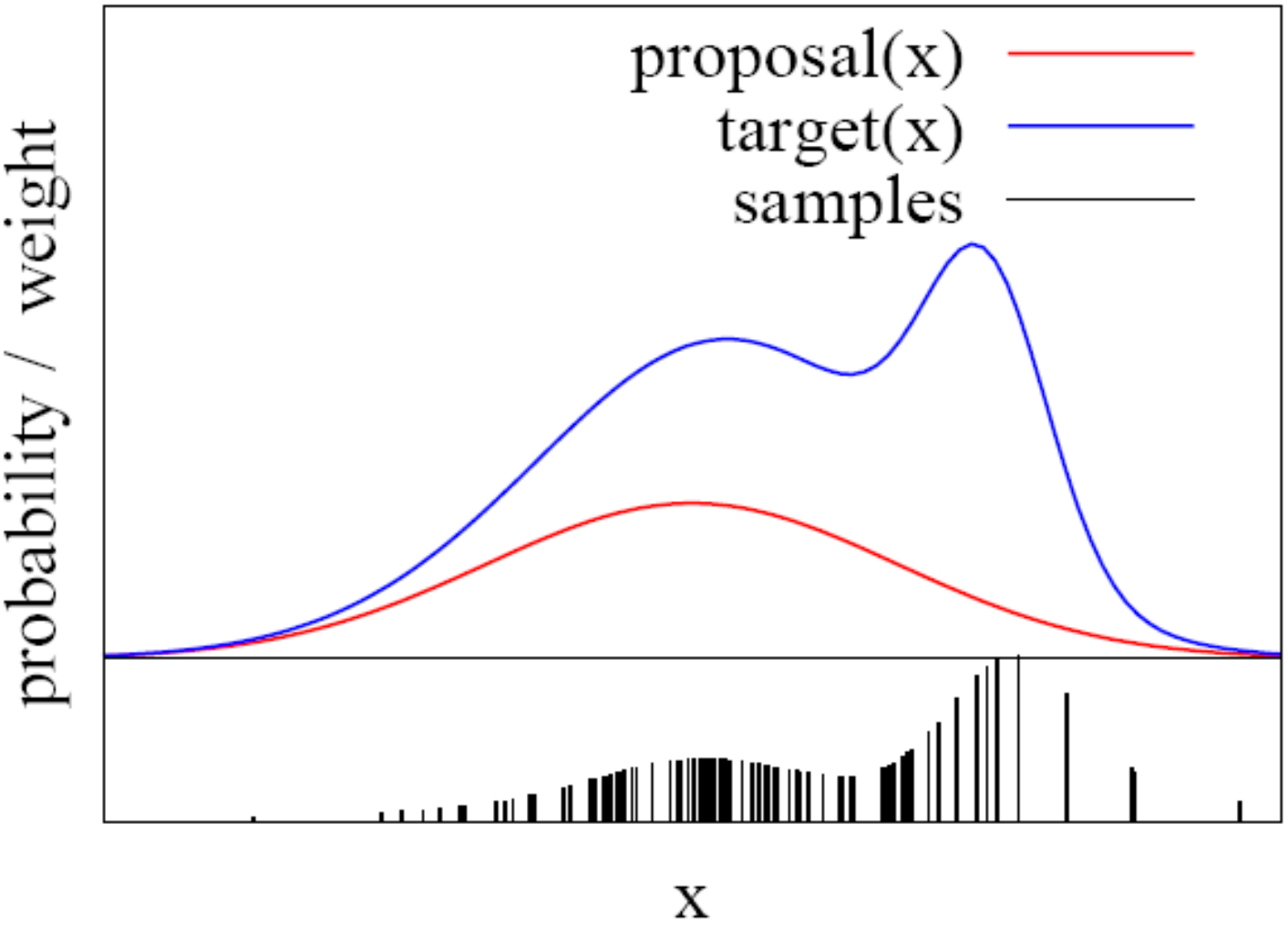
How to obtain such samples?

Importance Sampling Principle

- We can use a different distribution g to generate samples from f
- Account for the "differences between g and f" using a weight $w = f / g$
- target f
- proposal g
- Pre-condition:
  $f(x)>0 \rightarrow g(x)>0$



Importance Sampling Principle



Particle Filter

- Recursive Bayes filter
- Non-parametric approach
- Models the distribution by samples
- Prediction: draw from the proposal
- Correction: weighting by the ratio of target and proposal

The more samples we use, the better is the estimate!

Particle Filter Algorithm

1. Sample the particles using the proposal distribution

$$x_t^{[i]} \sim \pi(x_t \mid \ldots)$$

2. Compute the importance weights

$$w_t^{[i]} = \frac{target(x_t^{[i]})}{proposal(x_t^{[i]})}$$

3. Resampling: “Replace unlikely samples by more likely ones”

Particle Filter Algorithm

Particle_filter$(\mathcal{X}_{t-1}, u_t, z_t)$**:**

1: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$

2: *for* $m = 1$ *to* M *do*

3: *sample* $x_t^{[m]} \sim \pi(x_t)$

4: $w_t^{[m]} = \frac{p(x_t^{[m]})}{\pi(x_t^{[m]})}$

5: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$

6: *endfor*

7: *for* $m = 1$ *to* M *do*

8: *draw* i *with probability* $\propto w_t^{[i]}$

9: *add* $x_t^{[i]}$ *to* $\mathcal{X}_t$

10: *endfor*

11: *return* $\mathcal{X}_t$

Monte Carlo Localization

- Each particle is a pose hypothesis
- Proposal is the motion model

$$x_t^{[i]} \sim p(x_t \mid x_{t-1}, u_t)$$

- Correction via the observation model

$$w_t^{[i]} = \frac{target}{proposal} \propto p(z_t \mid x_t, m)$$

Particle Filter for Localization

Particle_filter$(\mathcal{X}_{t-1}, u_t, z_t)$**:**

1: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$

2: *for* $m = 1$ *to* M *do*

3: *sample* $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$

4: $w_t^{[m]} = p(z_t \mid x_t^{[m]})$

5: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$

6: *endfor*

7: *for* $m = 1$ *to* M *do*

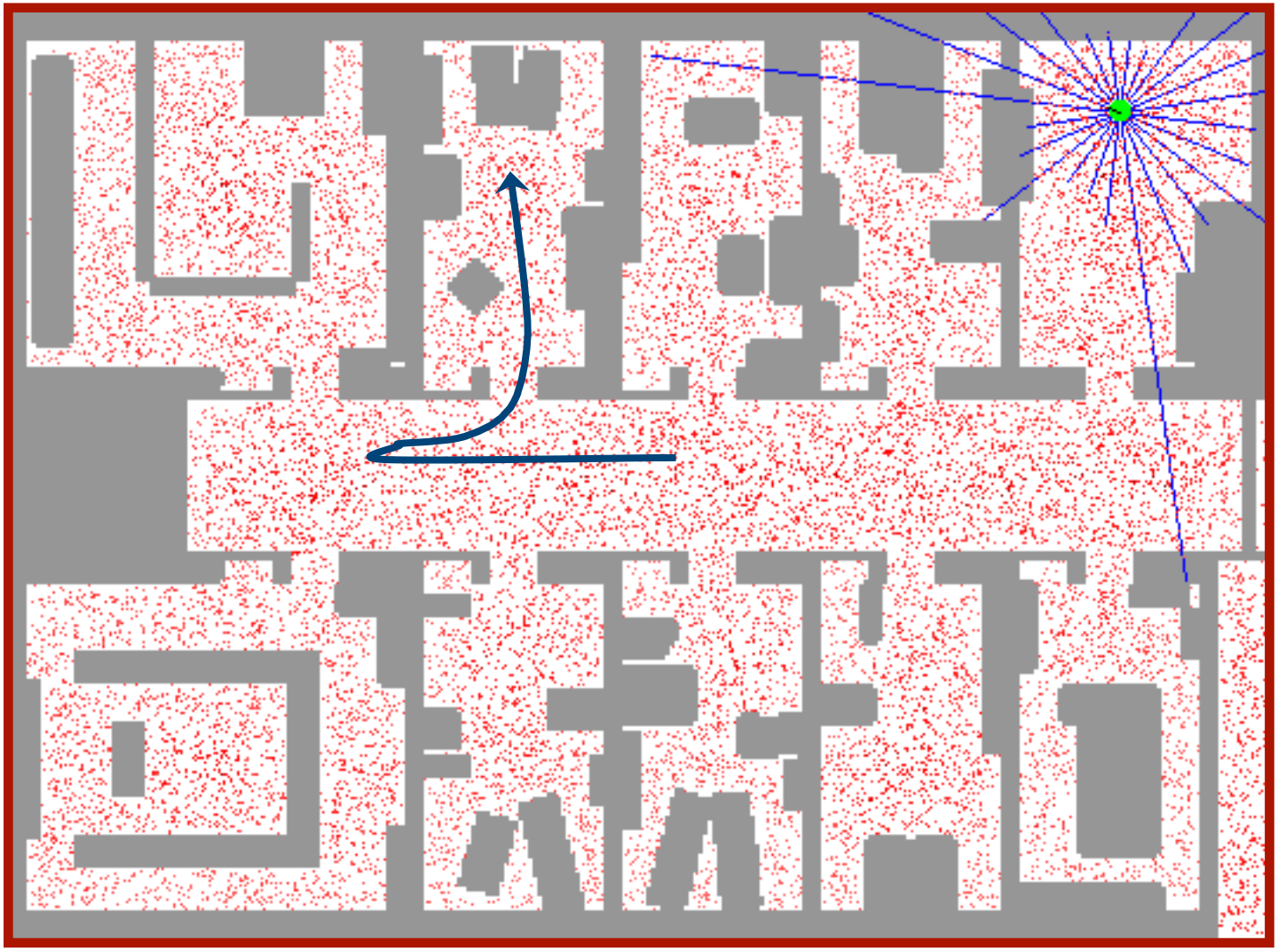
8: *draw* i *with probability* $\propto w_t^{[i]}$

9: *add* $x_t^{[i]}$ *to* $\mathcal{X}_t$

10: *endfor*

11: *return* $\mathcal{X}_t$

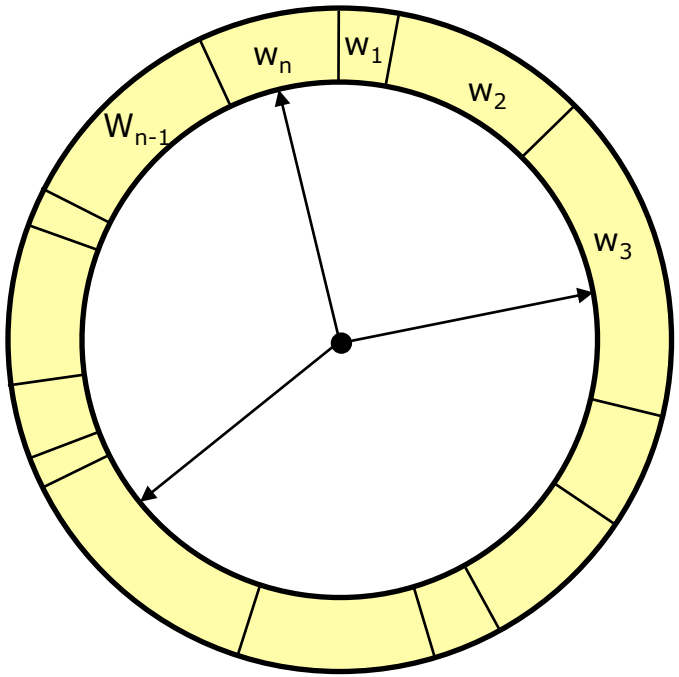
Application: Particle Filter for Localization (Known Map)



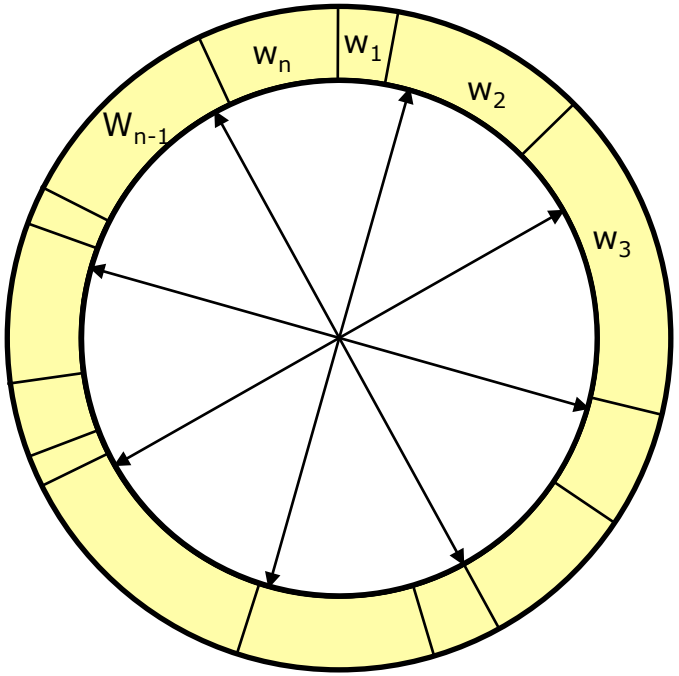
Resampling

- Survival of the fittest: “Replace unlikely samples by more likely ones”
- “Trick” to avoid that many samples cover unlikely states
- Needed as we have a limited number of samples

Resampling



- Roulette wheel
- Binary search
- O(n log n)



- Stochastic universal sampling
- Low variance
- O(n)

Low Variance Resampling

Low_variance_resampling$(\mathcal{X}_t, \mathcal{W}_t)$**:**

$\bar{\mathcal{X}}_t = \emptyset$

$r = \text{rand}(0; M^{-1})$

$c = w_t^{[1]}$

$i = 1$

for $m = 1$ *to* M *do*

  $U = r + (m-1) \cdot M^{-1}$

  while $U > c$

    $i = i + 1$

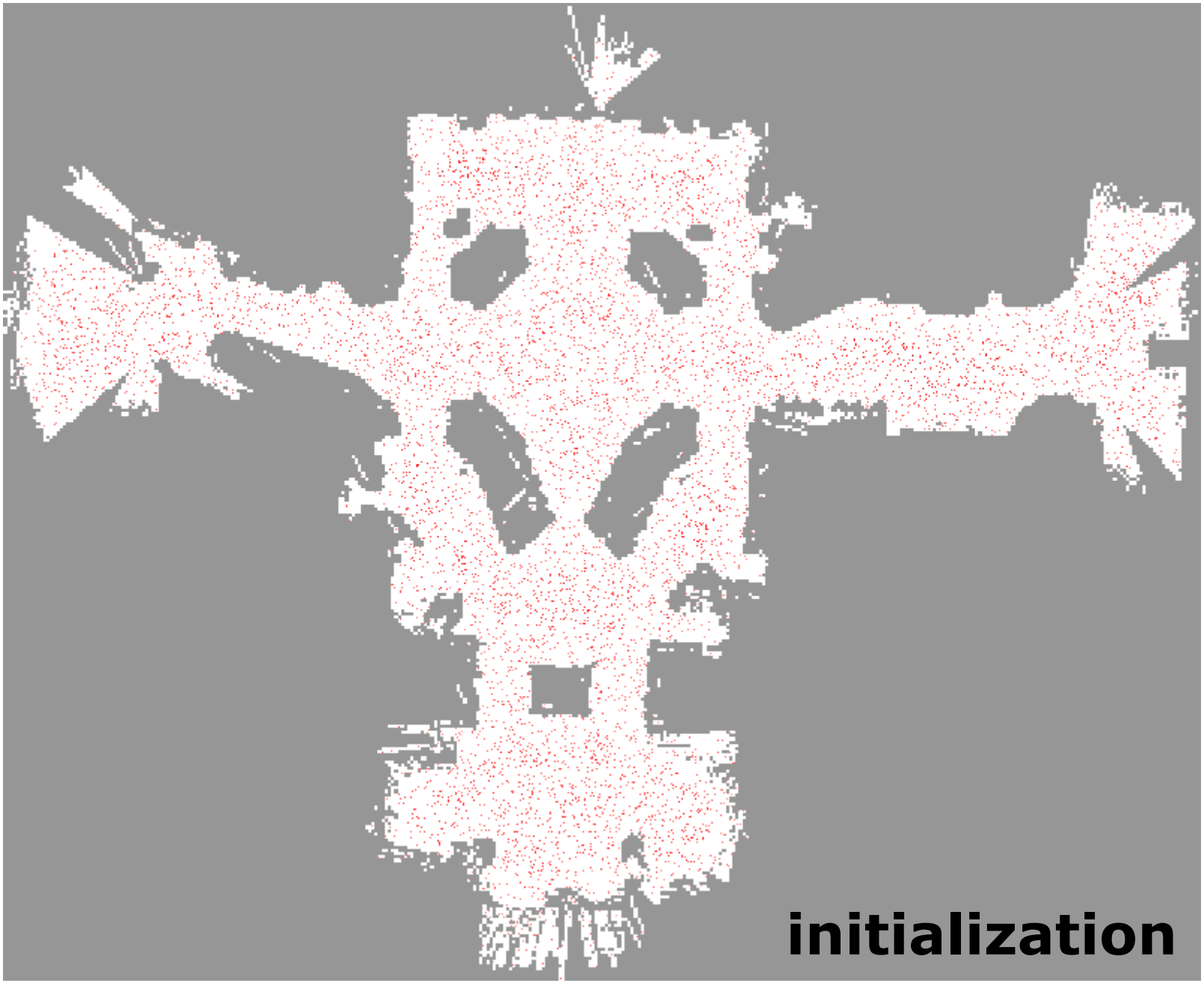
    $c = c + w_t^{[i]}$

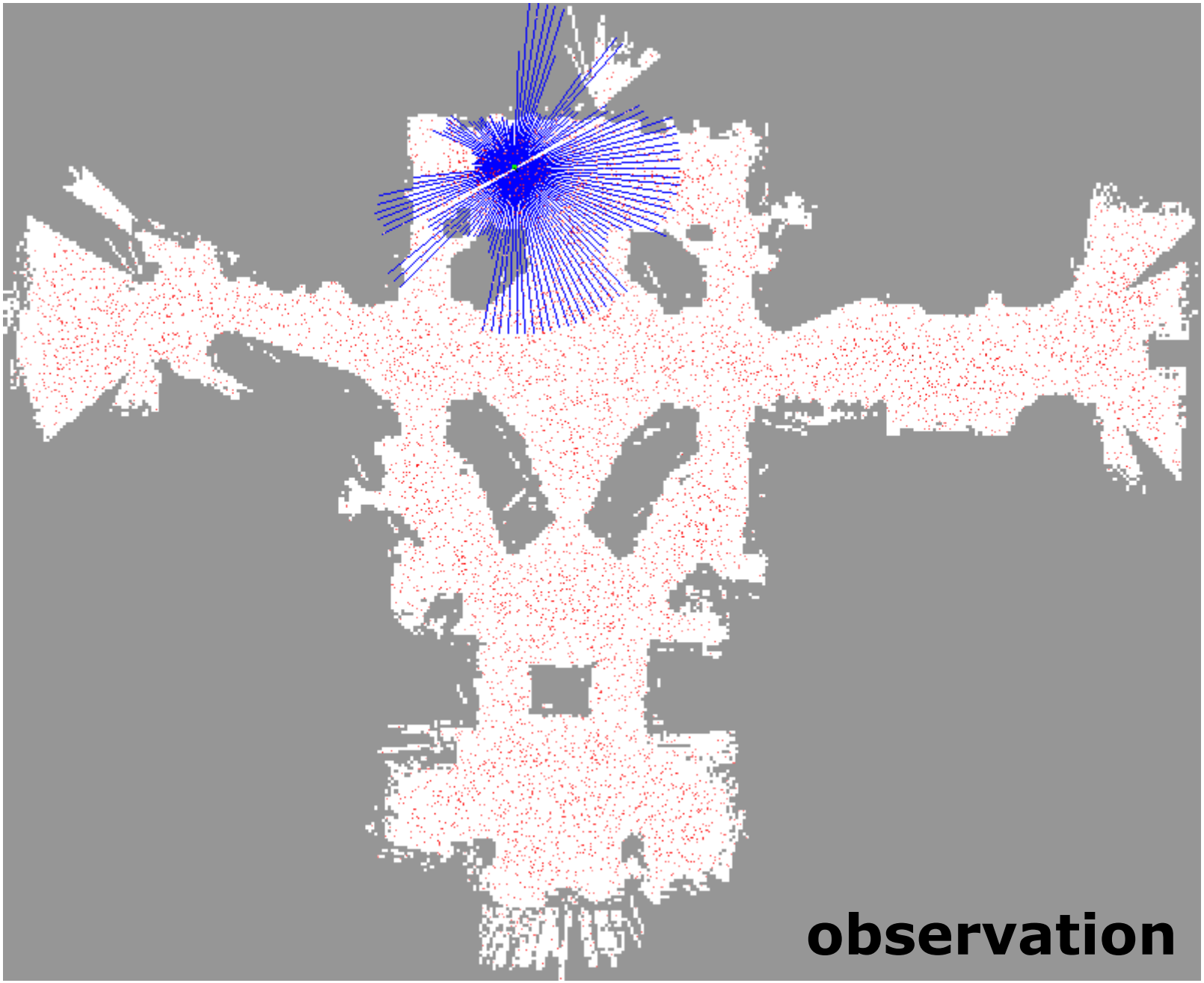
  endwhile

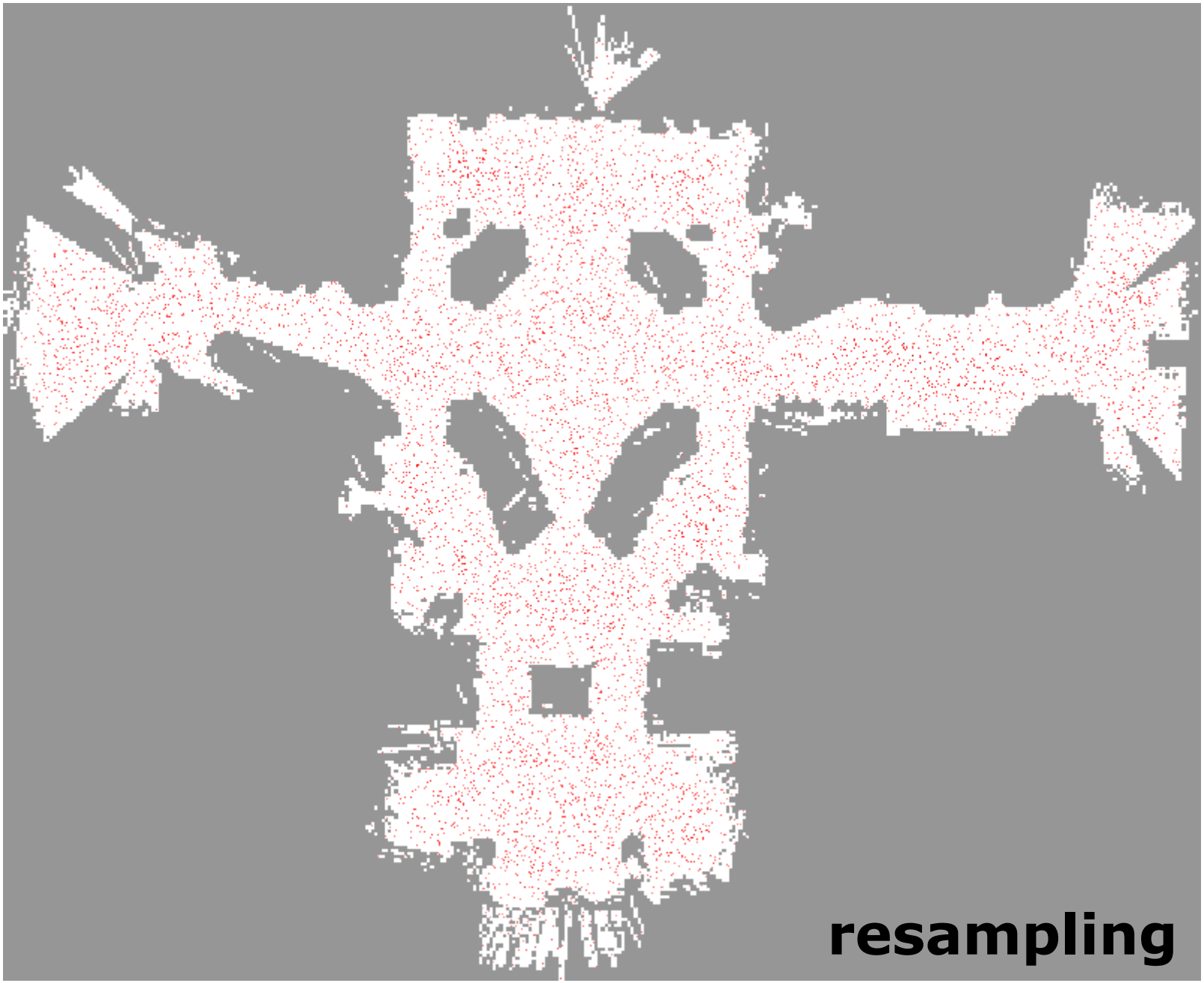
  add $x_t^{[i]}$ *to* $\bar{\mathcal{X}}_t$

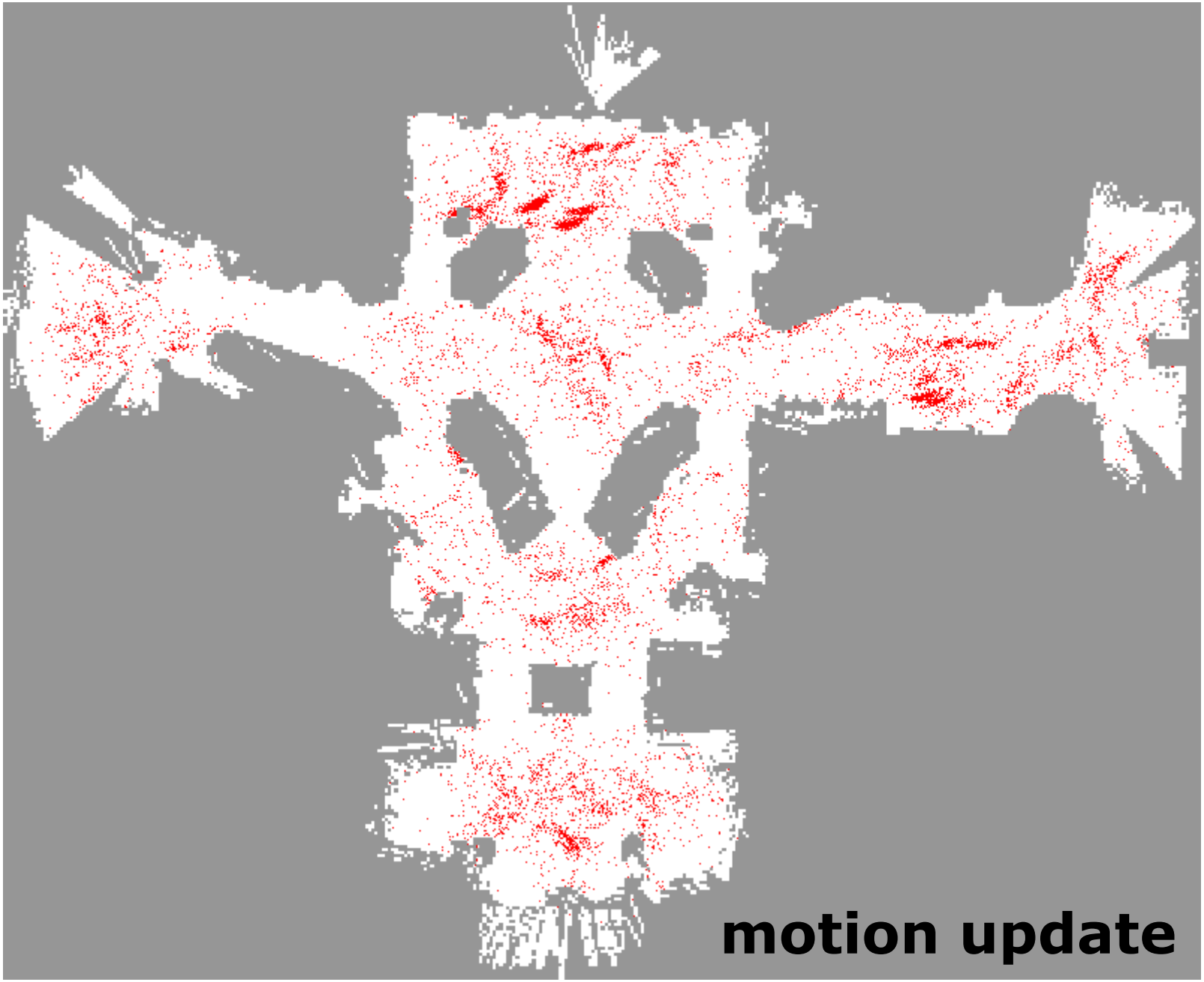
endfor

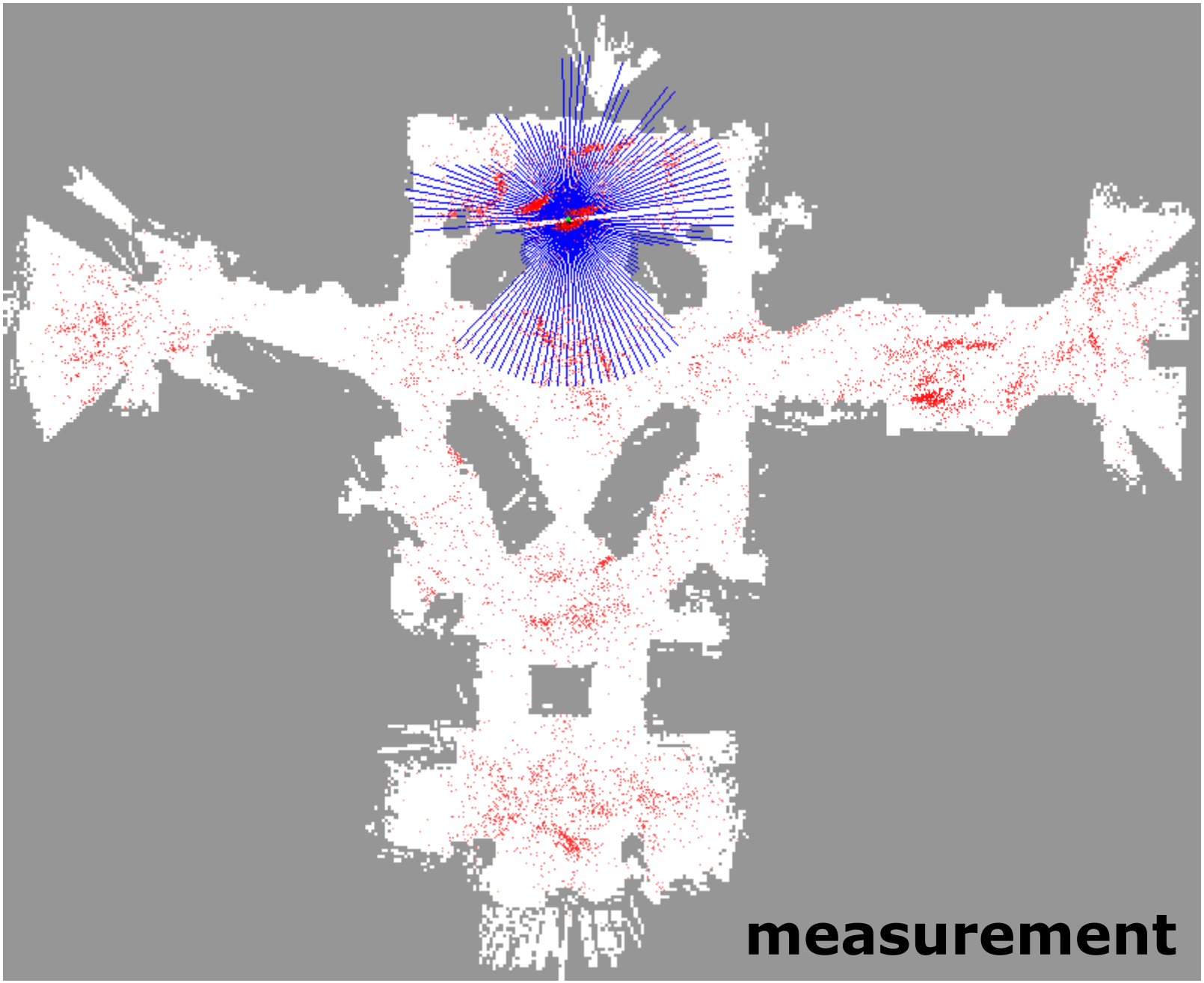
return $\bar{\mathcal{X}}_t$


initialization

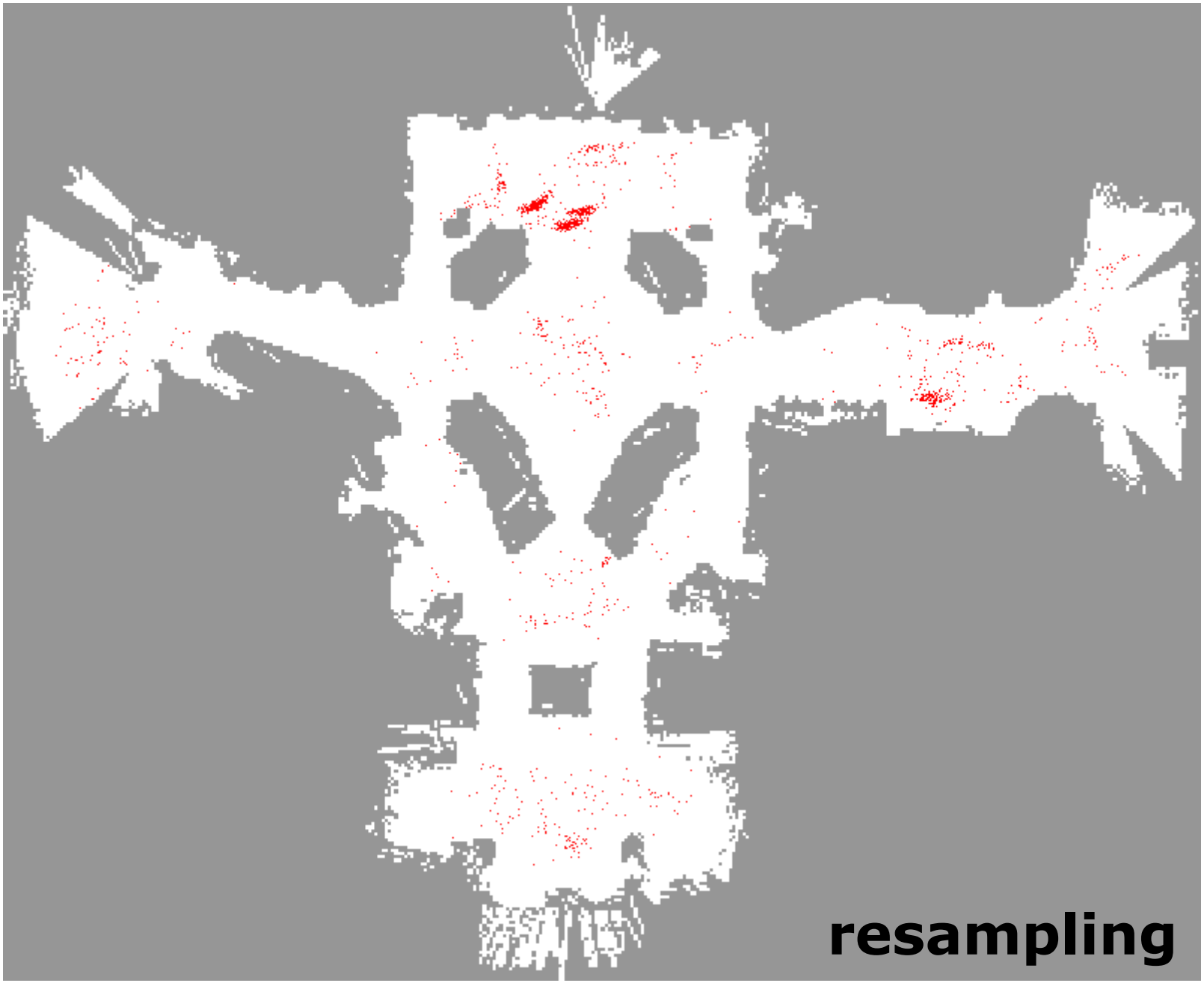

observation

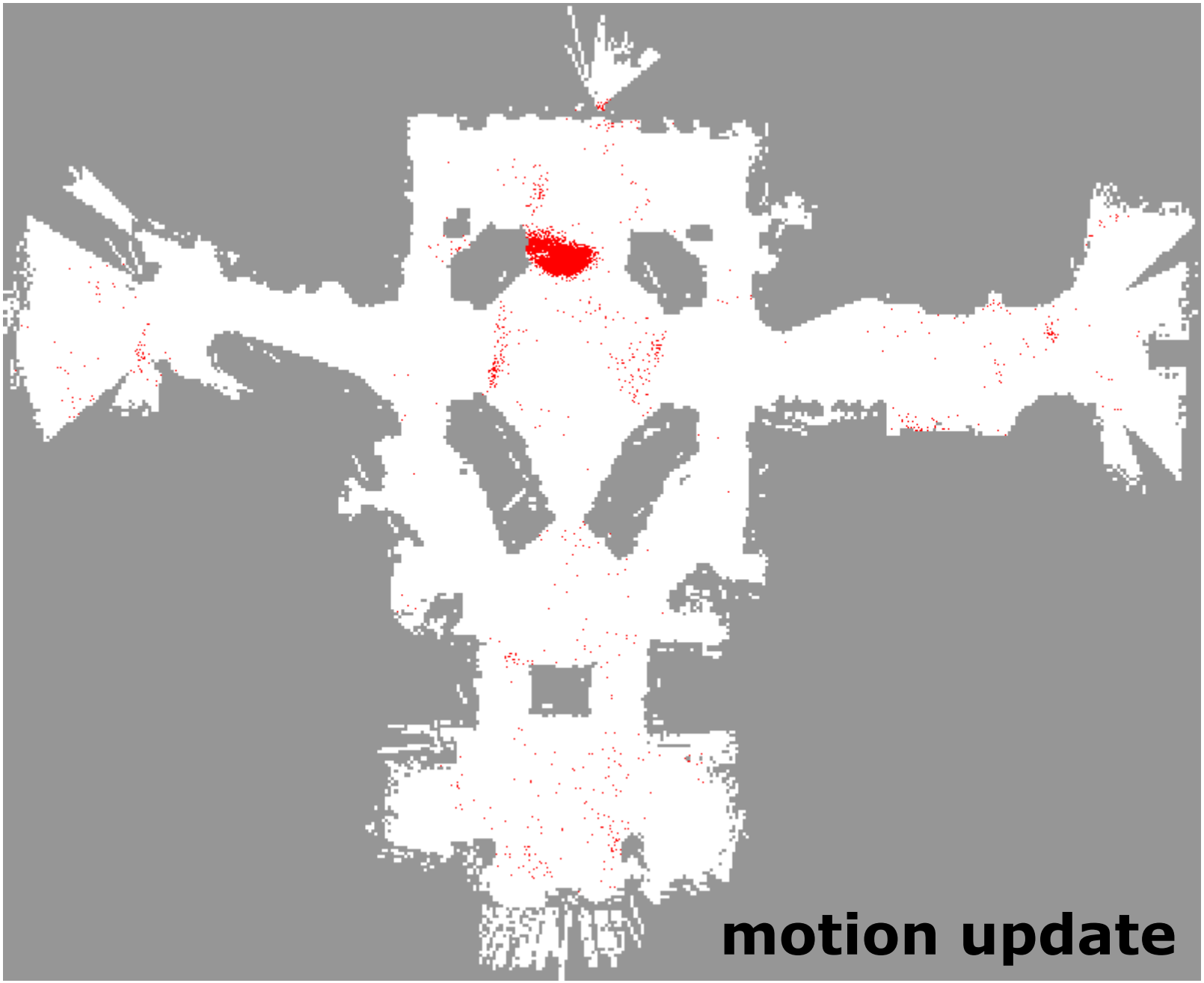

resampling

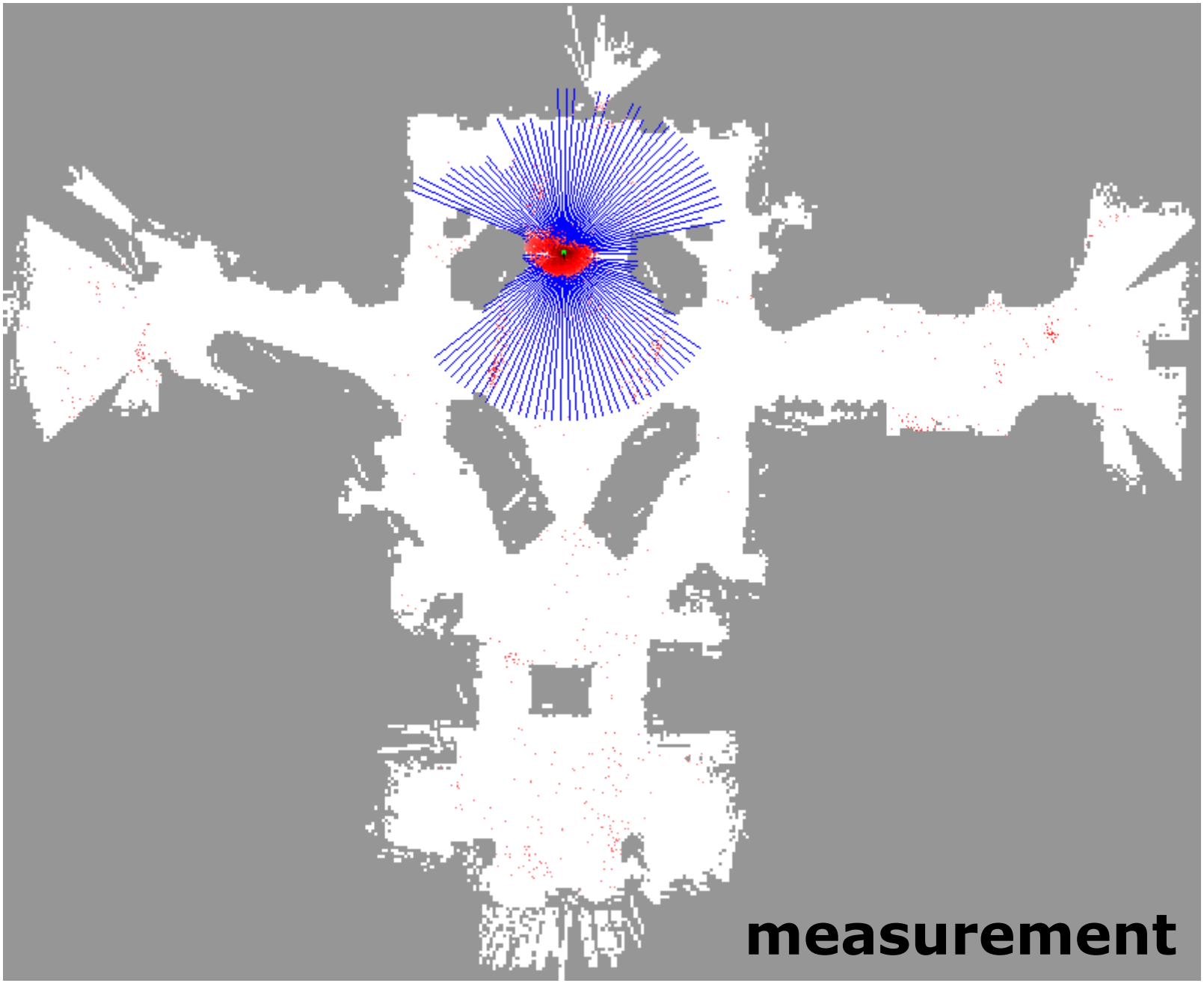

motion update


measurement

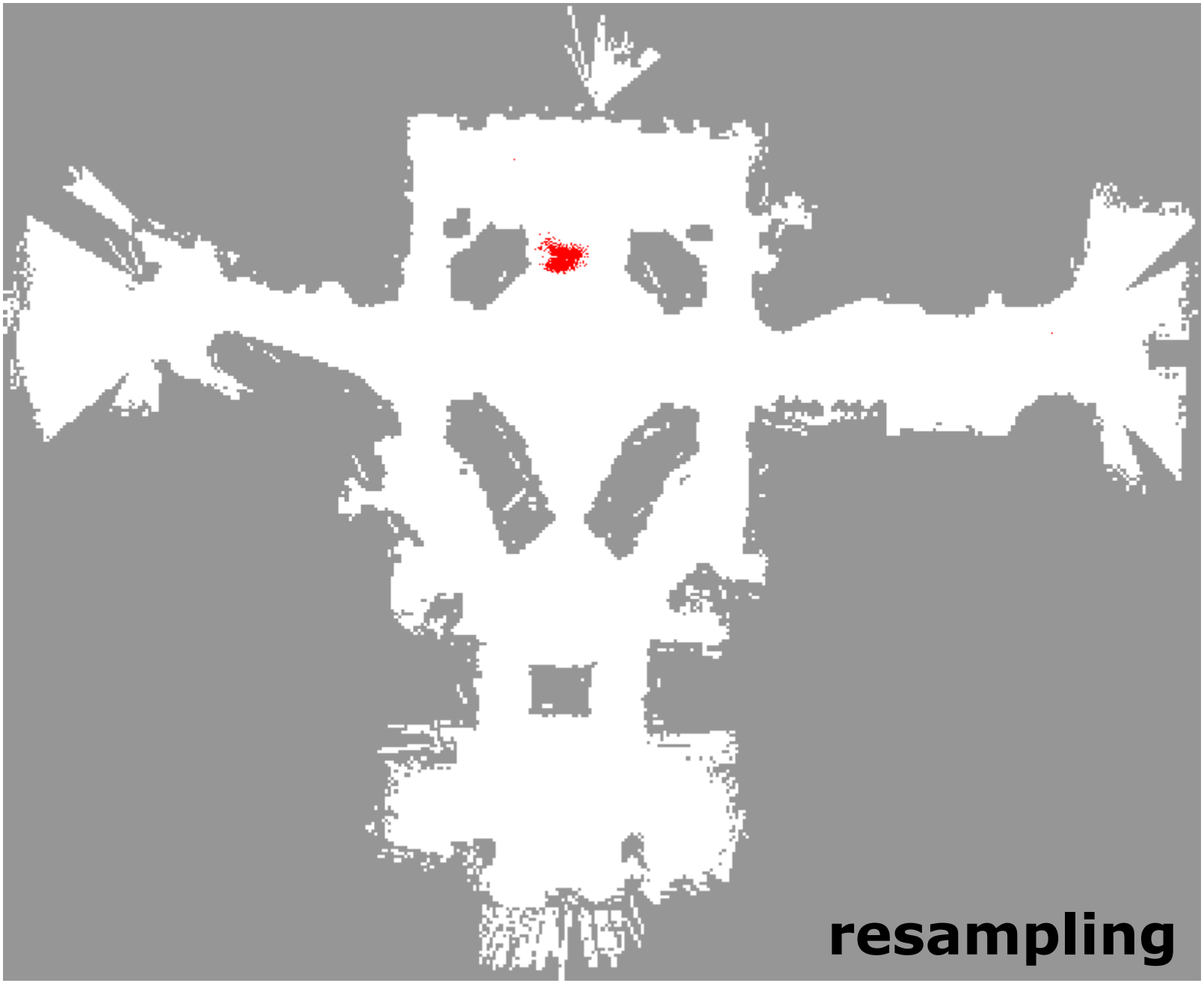

weight update

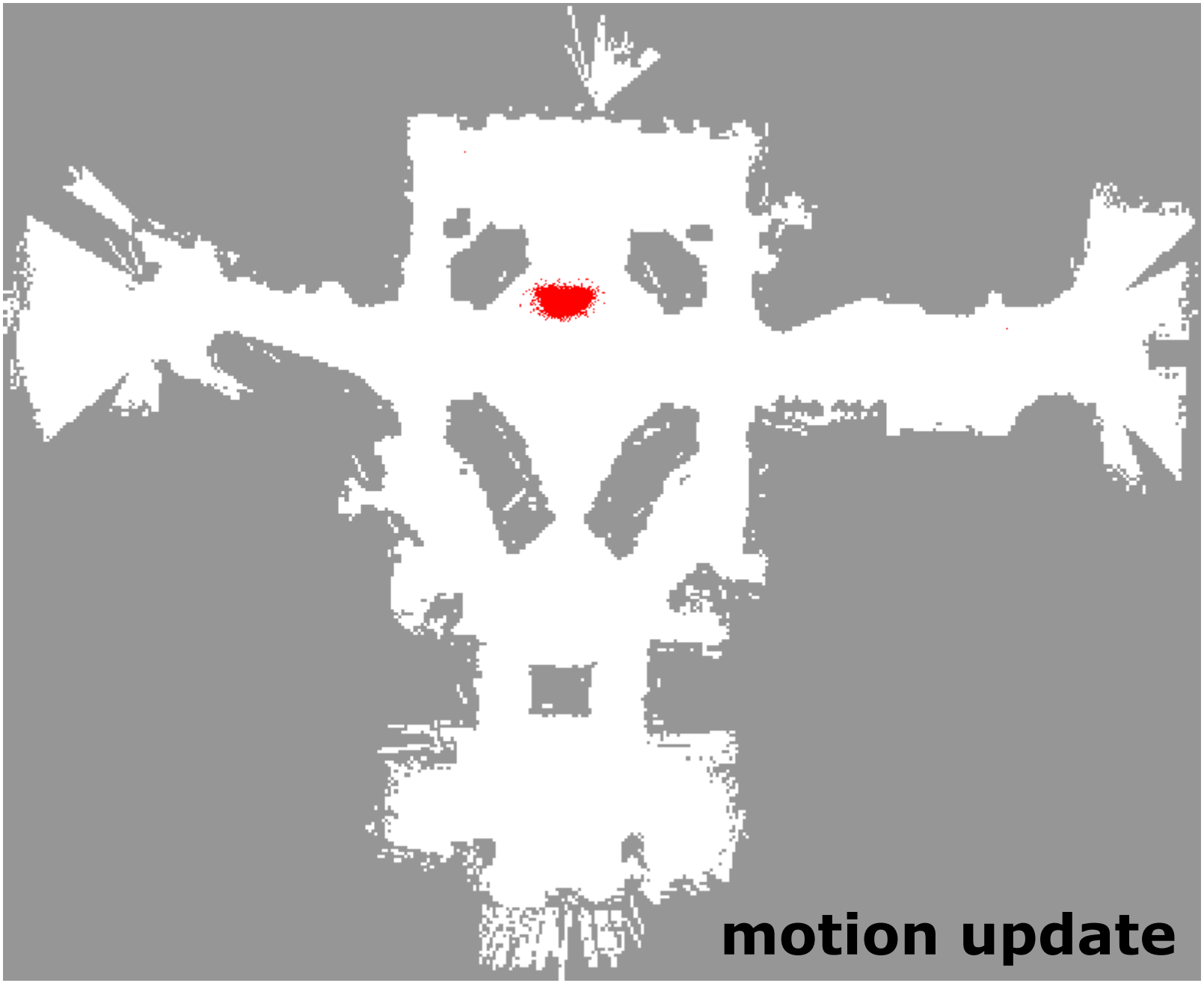

resampling

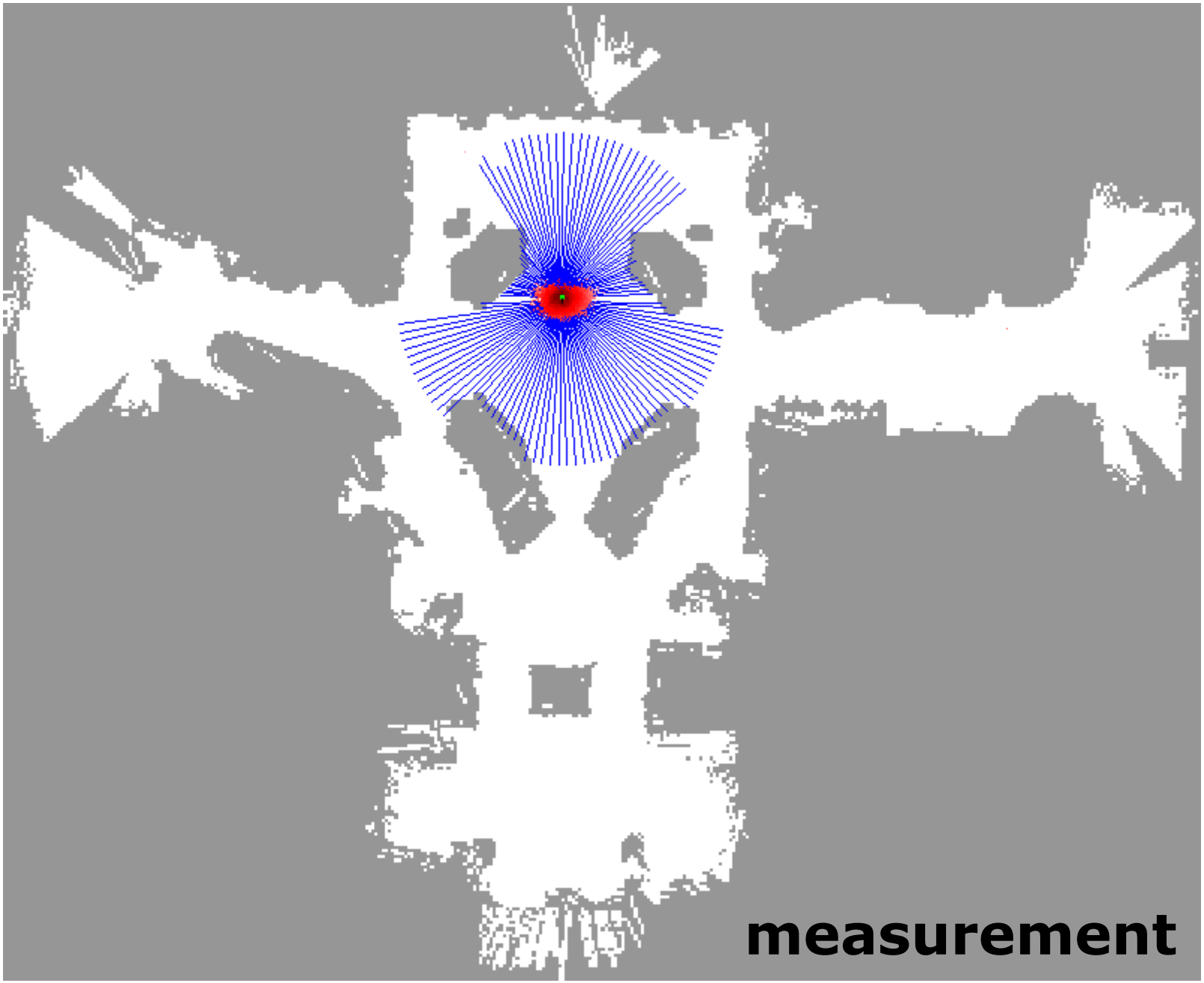

motion update

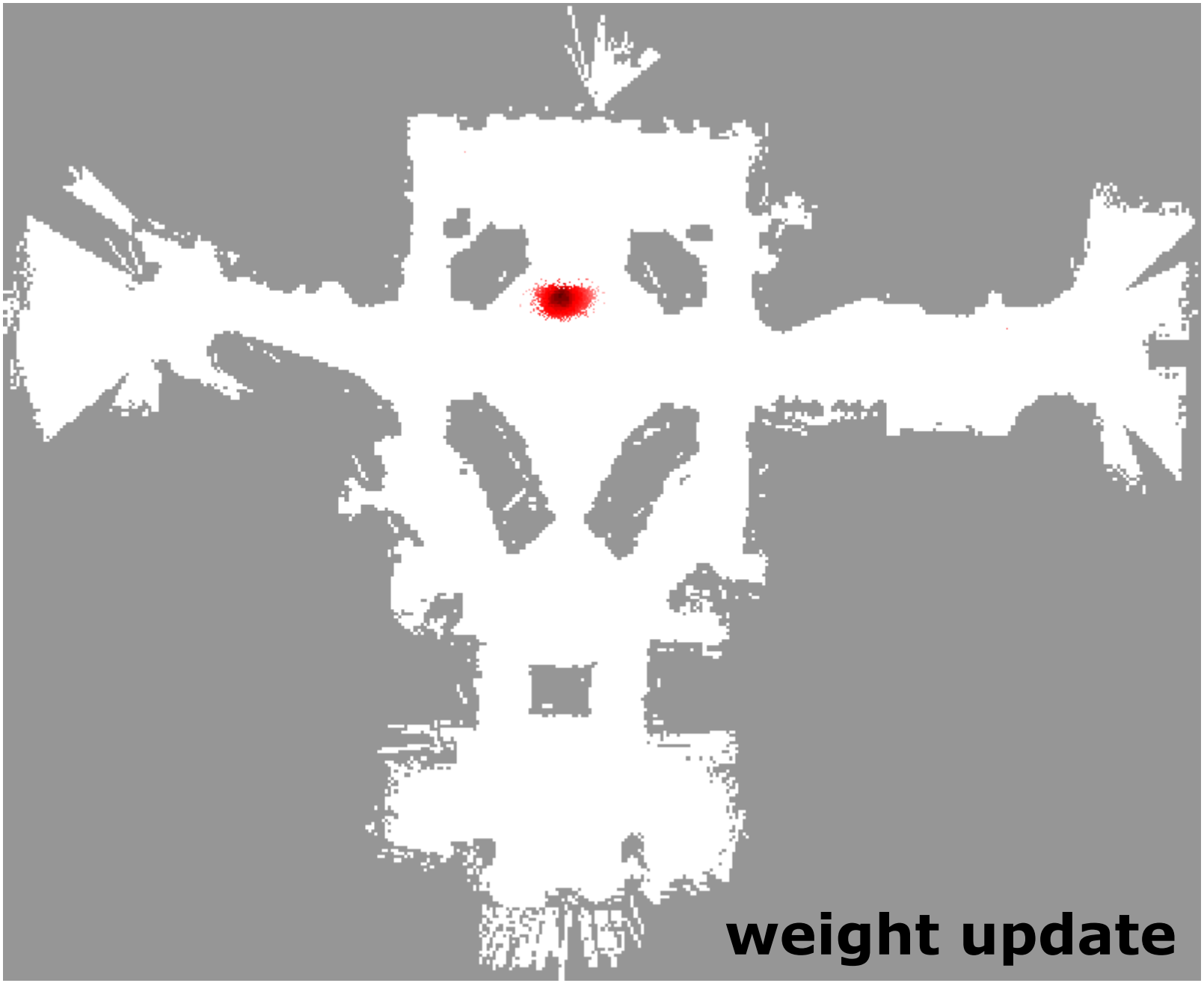

measurement

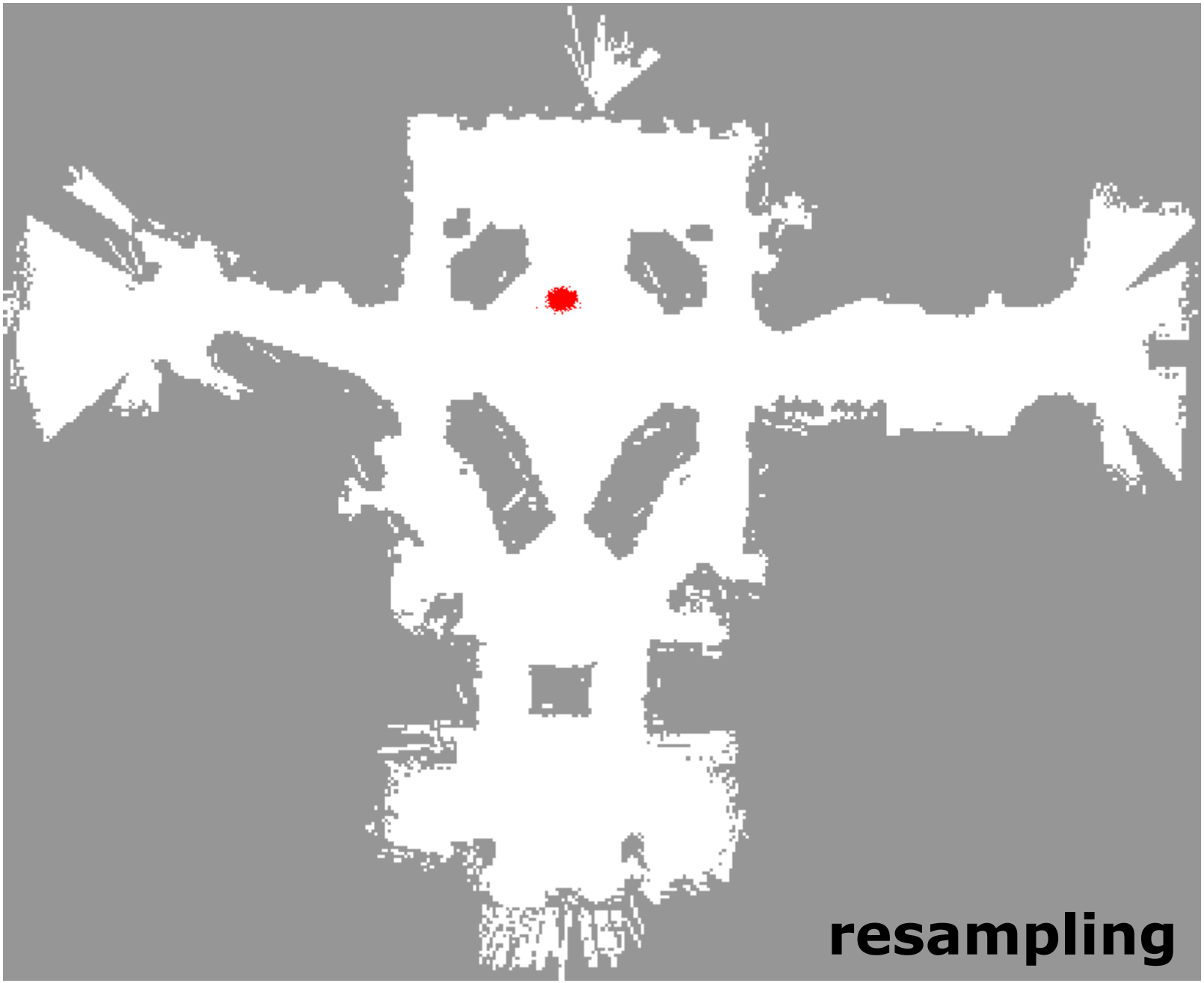

weight update

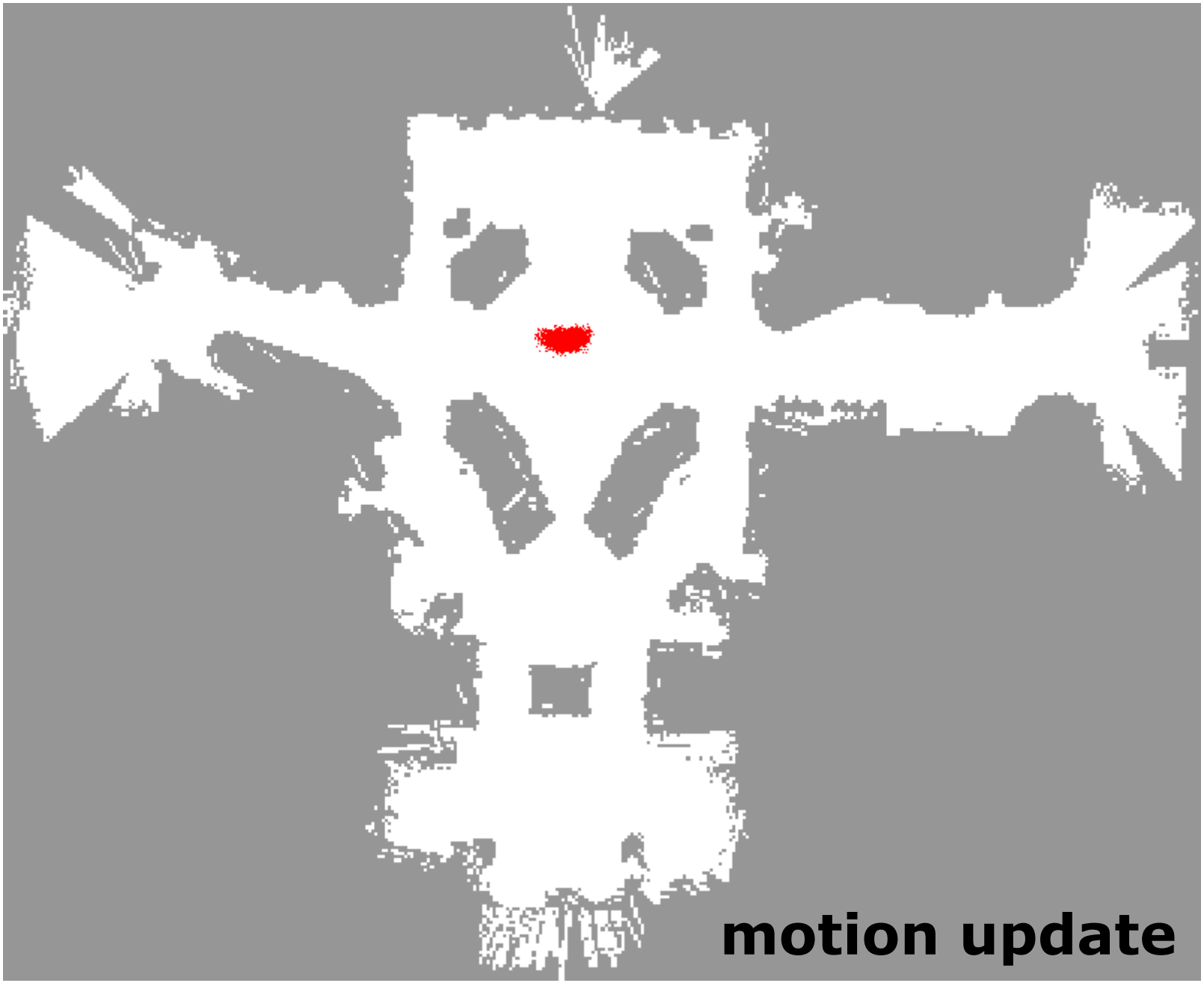

resampling

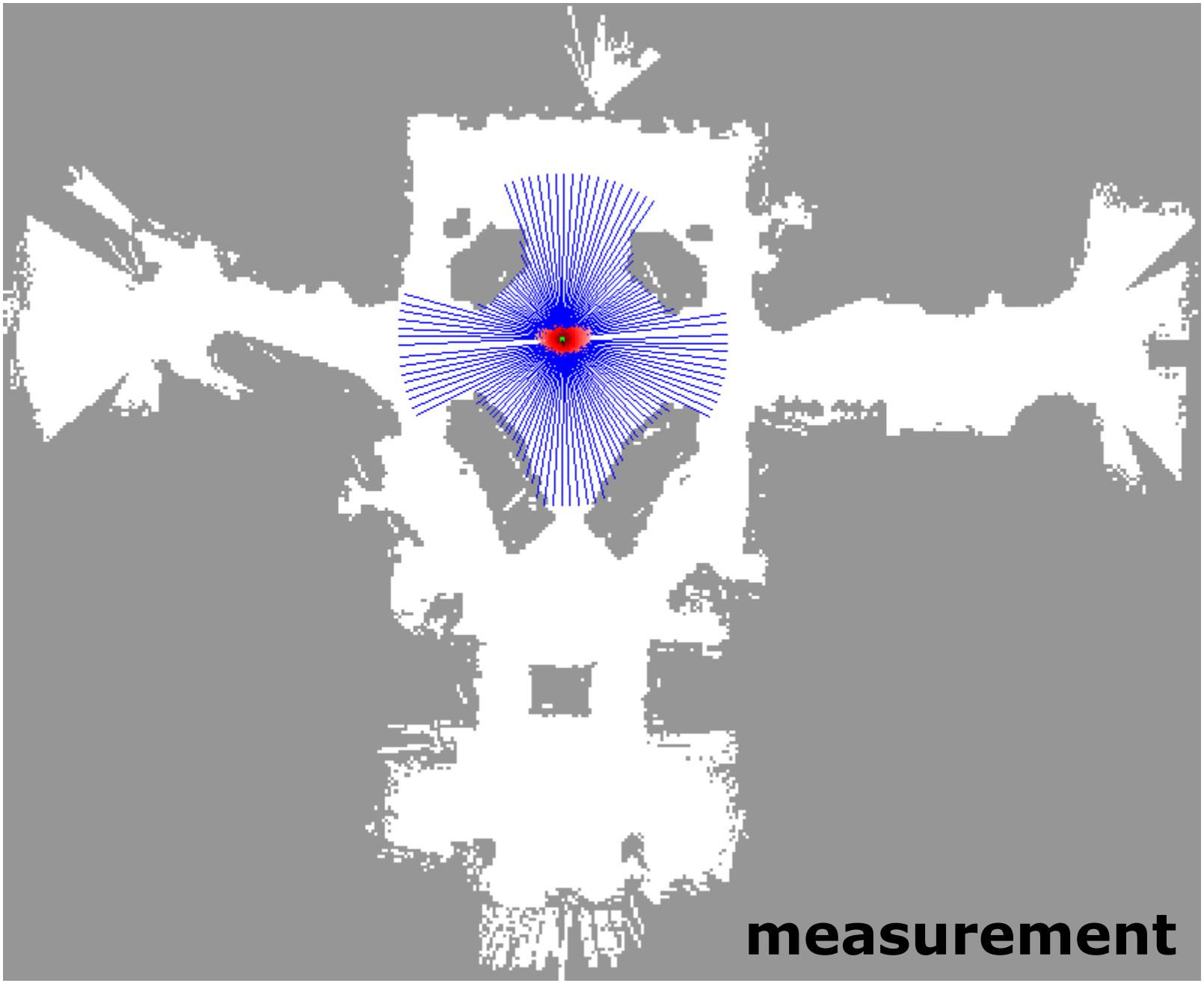

motion update


measurement


weight update


resampling


motion update


measurement

Summary – Particle Filters

- Particle filters are non-parametric, recursive Bayes filters
- Posterior is represented by a set of weighted samples
- Not limited to Gaussians
- Proposal to draw new samples
- Weight to account for the differences between the proposal and the target
- Work well in low-dimensional spaces

Summary – PF Localization

- Particles are propagated according to the motion model
- They are weighted according to the likelihood of the observation
- Called: Monte-Carlo localization (MCL)
- MCL is the gold standard for mobile robot localization today

ESE 650
Learning in Robotics

Spring 2019

Rigid Transforms

- A rigid transformation, g, from $\mathbb{R}^3$ to $\mathbb{R}^3$ satisfies the following properties for all $u, v, w \in \mathbb{R}^3$
  - $$\|g(u) - g(v)\| = \|u - v\|$$

    Where $\|x\| = \sqrt{x^T x}$
  - $$g((u - w) \times (v - w)) = (g(u) - g(w)) \times (g(v) - g(w))$$

Cross Product

- The cross product of two vectors $u, v \in \mathbb{R}^3$ produces a new vector orthogonal to the first two by the right hand rule with magnitude given by $\sin\theta\|u\|\|v\|$. Where θ denotes the angle between the two vectors
- Algebraically the cross product in $\mathbb{R}^3$ can be computed as follows.

$$a \times b = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} = \hat{a}b$$

- $\hat{a}$ denotes the skew symmetric matrix in $\mathbb{R}^{3\times 3}$ derived from the entries of the vector $a \in \mathbb{R}^3$. Note $\mathbb{R}^3$ is the only vector space where there is a one to one mapping from the vector space to the set of corresponding skew symmetric matrices.

$$\hat{a} = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix}$$

Skew Symmetric Matrices

- Where symmetric matrices are defined by the equation

$$A^T = A, \;\; A \in \mathbb{R}^{n \times n}$$

  Skew symmetric matrices are defined by the equation.

$$A^T = -A, \;\; A \in \mathbb{R}^{n \times n}$$

- You can think of them as the symmetric matrices evil twin with equal and opposite properties
- Note that every square matrix $B \in \mathbb{R}^{n \times n}$ can be expressed as a sum of symmetric and skew symmetric parts as follows.

$$B = \frac{1}{2}(B + B^T) + \frac{1}{2}(B - B^T)$$

Rigid Transforms

- Let's consider the set of rigid transformations where $g(0) = 0$, that is the origin is preserved.
- Since g preserves the Euclidean norm we have:

$$\|g(x) - g(0)\| = \|x - 0\| \Rightarrow \|g(x) - 0\| = \|x - 0\| \Rightarrow \|g(x)\| = \|x\|$$

- Now let's consider two vectors $x, y \in \mathbb{R}^3$

$$\|g(x) - g(y)\|^2 = \|x - y\|^2$$

Observe that

$$\|a - b\|^2 = (a - b)^T(a - b) = a^t a - 2a^t b + b^t b = \|a\|^2 + \|b\|^2 - 2a^t b$$

Expanding both sides of the first equation yields

$$\|g(x)\|^2 + \|g(y)\|^2 - 2g(x)^t g(y) = \|x\|^2 + \|y\|^2 - 2x^t y$$

$$\Rightarrow g(x)^t g(y) = x^t y$$

Since $\|g(x)\| = \|x\|$ and $\|g(y)\| = \|y\|$

- So we conclude that rigid transformations that preserve the origin also preserve inner products

Rigid Transformations

- Let $\{e_1, e_2, e_3\}$ be the standard orthonormal basis for $\mathbb{R}^3$. Since the transform g preserves inner products we can conclude that $\{g(e_1), g(e_2), g(e_3)\}$ must also be an orthonormal basis for $\mathbb{R}^3$.

- Further if $x = \sum_{i=1}^{3} x_i e_i$ then $g(x) = \sum_{i=1}^{3} x_i g(e_i)$ since $g(x)^t g(e_i) = x^t e_i = x_i$

- This means that:

$$g(x) = \sum_{i=1}^{3} x_i g(e_i) = \begin{pmatrix} g(e_1) & g(e_2) & g(e_3) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = Rx$$

Where $R = \begin{pmatrix} g(e_1) & g(e_2) & g(e_3) \end{pmatrix} \in \mathbb{R}^{3\times 3}$ is a matrix with orthonormal columns

Determinant

- Remember that the transformation g also preserves cross products

$$(g(u) - g(w)) \times (g(v) - g(w)) = g((u - w) \times (v - w))$$

Let $w = 0$ and $g(w) = 0$ and we get

$$g(u) \times g(v) = g(u \times v) \quad \forall u, v \in \mathbb{R}^3$$

- Now

$$\begin{aligned} \det(R) &= \det\left(\begin{pmatrix} g(e_1) & g(e_2) & g(e_3) \end{pmatrix}\right) \\ &= g(e_1)^T (g(e_2) \times g(e_3)) \\ &= g(e_1)^T (g(e_2 \times e_3)) \\ &= e_1^T (e_2 \times e_3) \\ &= \det\left(\begin{pmatrix} e_1 & e_2 & e_3 \end{pmatrix}\right) \\ &= 1 \end{aligned} \tag{1}$$

SO(3)

- The set of all orthonormal matrices in $\mathbb{R}^{3\times 3}$ with determinant $+1$ is referred to as $SO(3)$

$$SO(3) = \{R \in \mathbb{R}^{3\times 3} \mid R^T R = I,\ \det(R) = +1\}$$

- Elements of $SO(3)$ represent rotations about an axis.
- $SO(3)$ is a group with respect to the operation of matrix multiplication.

Groups

A *group* is an algebraic structure composed of a set, G, and a group operation, $\cdot$, that satisfies the following axioms:

- **Closure** : $a \cdot b \in G \ \ \forall a, b \in G$
- **Associativity** : $a \cdot (b \cdot c) = (a \cdot b) \cdot c \ \ \forall a, b, c \in G$
- **Identity and Inverse** : $\exists e \in G$ such that:

$$a \cdot e = a \ \ \forall a \in G$$

and

$$\forall a \in G \ \ \exists a^{-1} \in G \ \text{ such that } \ a \cdot a^{-1} = e$$

The element e is referred to as an *identity* element.

Orthonormal Matrices

- Consider the case of a matrix $Q \in \mathbb{R}^{m \times n}$ where the columns of Q form an orthonormal basis.

$$Q = [\hat{\mathbf{u}}_1, \hat{\mathbf{u}}_2, \cdots \hat{\mathbf{u}}_n]$$

- From our description of the vectors $\hat{\mathbf{u}}_i$ we can conclude that.

$$Q^T Q = I_n$$

  Since the entries in $Q^T Q$ correspond to the inner products of the columns of Q.

- In the special case where Q is a square matrix , $Q \in \mathbb{R}^{n \times n}$, we can conclude that $Q^{-1} = Q^T$. That is, the matrix is invertible and its inverse is its transpose. We like these kinds of matrices because they are simple to deal with.

$$Q\mathbf{x} = \mathbf{b} \;\Rightarrow\; \mathbf{x} = Q^T \mathbf{b}$$

Orthonormal Matrices

- Square orthonormal matrices actually form a *subgroup* of the group of invertible $n \times n$ matrices denoted by $O(n) \subset GL(n)$.

- To prove that square orthogonal matrices are in fact a subgroup all we have to show is that the set contains the identity, trivial, and that the set is closed under matrix multiplication.

$$
\begin{aligned}
Q_1, Q_2 \in O(n) & & & (1)\\
(Q_1 Q_2)^T (Q_1 Q_2) &= Q_2^T Q_1^T Q_1 Q_2 & & (2)\\
&= Q_2^T I_n Q_2 & & (3)\\
&= Q_2^T Q_2 & & (4)\\
&= I_n & & (5)
\end{aligned}
$$

So if Q_1 and Q_2 are both square orthonormal matrices then their product will be as well.

Orthonormal Matrices

- Square orthonormal matrices preserve the Euclidean norm:

$$\mathbf{x} = Q\mathbf{y} \;\Rightarrow\; Q^T\mathbf{x} = \mathbf{y}$$

$$\begin{aligned}
\|\mathbf{x}\|^2 &= \mathbf{x}^T\mathbf{x} && (1)\\
&= (Q\mathbf{y})^T(Q\mathbf{y}) && (2)\\
&= \mathbf{y}^T Q^T Q\mathbf{y} && (3)\\
&= \mathbf{y}^T I\mathbf{y} && (4)\\
\|\mathbf{x}\|^2 &= \|\mathbf{y}\|^2 && (5)
\end{aligned}$$

Alternative view of Rotations

- Consider a point $r(t) \in \mathbb{R}^3$ that's being rotated around an axis denoted by a unit vector $\omega \in \mathbb{R}^3$ with an angular velocity of 1 radian per second.
- The instantaneous linear velocity of the point is given by

$$\dot{r}(t) = \omega \times r(t) = \hat{\omega} r(t)$$

- This is a linear differential equation which can be solved as follows.

$$r(t) = \exp(\hat{\omega} t) r(0)$$

  Substituting θ for t yields.

$$r(\theta) = \exp(\hat{\omega} \theta) r(0)$$

- Definition of matrix exponential

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \frac{A^4}{4!} + \cdots = \sum_{i=0}^{\infty} \frac{A^i}{i!}$$

Alternative view of Rotations

- A rotation about a fixed axis ω by an angle θ can be represented by the matrix

$$R = \exp(\hat{\omega}\theta)$$

- Proving that $R \in SO(3)$

$$\begin{aligned} R^T R &= \exp(\hat{\omega}\theta)^T \exp(\hat{\omega}\theta) \\ &= \exp(\hat{\omega}^T\theta)\exp(\hat{\omega}\theta) \\ &= \exp(-\hat{\omega}\theta)\exp(\hat{\omega}\theta) \\ &= \exp(-\hat{\omega}\theta + \hat{\omega}\theta) \\ &= \exp(0) \\ &= I \end{aligned}$$

Note that the last step is possible because $\exp(A)\exp(B) = \exp(A+B)$ iff $AB = BA$ which is clearly true for $\hat{\omega}$ and $-\hat{\omega}$

Alternative view of Rotations

- A rotation about a fixed axis ω by an angle θ can be represented by the matrix

$$R = \exp(\hat{\omega}\theta)$$

- Proving that $R \in SO(3)$ part 2.

  Since $R^T R = I$ we can easily conclude that $\det(R)^2 = 1$. To show that $\det(R) = +1$ we note that $\det(\exp(\hat{\omega}\theta))$ is a continuous function of θ and that when $\theta = 0$, $R = I$ and $\det(R) = +1$. So we conclude that it must remain $+1$ for all θ by continuity.

Alternative view of Rotations

- Expanding the exponential

$$R = \exp(\hat{\omega}\theta) = I + \hat{\omega}\theta + \frac{\hat{\omega}^2\theta^2}{2} + \frac{\hat{\omega}^3\theta^3}{3!} + \cdots$$

- In general

$$\hat{a}\hat{b} = ba^t - a^t bI$$

  Note that $\omega^t\omega = 1$ so

$$\hat{\omega}^2 = \omega\omega^T - I$$

  and

$$\hat{\omega}^3 = \hat{\omega}(\omega\omega^T - I) = -\hat{\omega}$$

- More generally

$$\hat{\omega}^{2i} = \hat{\omega}^2(-1)^{i+1}$$

$$\hat{\omega}^{2i+1} = \hat{\omega}(-1)^{i+1}$$

- This means that R can be expressed as follows grouping odd and even powers of $\hat{\omega}$.

$$R = I + (\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!}\cdots)\hat{\omega} + (\frac{\theta^2}{2!} - \frac{\theta^4}{4!} + \frac{\theta^6}{6!} - \frac{\theta^8}{8!}\cdots)\hat{\omega}^2$$

Rodrigues Formula

- This means that $R = \exp(\hat{\omega}\theta)$ can be expressed as follows grouping odd and even powers of $\hat{\omega}$.

$$R = I + (\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} \cdots)\hat{\omega} + (\frac{\theta^2}{2!} - \frac{\theta^4}{4!} + \frac{\theta^6}{6!} - \frac{\theta^8}{8!} \cdots)\hat{\omega}^2$$

- This expansion leads to the Rodrigues formula that relates an angle θ and axis ω to the corresponding rotation matrix.

$$R = I + \sin\theta\ \hat{\omega} + (1 - \cos\theta)\ \hat{\omega}^2$$

Recovering angle axis

- Given a rotation matrix $R \in SO(3)$ we can recover the corresponding angle and axis using the Rodrigues formula as follows.

$$R = I + \sin\theta\, \hat{\omega} + (1 - \cos\theta)\, \hat{\omega}^2$$

- $\text{trace}(R) = 1 + 2\cos\theta$: We can compute the angle θ from this.
- $R - R^T = 2\sin\theta\, \hat{\omega}$: We can recover the axis ω from this skew symmetric matrix.

Euler's Theorem

- Euler's theorem states that every matrix $R \in SO(3)$ can be written as $\exp(\hat{\omega}\theta)$ for some choice of ω and θ.

Unit Quaternions

- Unit quaternions provide an alternative representation of rotations.
- The set of unit quaternions can be thought of as the set of tuples (u_0, u) where $u_0 \in \mathbb{R}$ and $u \in \mathbb{R}^3$ such that $u_0^2 + u^t u = 1$.
- This set forms a group under the operation of quaternion multiplication defined as follows.

$$(u_0, u) \cdot (v_0, v) = (u_0 v_0 - u^T v, u_0 v + v_0 u + u \times v)$$

  This group is referred to as the Symplectic Group $Sp(1)$.
- There is a 2 to 1 mapping between elements of $Sp(1)$ and $SO(3)$ which can be defined by the following mapping.

$$H(u_0, u) = (u_0^2 - u^T u)I + 2u_0\hat{u} + 2uu^T$$

  You can verify that $H(u_0, u) \in SO(3)$. Note that (u_0, u) and $(-u_0, -u)$ map to the same matrix.

Quaternions

- Given a quaternion, $q = (u_0, u)$, we can define its conjugate as follows $q^* = (u_0, -u)$.
- Given a vector $x \in \mathbb{R}^3$ we can form a quaternion $(0, x)$.
- You can show that

$$q \cdot (0, x) \cdot q^* = (0, H(q)x)$$

- You can also show that

$$H(q_1 \cdot q_2) = H(q_1)H(q_2)$$

- This means that we can represent rotations using quaternions, 4 numbers instead of 9, and easier to normalize. We can then perform quaternion multiplications instead of matrix multiplications.
- Lastly we have the following important relationship.

$$q = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\omega) \Rightarrow H(q) = \exp(\hat{\omega}\theta)$$

Summary

- We have discussed three different ways of representing rotations
  - As matrices $R \in SO(3) \subset R^{3\times 3}$, $R^T R = I$, $\det(R) = 1$
  - Angle axis representations $\theta \in \mathbb{R}, \omega \in \mathbb{R}^3$
  - Unit quaternions (u_0, u) such that $u_0^2 + u^T u = 1$
- These representations are related by the Rodrigues formula

$$R = \exp(\hat{\omega}\theta) = I + \sin\theta\hat{\omega} + (1 - \cos\theta)\hat{\omega}^2$$

and by the equations

$$q = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\omega) \Rightarrow H(q) = \exp(\hat{\omega}\theta)$$

and

$$H(u_0, u) = (u_0^2 - u^T u)I + 2u_0\hat{u} + 2uu^T$$

Euler Angles

- Any rotation matrix can be represented as the product of three successive rotations around the coordinate axis

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

$$R_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Note that there is no way to construct a one to one mapping between triples of angles and rotation matrices. Such parameterizations will always involve singularities because the spaces are topologically different.

SE(3)

- Consider a rigid transform, g, such that $g(0) = t$. If we consider $h(x) = g(x) - t$. This will also be a rigid transformation and $h(0) = 0$. This means that $h(x) = Rx$ for some $R \in SO(3)$.

- This allows us to conclude that every rigid transformation can be written in the following form

$$g(x) = Rx + t$$

  Where $R \in SO(3)$ and $t \in \mathbb{R}^3$.

- The set of rigid transformations of $\mathbb{R}^3$ is referred to as $SE(3)$ The Special Euclidean group on $\mathbb{R}^3$

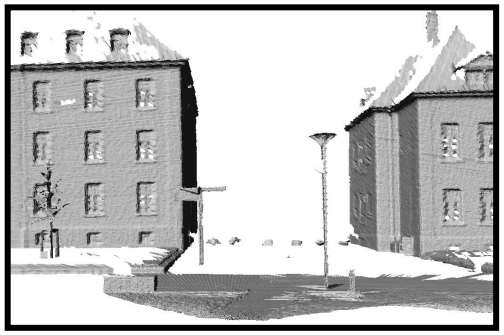
Robot Mapping

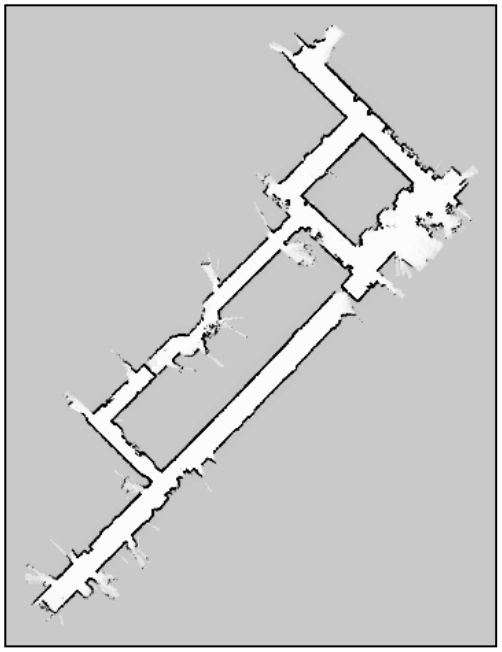
Grid Maps

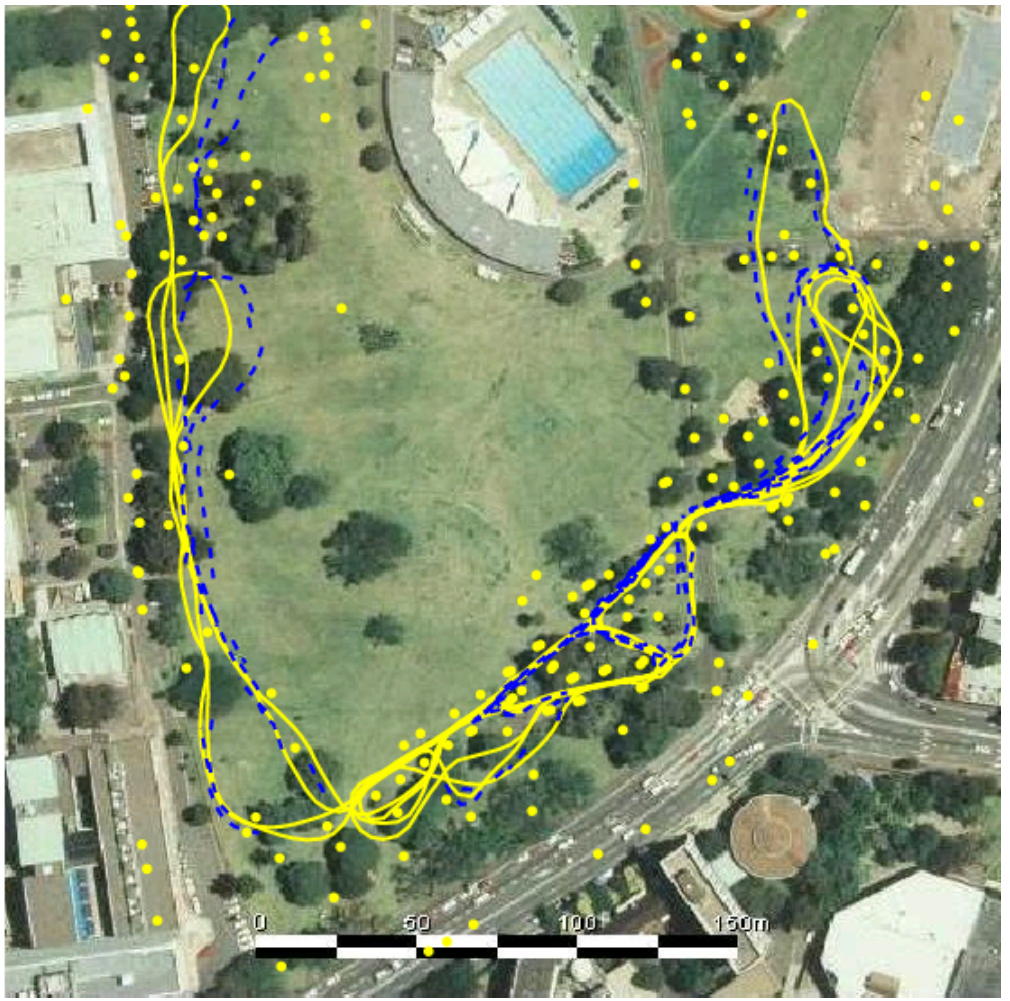
Cyrill Stachniss

AIS Autonomous Intelligent Systems

Features vs. Volumetric Maps







Courtesy by E. Nebot

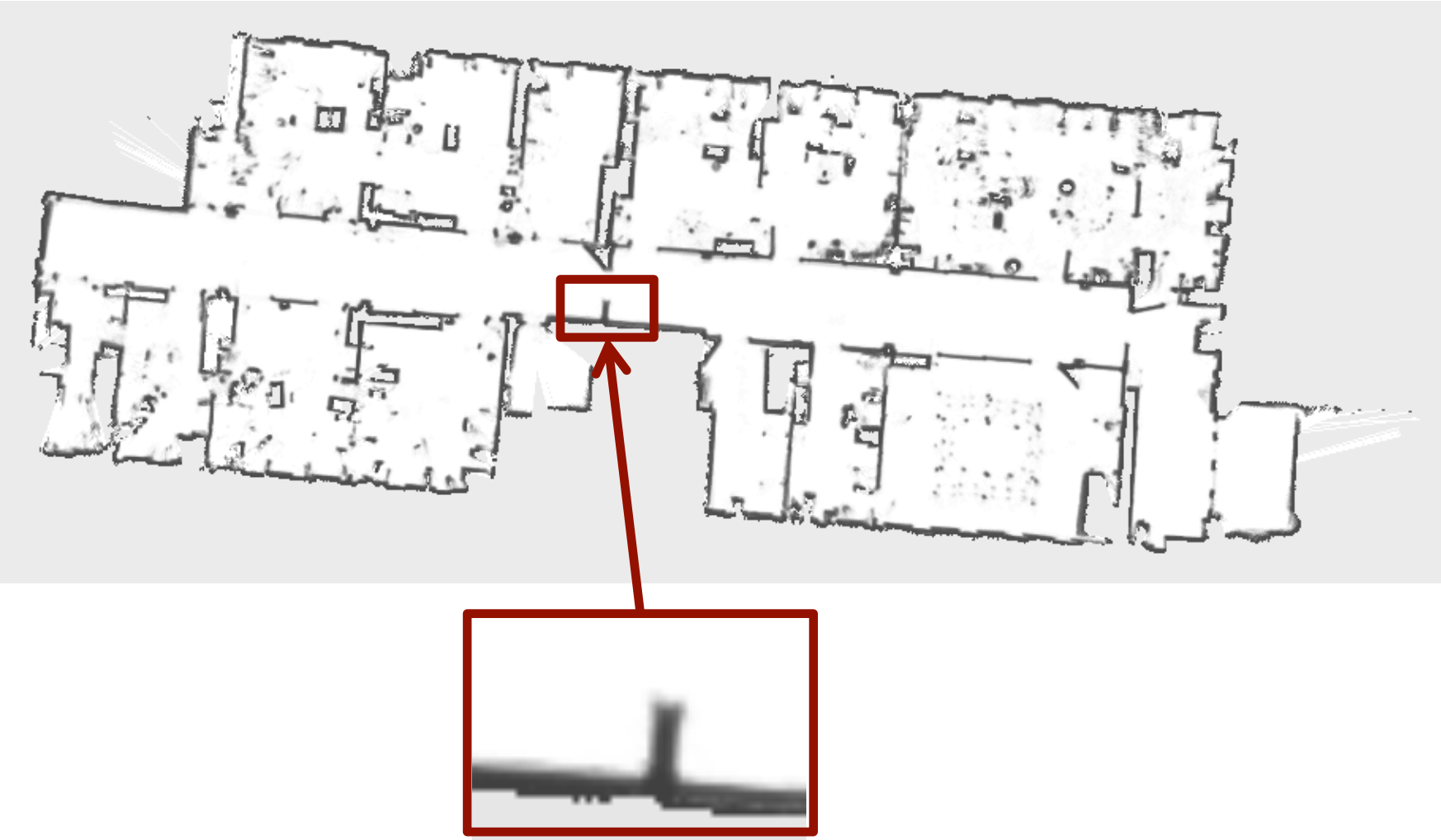
Features

- So far, we only used feature maps
- Natural choice for Kalman filter-based SLAM systems
- Compact representation
- Multiple feature observations improve the position estimate (EKF)

Grid Maps

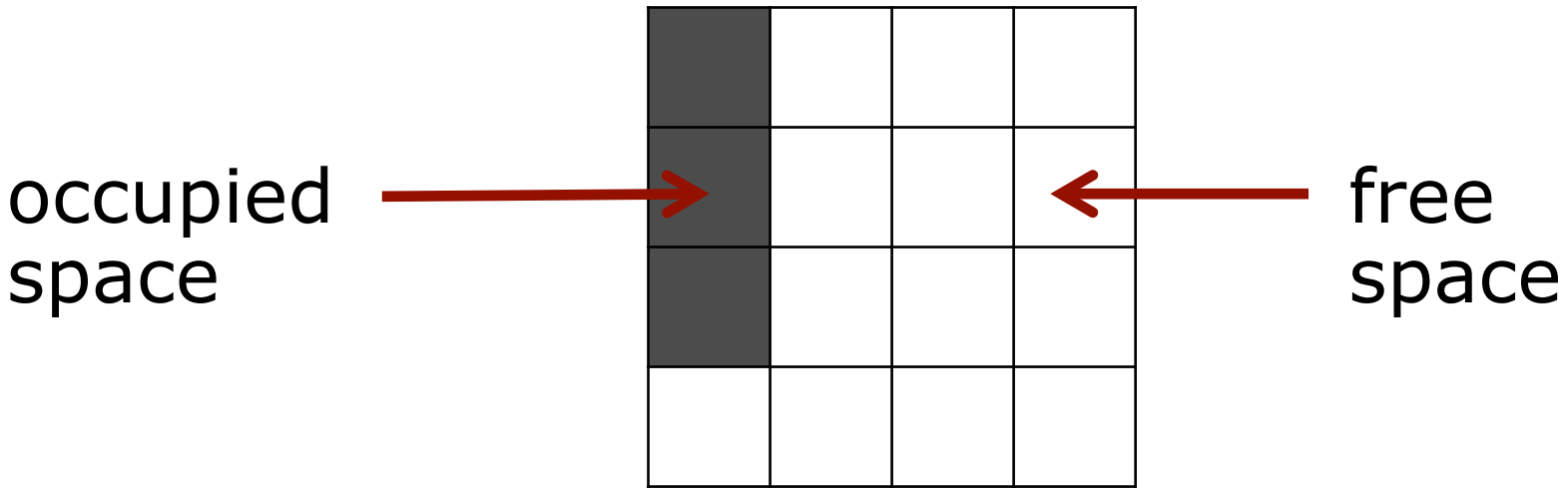
- Discretize the world into cells
- Grid structure is rigid
- Each cell is assumed to be occupied or free space
- Non-parametric model
- Require substantial memory resources
- Does not rely on a feature detector

Example



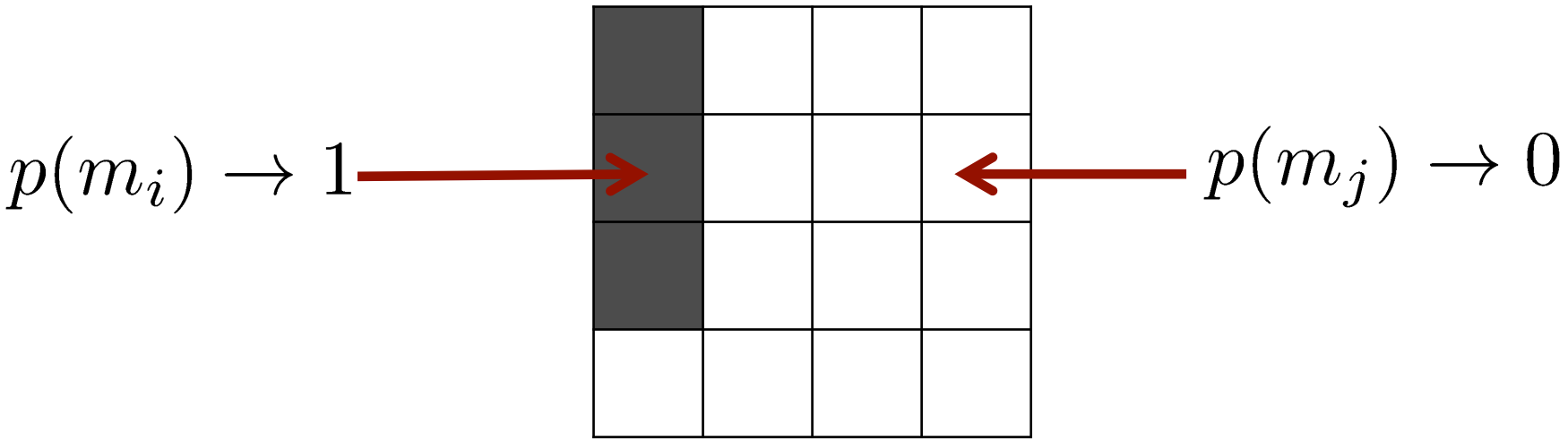
Assumption 1

- The area that corresponds to a cell is either completely free or occupied



Representation

- Each cell is a **binary random variable** that models the occupancy



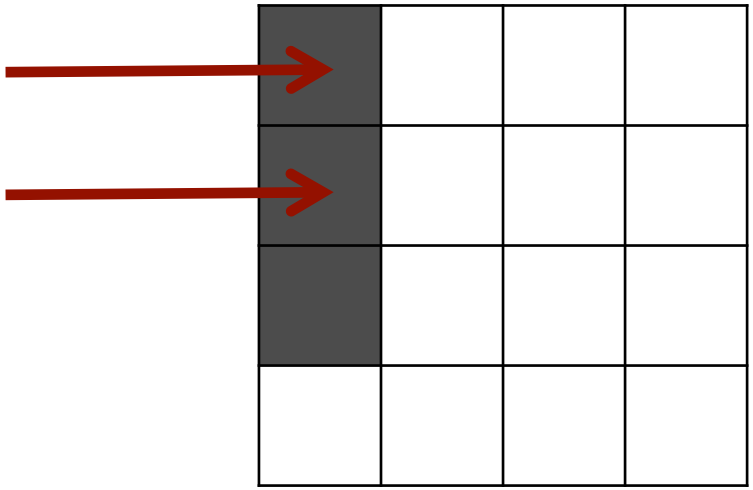
Occupancy Probability

- Each cell is a **binary random variable** that models the occupancy
- Cell is occupied $p(m_i) = 1$
- Cell is not occupied $p(m_i) = 0$
- No knowledge $p(m_i) = 0.5$
- The **state** is assumed to be **static**

Assumption 2

- The cells (the random variables) are **independent** of each other

no dependency
between the cells



Representation

- The probability distribution of the map is given by the product over the cells

$$p(m) = \prod_i p(m_i)$$

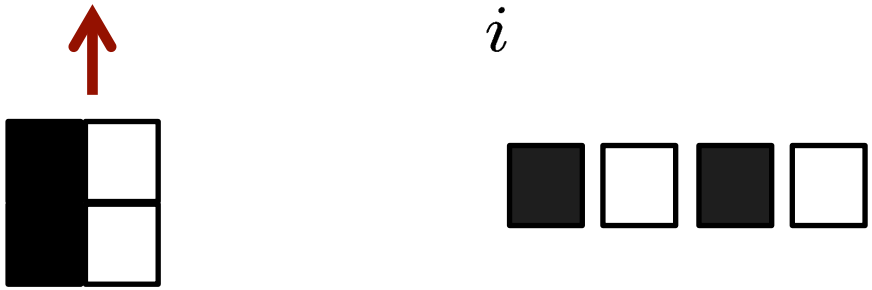
map

cell

Representation

- The probability distribution of the map is given by the product over the cells

$$p(m) = \prod_i p(m_i)$$



example map
(4-dim vector)

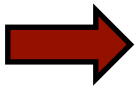
4 individual cells

Estimating a Map From Data

- Given sensor data $z_{1:t}$ and the poses $x_{1:t}$ of the sensor, estimate the map

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i p(m_i \mid z_{1:t}, x_{1:t})$$

binary random variable



Binary Bayes filter
(for a static state)

Static State Binary Bayes Filter

$$p(m_i \mid z_{1:t}, x_{1:t}) \overset{\text{Bayes rule}}{=} \frac{p(z_t \mid m_i, z_{1:t-1}, x_{1:t})\, p(m_i \mid z_{1:t-1}, x_{1:t})}{p(z_t \mid z_{1:t-1}, x_{1:t})}$$

Static State Binary Bayes Filter

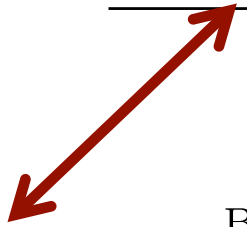
$$p(m_i \mid z_{1:t}, x_{1:t}) \overset{\text{Bayes rule}}{=} \frac{p(z_t \mid m_i, z_{1:t-1}, x_{1:t})\, p(m_i \mid z_{1:t-1}, x_{1:t})}{p(z_t \mid z_{1:t-1}, x_{1:t})}$$

$$\overset{\text{Markov}}{=} \frac{p(z_t \mid m_i, x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(z_t \mid z_{1:t-1}, x_{1:t})}$$

Static State Binary Bayes Filter

$$p(m_i \mid z_{1:t}, x_{1:t}) \overset{\text{Bayes rule}}{=} \frac{p(z_t \mid m_i, z_{1:t-1}, x_{1:t})\, p(m_i \mid z_{1:t-1}, x_{1:t})}{p(z_t \mid z_{1:t-1}, x_{1:t})}$$

$$\overset{\text{Markov}}{=} \frac{p(z_t \mid m_i, x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(z_t \mid z_{1:t-1}, x_{1:t})}$$



$$p(z_t \mid m_i, x_t) \overset{\text{Bayes rule}}{=} \frac{p(m_i \mid z_t, x_t)\, p(z_t \mid x_t)}{p(m_i \mid x_t)}$$

Static State Binary Bayes Filter

$$
\begin{aligned}
p(m_i \mid z_{1:t}, x_{1:t}) &\overset{\text{Bayes rule}}{=} \frac{p(z_t \mid m_i, z_{1:t-1}, x_{1:t})\, p(m_i \mid z_{1:t-1}, x_{1:t})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Markov}}{=} \frac{p(z_t \mid m_i, x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Bayes rule}}{=} \frac{p(m_i \mid z_t, x_t)\, p(z_t \mid x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(m_i \mid x_t)\, p(z_t \mid z_{1:t-1}, x_{1:t})}
\end{aligned}
$$

Static State Binary Bayes Filter

$$
\begin{aligned}
p(m_i \mid z_{1:t}, x_{1:t}) &\overset{\text{Bayes rule}}{=} \frac{p(z_t \mid m_i, z_{1:t-1}, x_{1:t})\, p(m_i \mid z_{1:t-1}, x_{1:t})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Markov}}{=} \frac{p(z_t \mid m_i, x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Bayes rule}}{=} \frac{p(m_i \mid z_t, x_t)\, p(z_t \mid x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(m_i \mid x_t)\, p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Markov}}{=} \frac{p(m_i \mid z_t, x_t)\, p(z_t \mid x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(m_i)\, p(z_t \mid z_{1:t-1}, x_{1:t})}
\end{aligned}
$$

Static State Binary Bayes Filter

$$
\begin{aligned}
p(m_i \mid z_{1:t}, x_{1:t}) &\overset{\text{Bayes rule}}{=} \frac{p(z_t \mid m_i, z_{1:t-1}, x_{1:t})\, p(m_i \mid z_{1:t-1}, x_{1:t})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Markov}}{=} \frac{p(z_t \mid m_i, x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Bayes rule}}{=} \frac{p(m_i \mid z_t, x_t)\, p(z_t \mid x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(m_i \mid x_t)\, p(z_t \mid z_{1:t-1}, x_{1:t})} \\
&\overset{\text{Markov}}{=} \frac{p(m_i \mid z_t, x_t)\, p(z_t \mid x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(m_i)\, p(z_t \mid z_{1:t-1}, x_{1:t})}
\end{aligned}
$$

Do exactly the same for the opposite event:

$$
p(\neg m_i \mid z_{1:t}, x_{1:t}) \overset{\text{the same}}{=} \frac{p(\neg m_i \mid z_t, x_t)\, p(z_t \mid x_t)\, p(\neg m_i \mid z_{1:t-1}, x_{1:t-1})}{p(\neg m_i)\, p(z_t \mid z_{1:t-1}, x_{1:t})}
$$

Static State Binary Bayes Filter

- By computing the ratio of both probabilities, we obtain:

$$\frac{p(m_i \mid z_{1:t}, x_{1:t})}{p(\neg m_i \mid z_{1:t}, x_{1:t})} = \frac{\dfrac{p(m_i|z_t,x_t)\, p(z_t|x_t)\, p(m_i|z_{1:t-1},x_{1:t-1})}{p(m_i)\, p(z_t|z_{1:t-1},x_{1:t})}}{\dfrac{p(\neg m_i|z_t,x_t)\, p(z_t|x_t)\, p(\neg m_i|z_{1:t-1},x_{1:t-1})}{p(\neg m_i)\, p(z_t|z_{1:t-1},x_{1:t})}}$$

Static State Binary Bayes Filter

- By computing the ratio of both probabilities, we obtain:

$$\frac{p(m_i \mid z_{1:t}, x_{1:t})}{p(\neg m_i \mid z_{1:t}, x_{1:t})}$$

$$= \frac{p(m_i \mid z_t, x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})\, p(\neg m_i)}{p(\neg m_i \mid z_t, x_t)\, p(\neg m_i \mid z_{1:t-1}, x_{1:t-1})\, p(m_i)}$$

$$= \frac{p(m_i \mid z_t, x_t)}{1 - p(m_i \mid z_t, x_t)} \frac{p(m_i \mid z_{1:t-1}, x_{1:t-1})}{1 - p(m_i \mid z_{1:t-1}, x_{1:t-1})} \frac{1 - p(m_i)}{p(m_i)}$$

Static State Binary Bayes Filter

- By computing the ratio of both probabilities, we obtain:

$$\frac{p(m_i \mid z_{1:t}, x_{1:t})}{1 - p(m_i \mid z_{1:t}, x_{1:t})}$$

$$= \frac{p(m_i \mid z_t, x_t)\, p(m_i \mid z_{1:t-1}, x_{1:t-1})\, p(\neg m_i)}{p(\neg m_i \mid z_t, x_t)\, p(\neg m_i \mid z_{1:t-1}, x_{1:t-1})\, p(m_i)}$$

$$= \underbrace{\frac{p(m_i \mid z_t, x_t)}{1 - p(m_i \mid z_t, x_t)}}_{\text{uses } z_t} \underbrace{\frac{p(m_i \mid z_{1:t-1}, x_{1:t-1})}{1 - p(m_i \mid z_{1:t-1}, x_{1:t-1})}}_{\text{recursive term}} \underbrace{\frac{1 - p(m_i)}{p(m_i)}}_{\text{prior}}$$

Log Odds Notation

- Log odds ratio is defined as

$$l(x) = \log \frac{p(x)}{1 - p(x)}$$

- and with the ability to retrieve $p(x)$

$$p(x) = 1 - \frac{1}{1 + \exp l(x)}$$

Occupancy Mapping in Log Odds Form

- The product turns into a sum

$$l(m_i \mid z_{1:t}, x_{1:t})$$
$$= \underbrace{l(m_i \mid z_t, x_t)}_{\text{inverse sensor model}} + \underbrace{l(m_i \mid z_{1:t-1}, x_{1:t-1})}_{\text{recursive term}} - \underbrace{l(m_i)}_{\text{prior}}$$

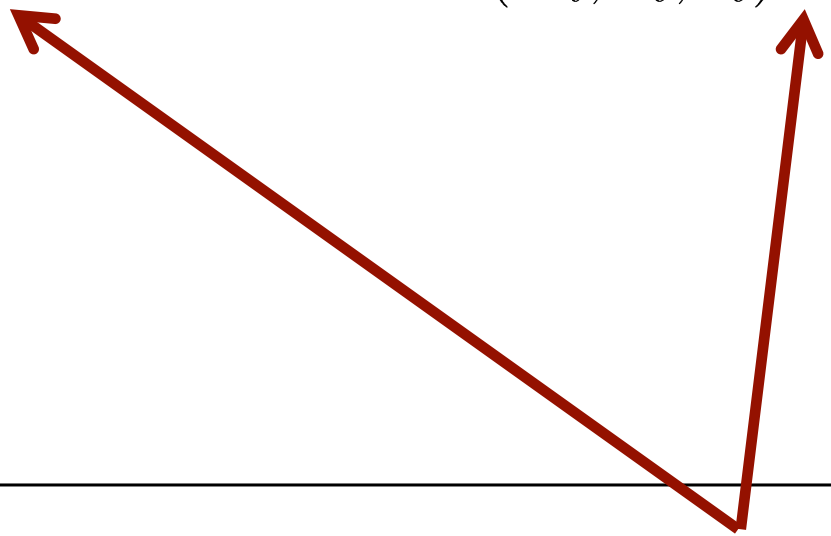
- or in short

$$l_{t,i} = \text{inv\_sensor\_model}(m_i, x_t, z_t) + l_{t-1,i} - l_0$$

Occupancy Mapping Algorithm

occupancy_grid_mapping($\{l_{t-1,i}\}, x_t, z_t$):

1: *for all cells* m_i *do*
2: *if* m_i *in perceptual field of* z_t *then*
3: $l_{t,i} = l_{t-1,i} + \text{inv\_sensor\_model}(m_i, x_t, z_t) - l_0$
4: *else*
5: $l_{t,i} = l_{t-1,i}$
6: *endif*
7: *endfor*
8: *return* $\{l_{t,i}\}$



highly efficient, only requires to compute sums

Occupancy Grid Mapping

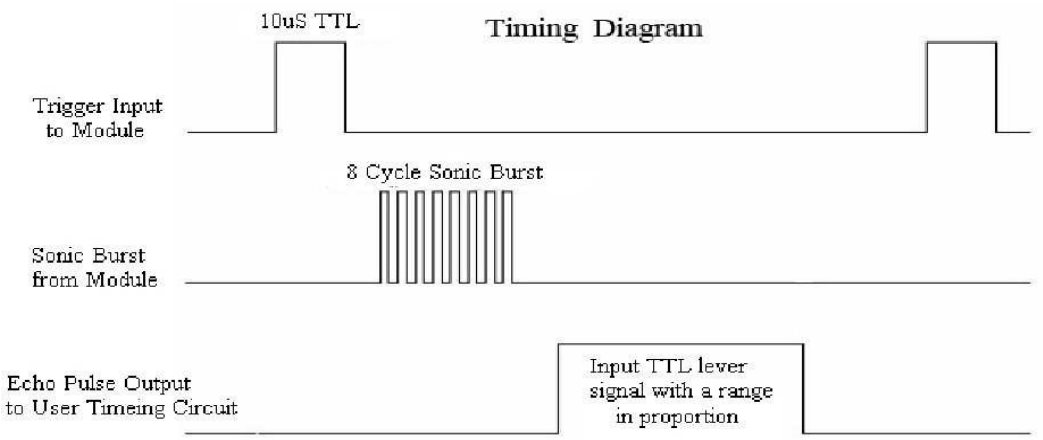
- Developed in the mid 80'ies by Moravec and Elfes
- Originally developed for noisy sonars
- Also called "mapping with know poses"

Range Sensors

- There are a variety of sensors that are used to measure the range to objects in the scene.
- These sensors are commonly used to help a robot perceive it's environment and navigate accordingly

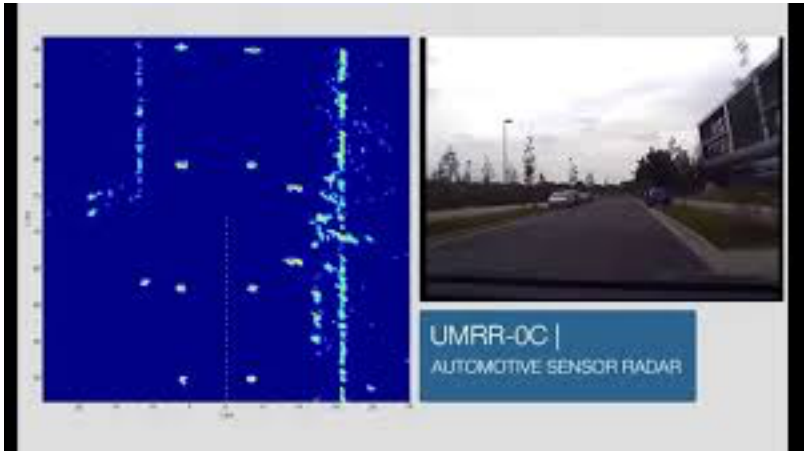
Ultrasonic Sensors

- Work by sending out an ultrasonic 'chirp' and recording the time between the emission and the returned signal.
- Often a pretty wide measurement cone 15 degrees for this unit



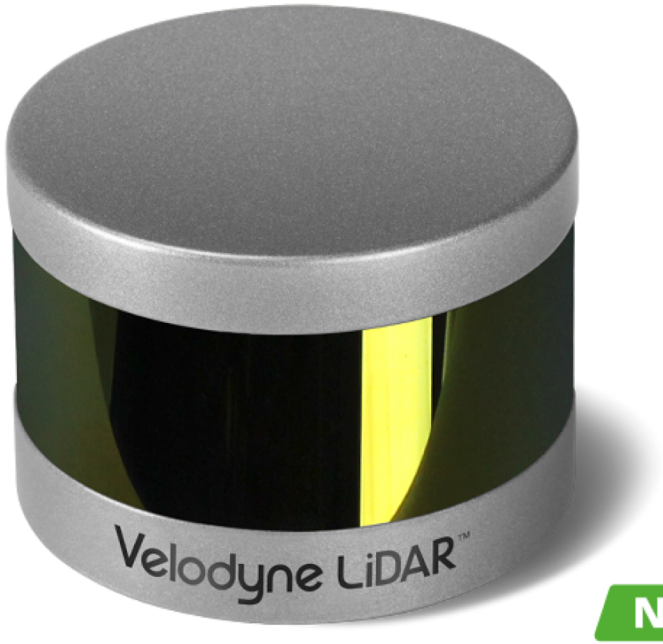
Automotive RADAR

- Much like Ultrasonic sensors it works by sending out a radio wave pulse and measuring the phase difference of the return. Typically lower resolution than LIDAR but cheaper.



LIDAR

- Light Detection and Ranging
  - Scanning sensor - egs. Velodyne
  - Laser unit generates a focused beam – return circuitry measures phase difference between emitted and returned light signal and determines range



FLASH Lidar Systems - PMD

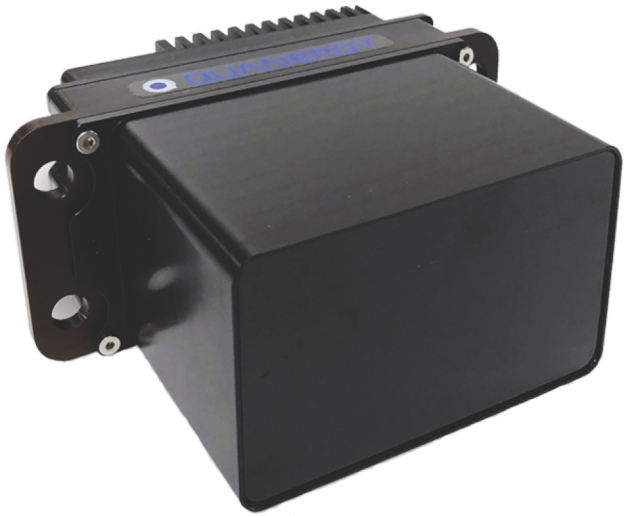
- Solid state – single image sensor with multiple pixels. Range measured to multiple points simultaneously.
- Pros
  - smaller size – can be integrated into cell phones
- Cons
  - Smaller effective range

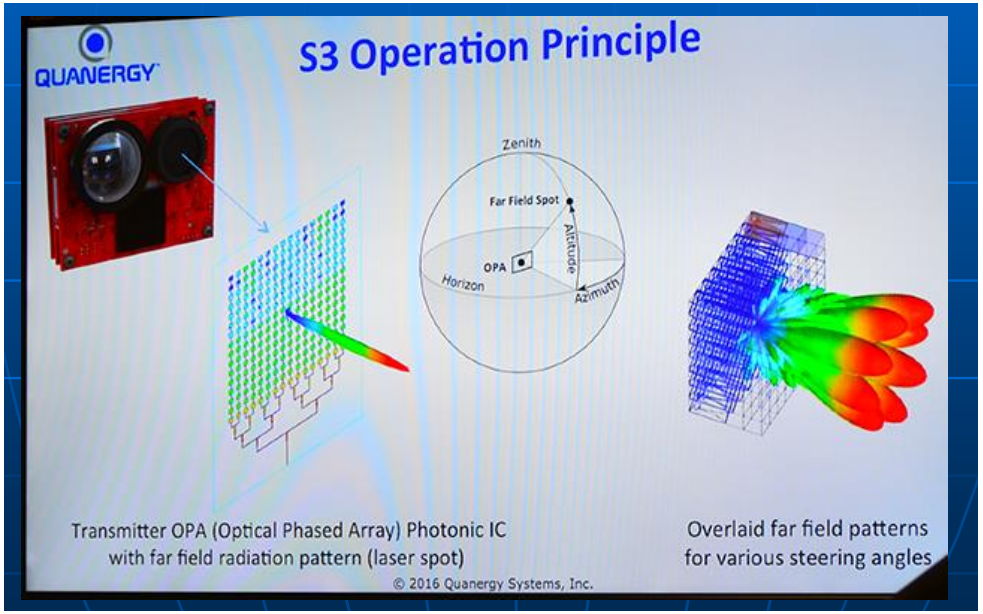
LIDAR

- Pros
  - Direct measurement of range – easy to interpret and integrate
- Cons
  - Relatively low resolution
  - Power consumption
  - Range dependent on power of light source and reflectivity of surfaces

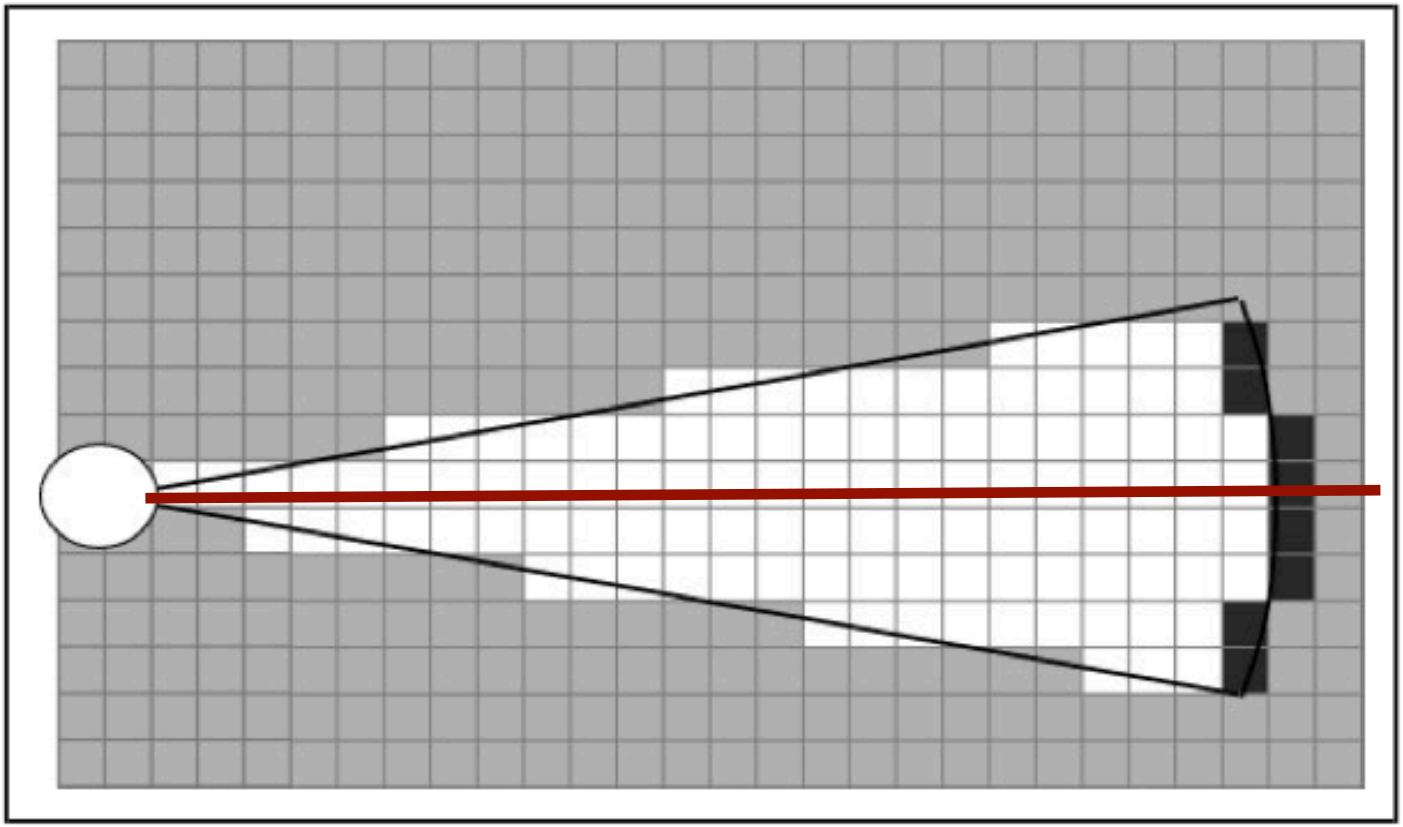
Solid State LIDAR

- Several companies are working to develop solid state LIDAR like this unit from Quanergy
- The advantage would be no moving parts hence higher reliability and lower cost.



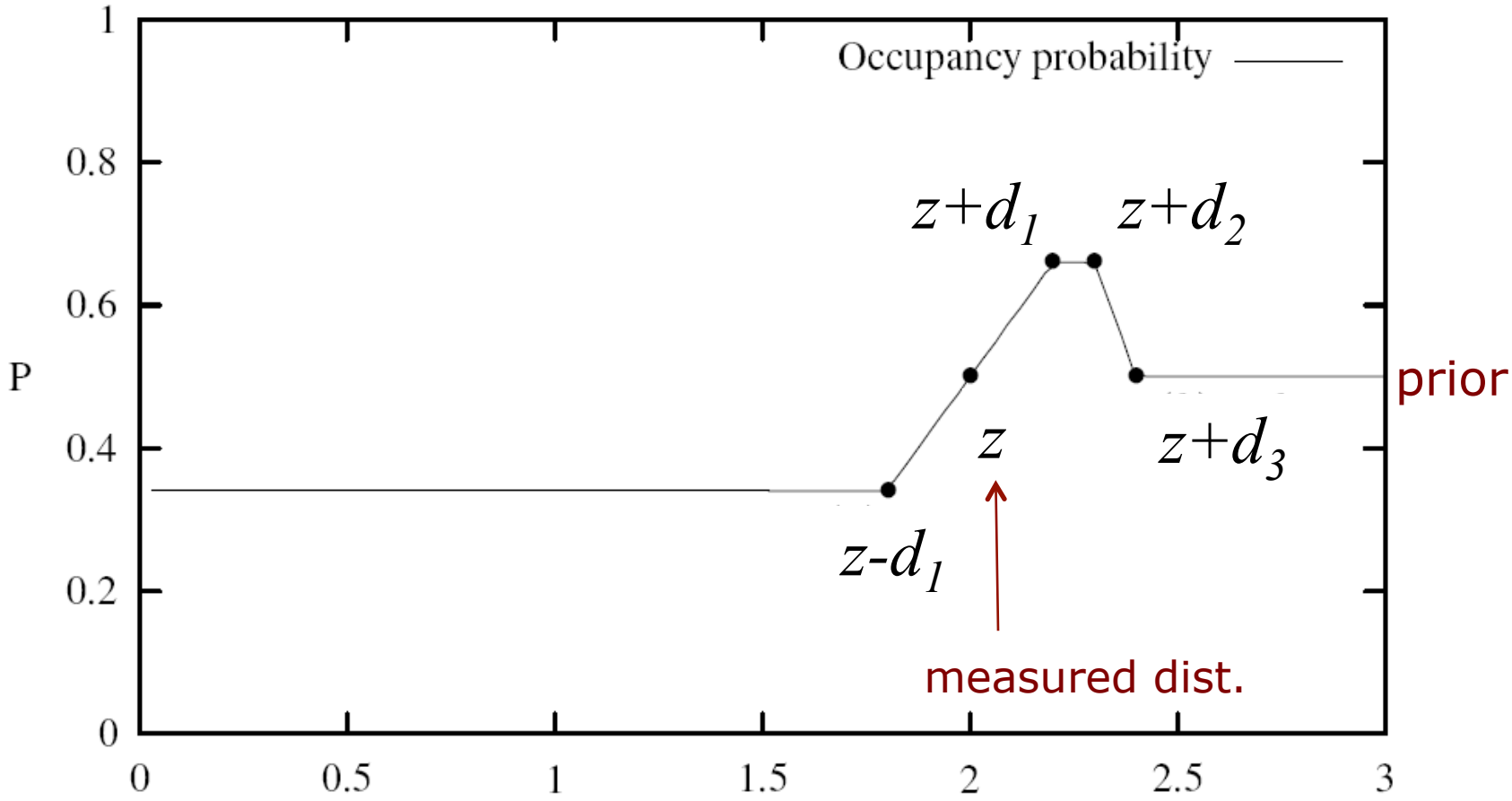


Inverse Sensor Model for Sonars Range Sensors

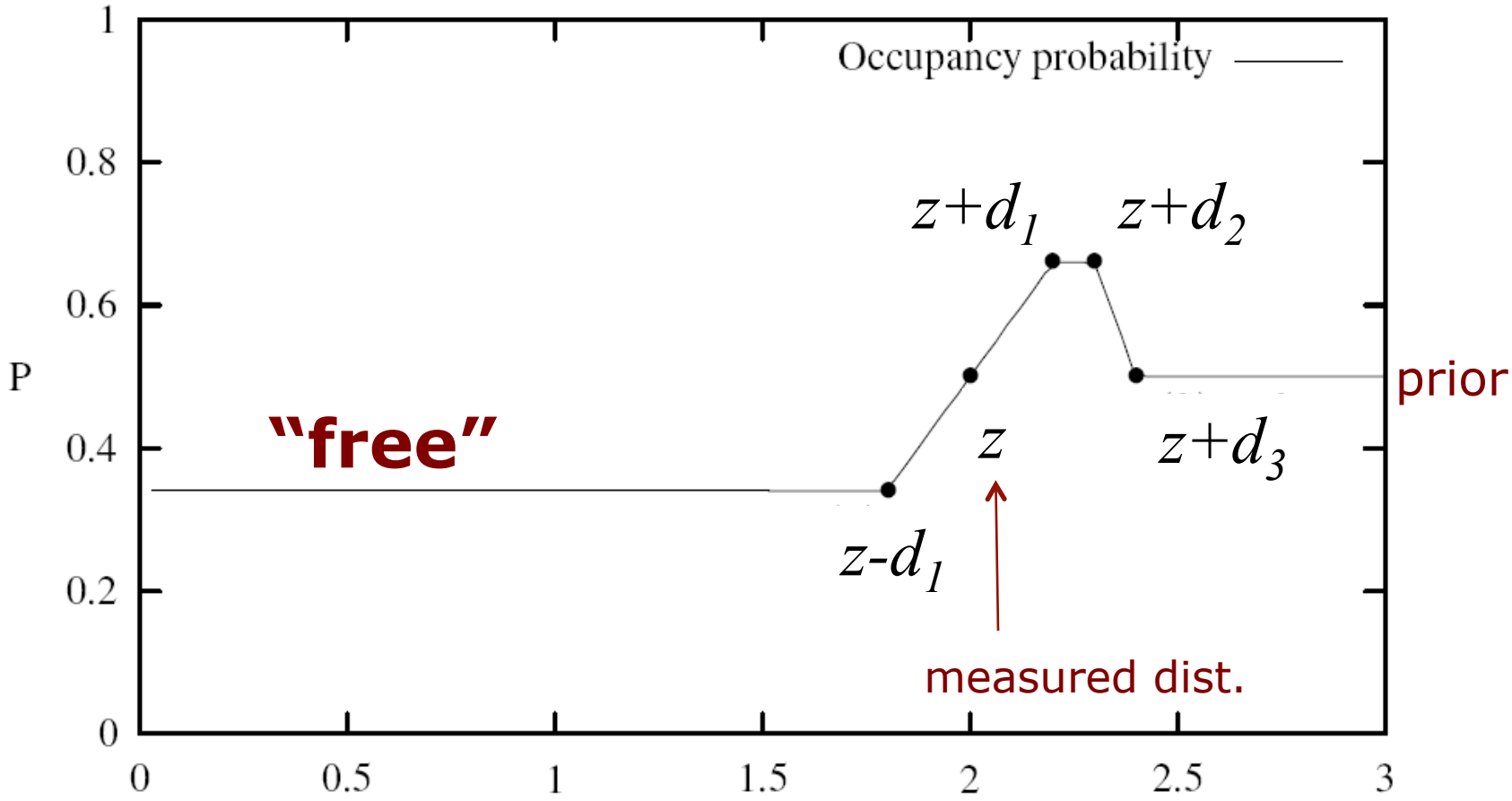


In the following, consider the cells along the optical axis (red line)

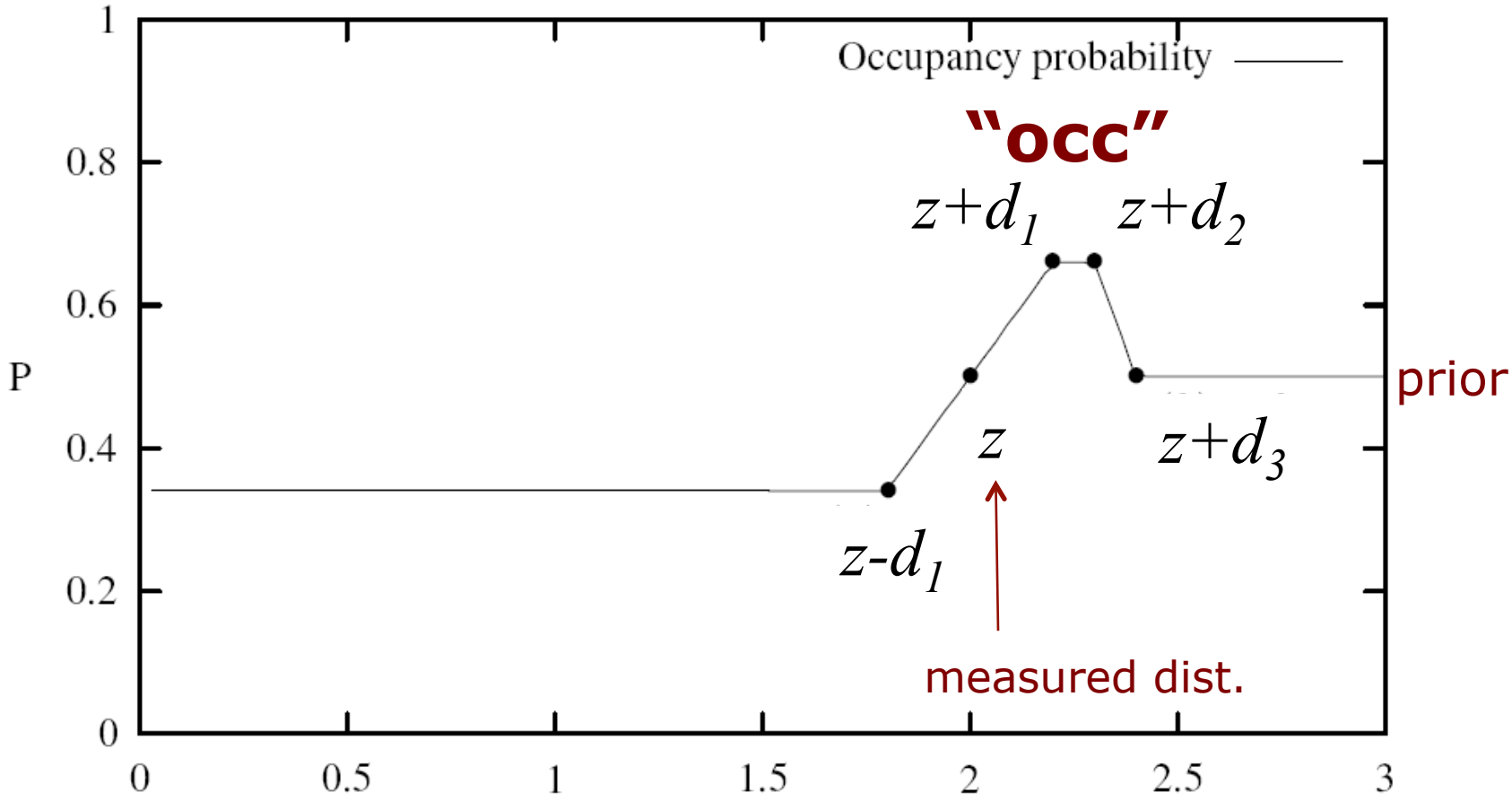
Occupancy Value Depending on the Measured Distance



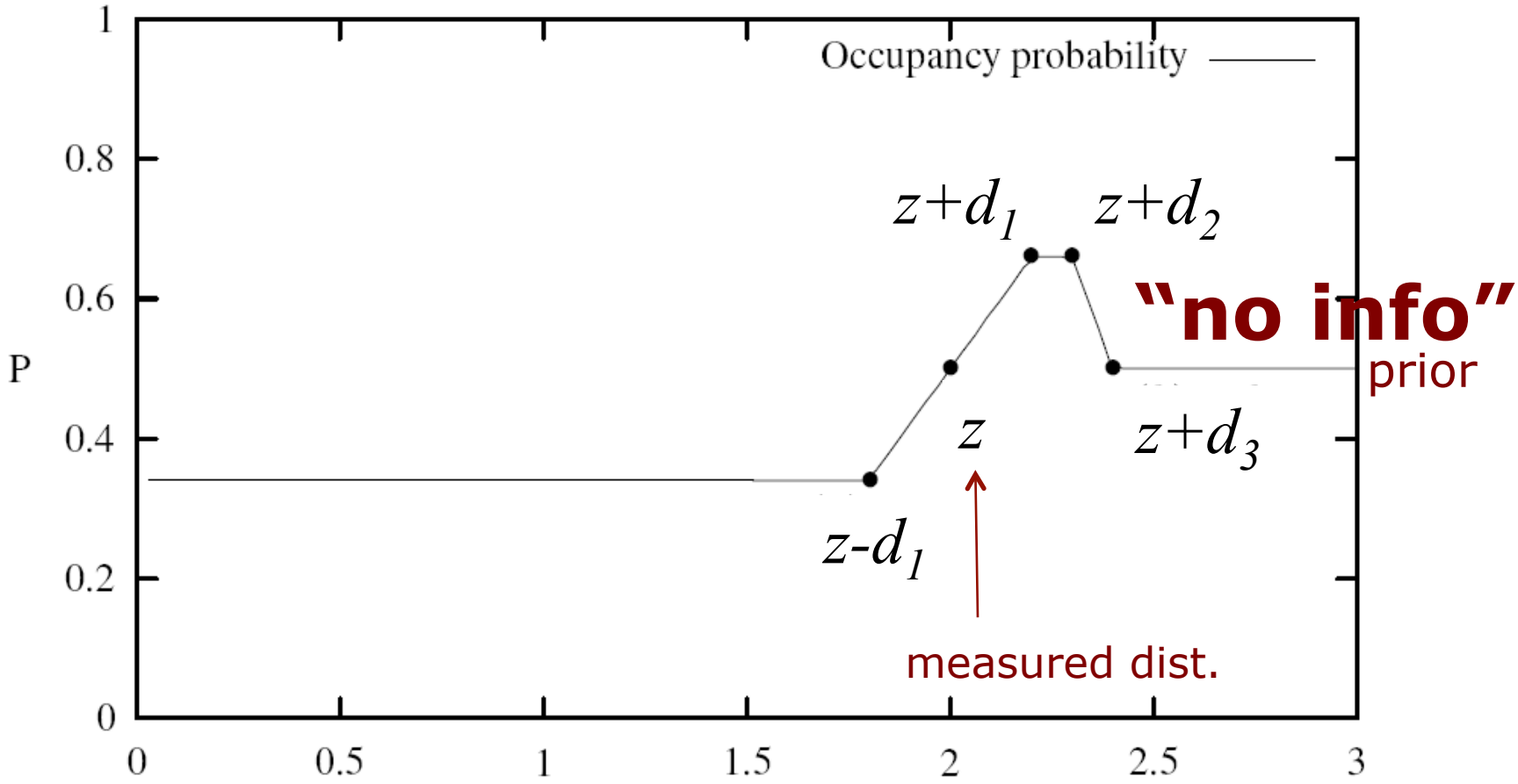
Occupancy Value Depending on the Measured Distance



Occupancy Value Depending on the Measured Distance

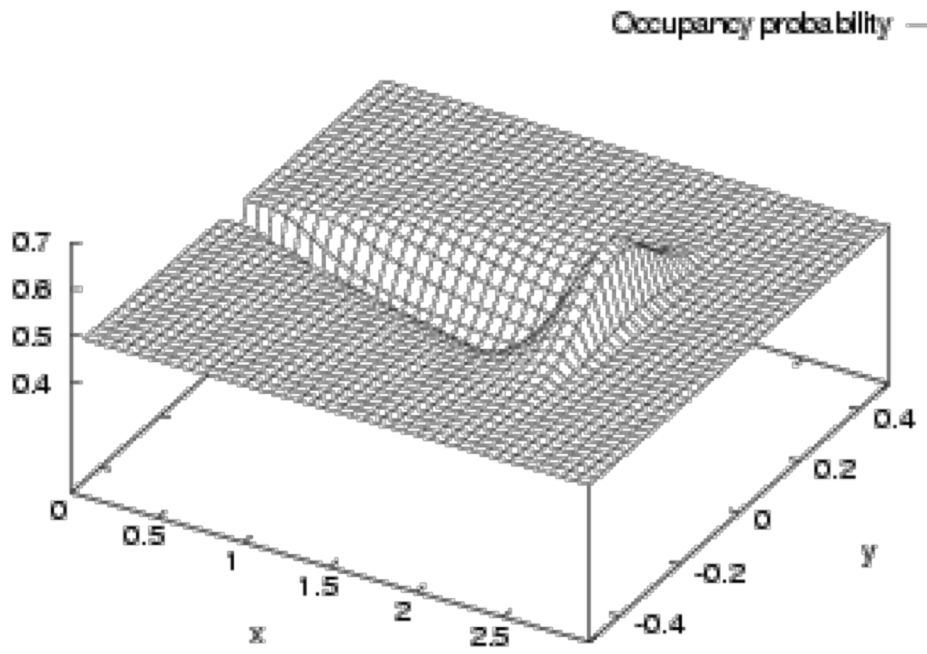


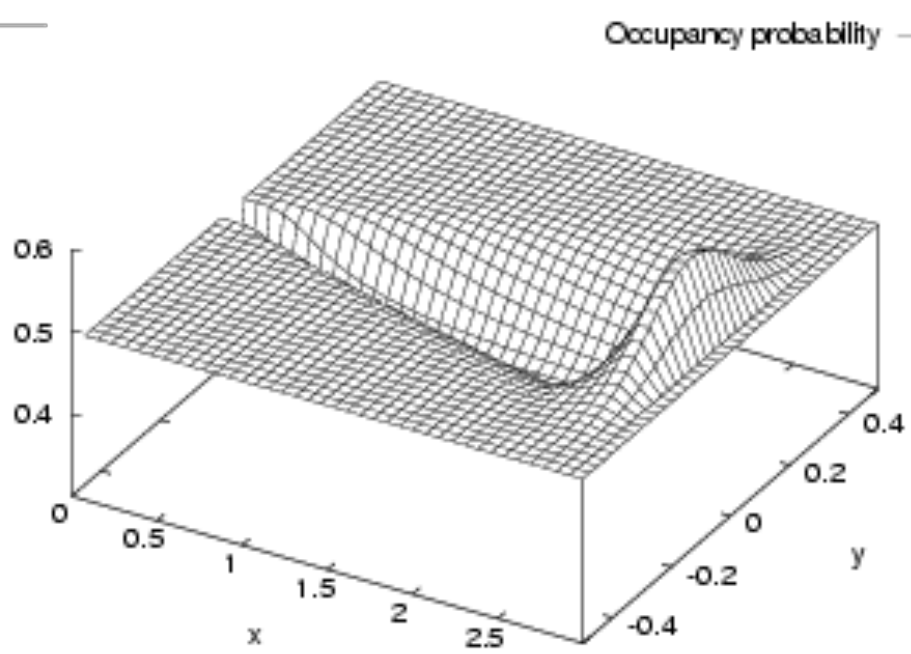
Occupancy Value Depending on the Measured Distance



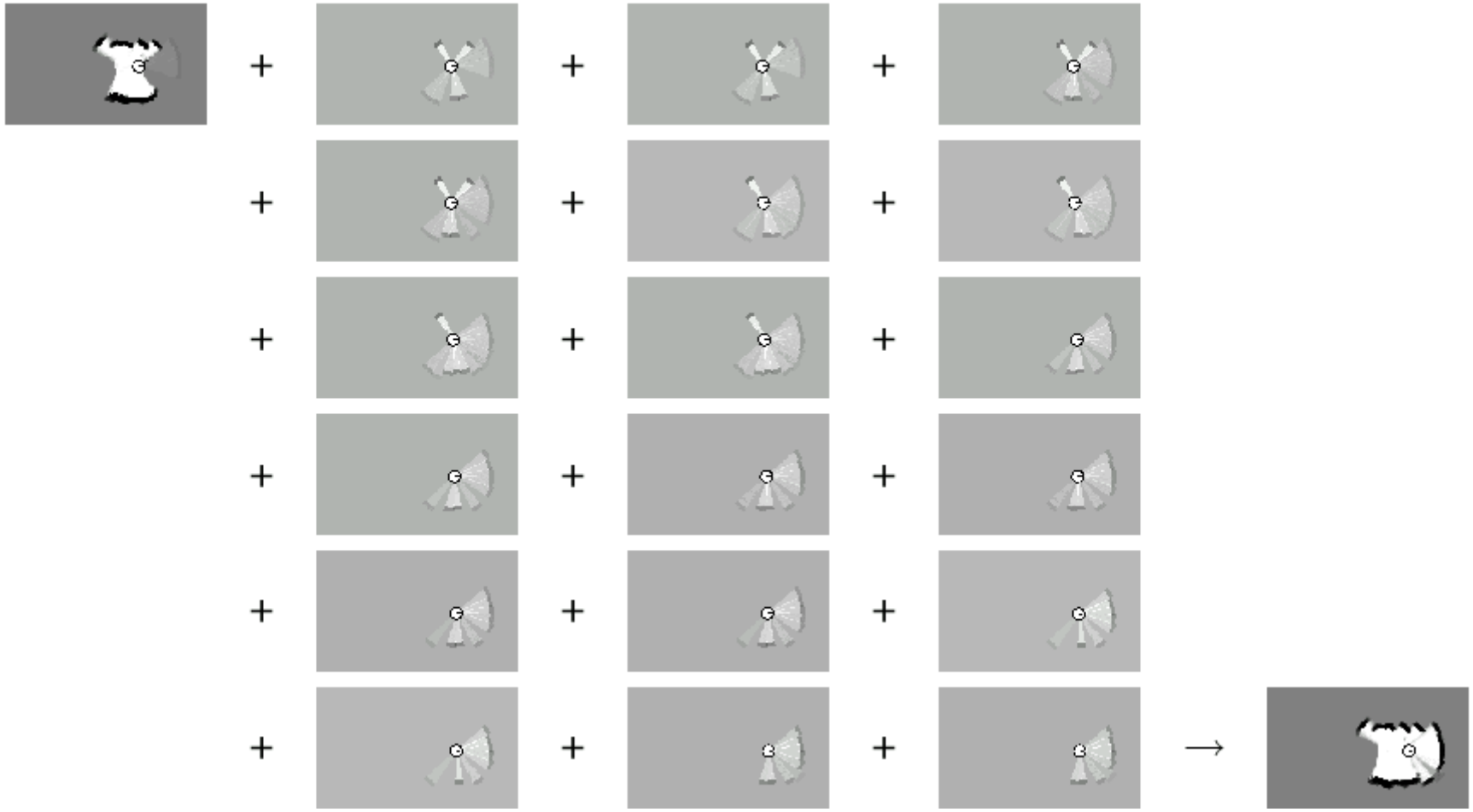
Typical Sensor Model for Occupancy Grid Maps

Combination of a linear function and a Gaussian:

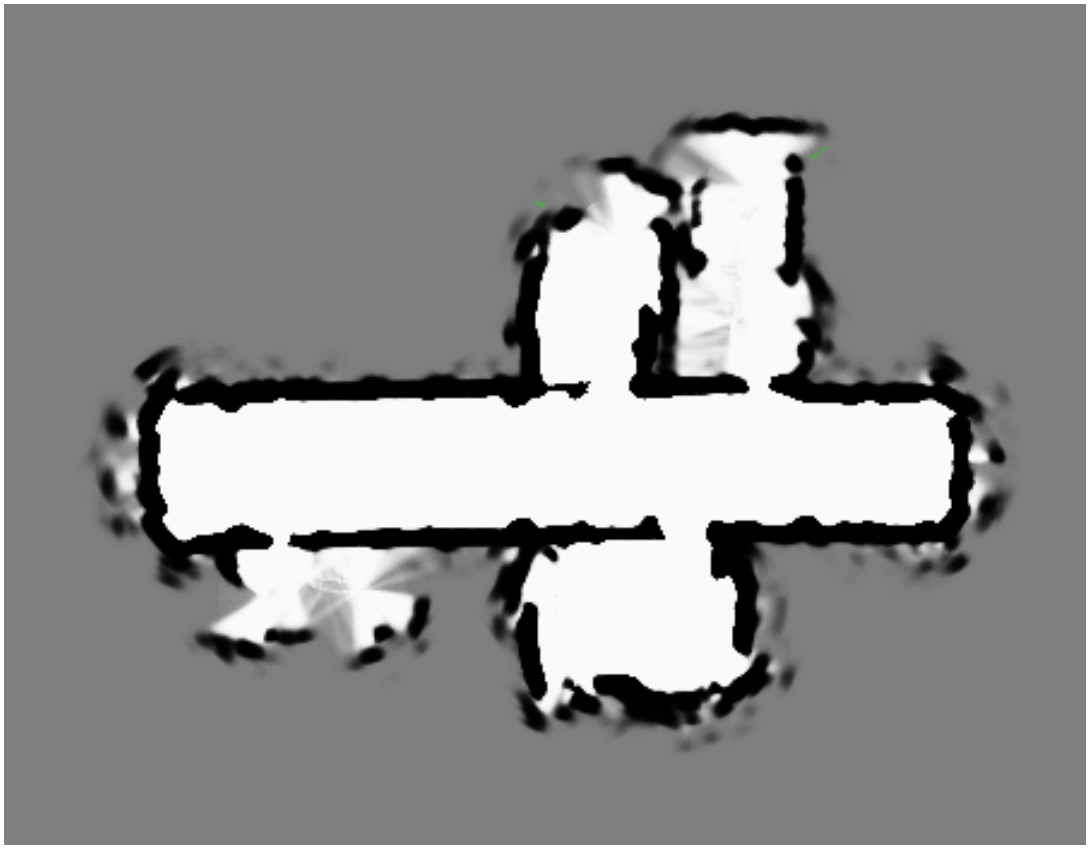




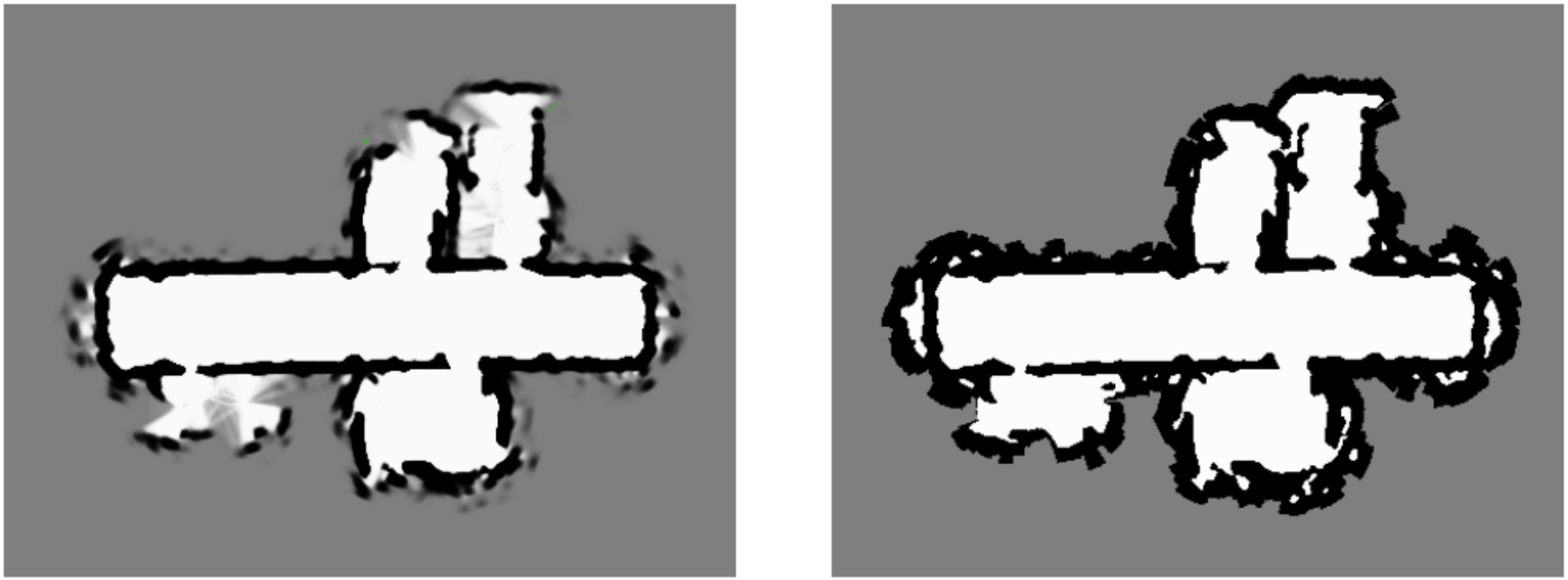
Example: Incremental Updating of Occupancy Grids



Resulting Map Obtained with Ultrasound Sensors

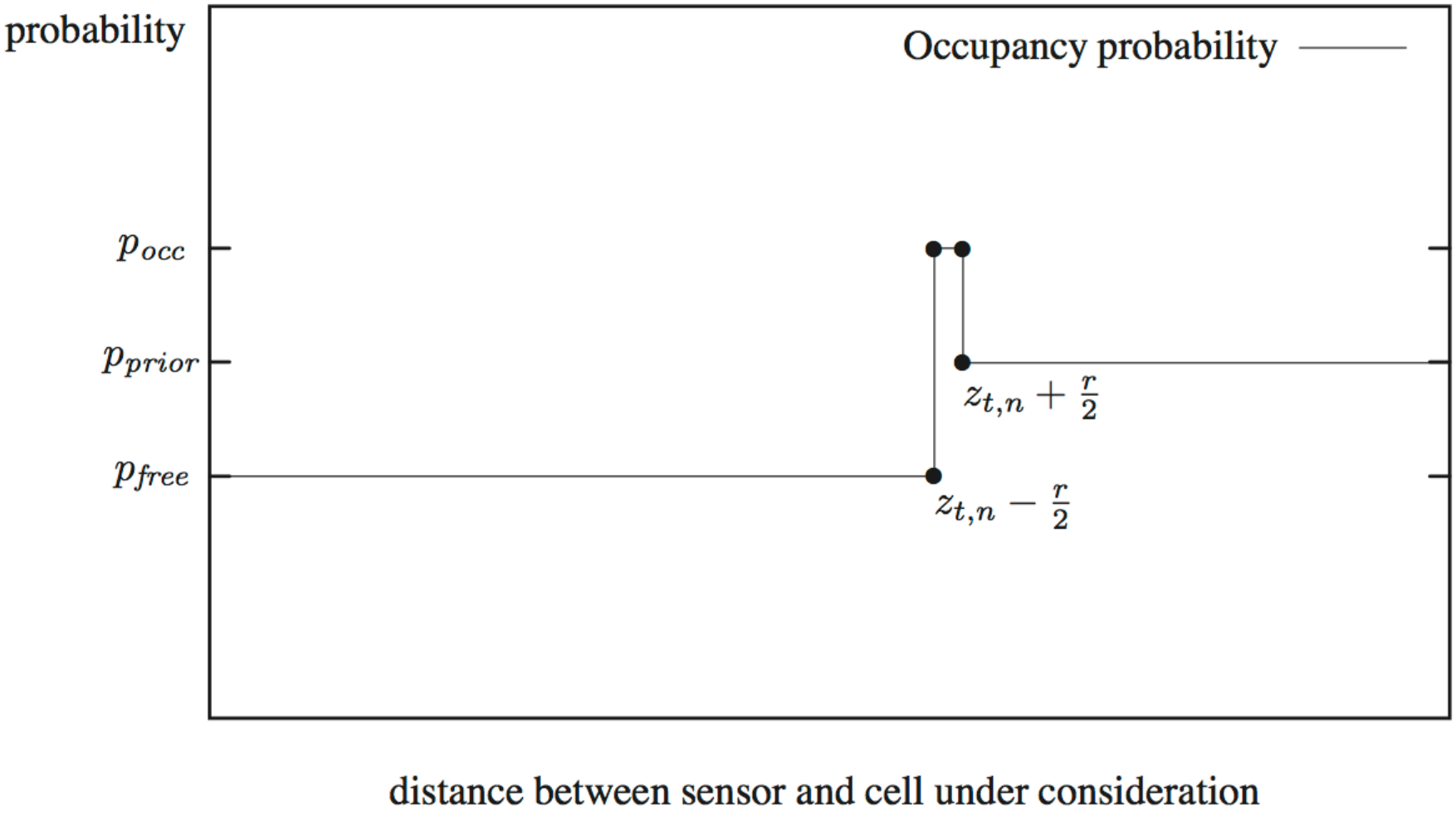


Resulting Occupancy and Maximum Likelihood Map

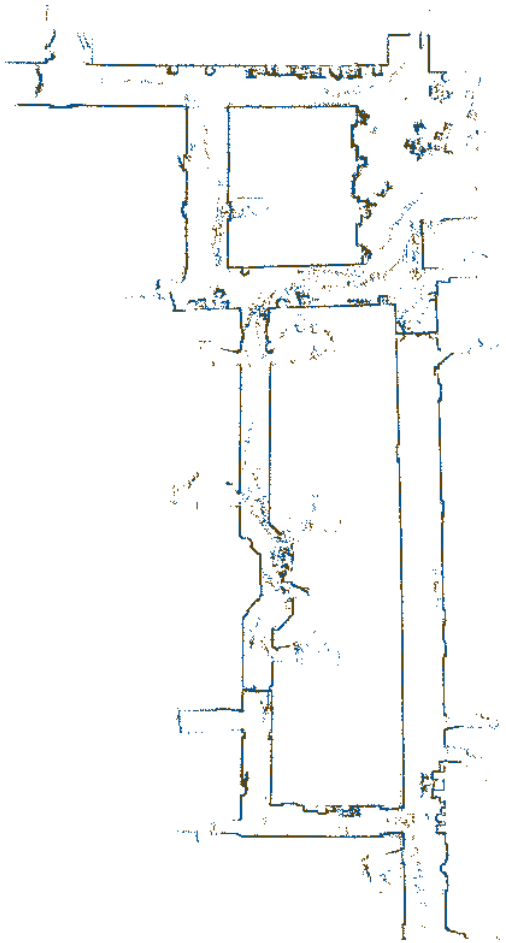


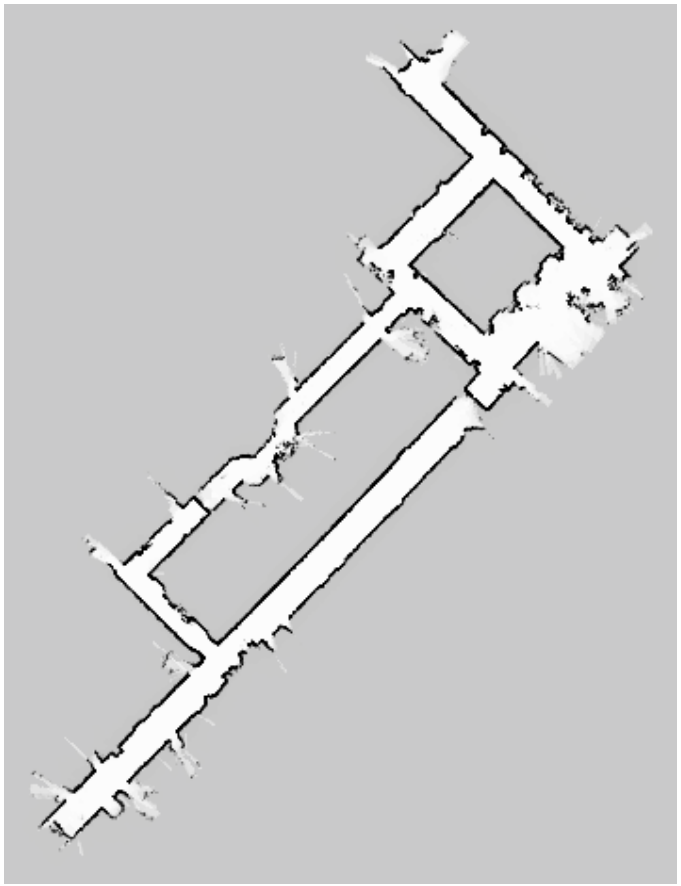
The maximum likelihood map is obtained by rounding the probability for each cell to 0 or 1.

Inverse Sensor Model for Laser Range Finders

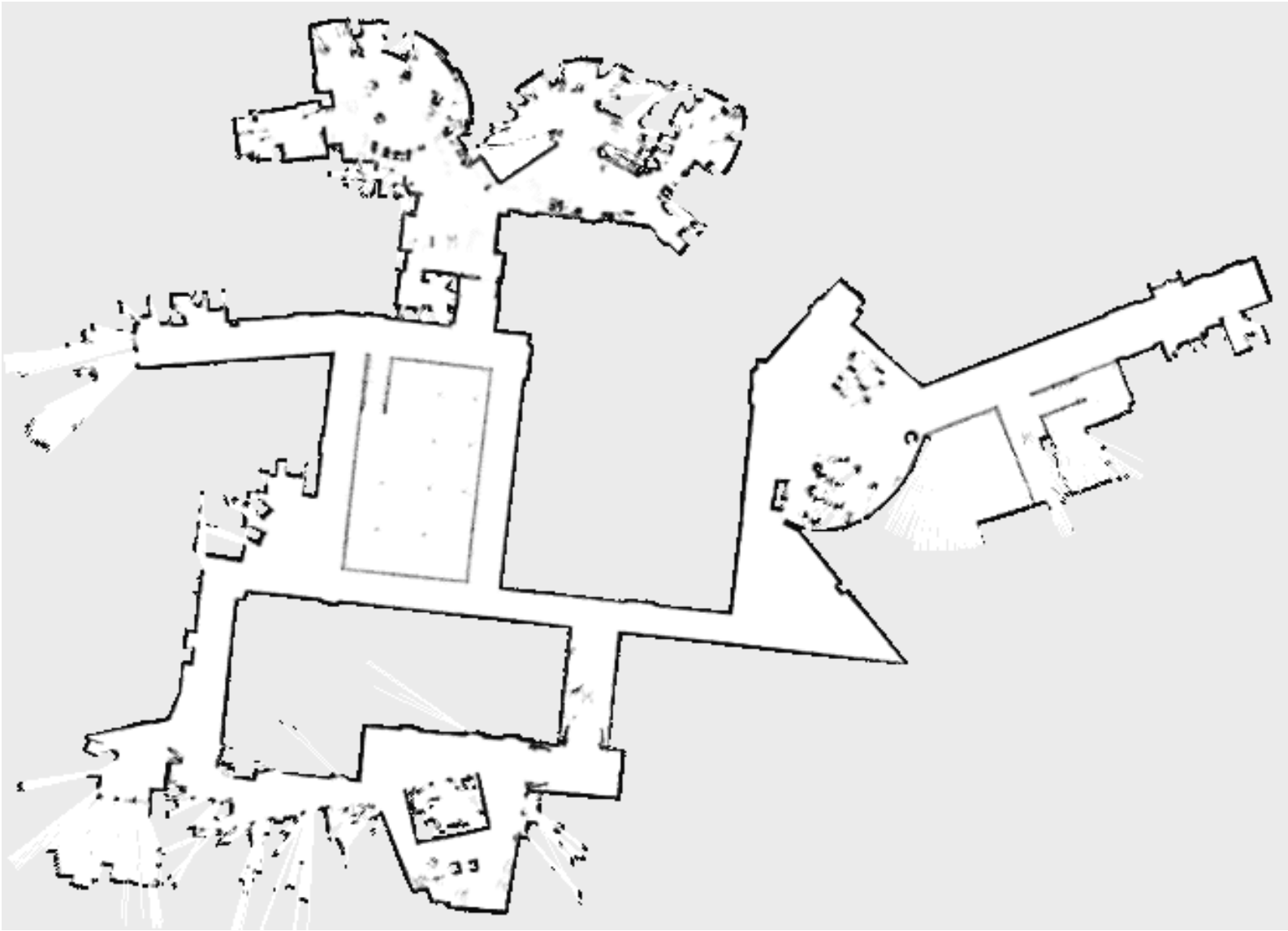


Occupancy Grids
From Laser Scans to Maps

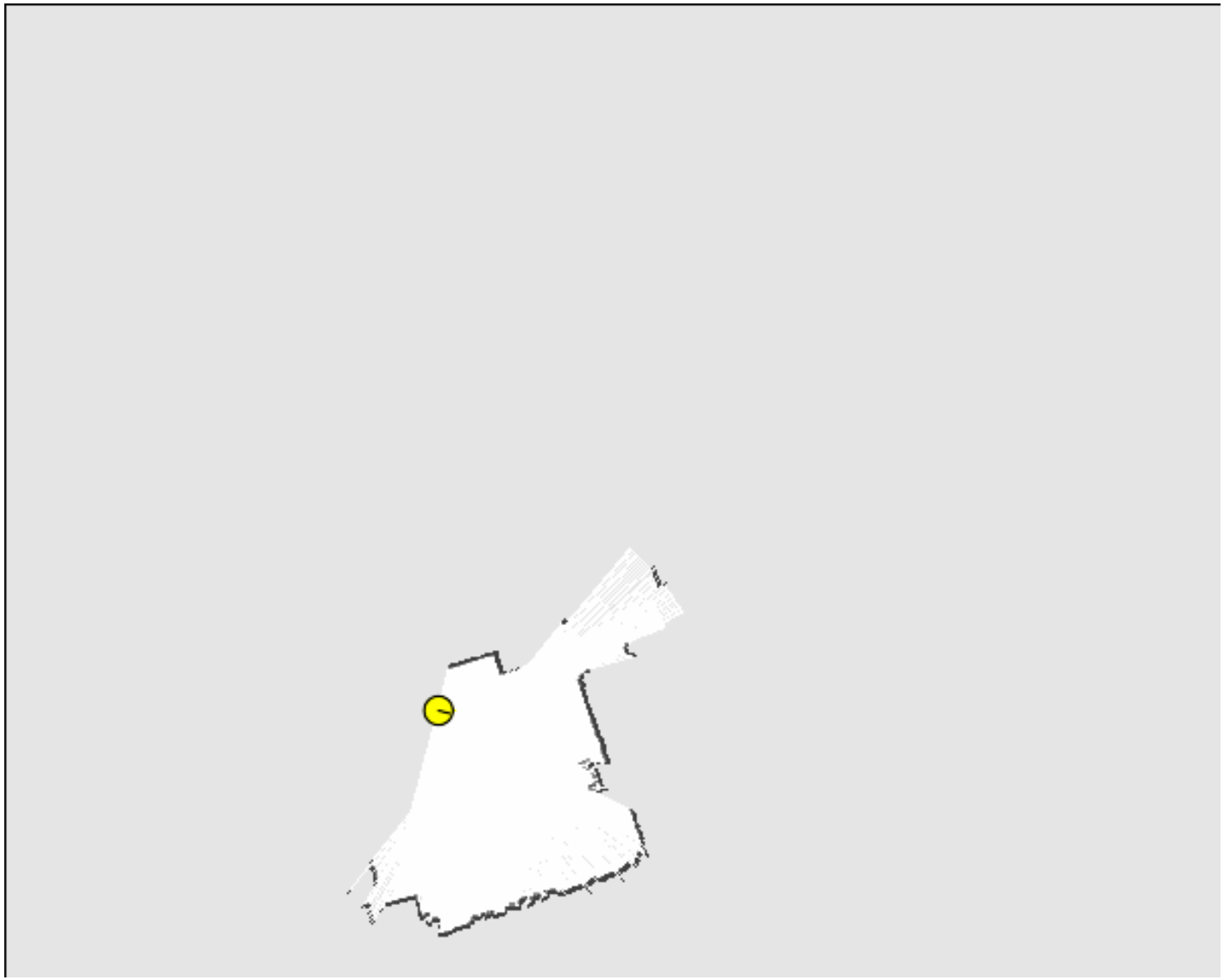




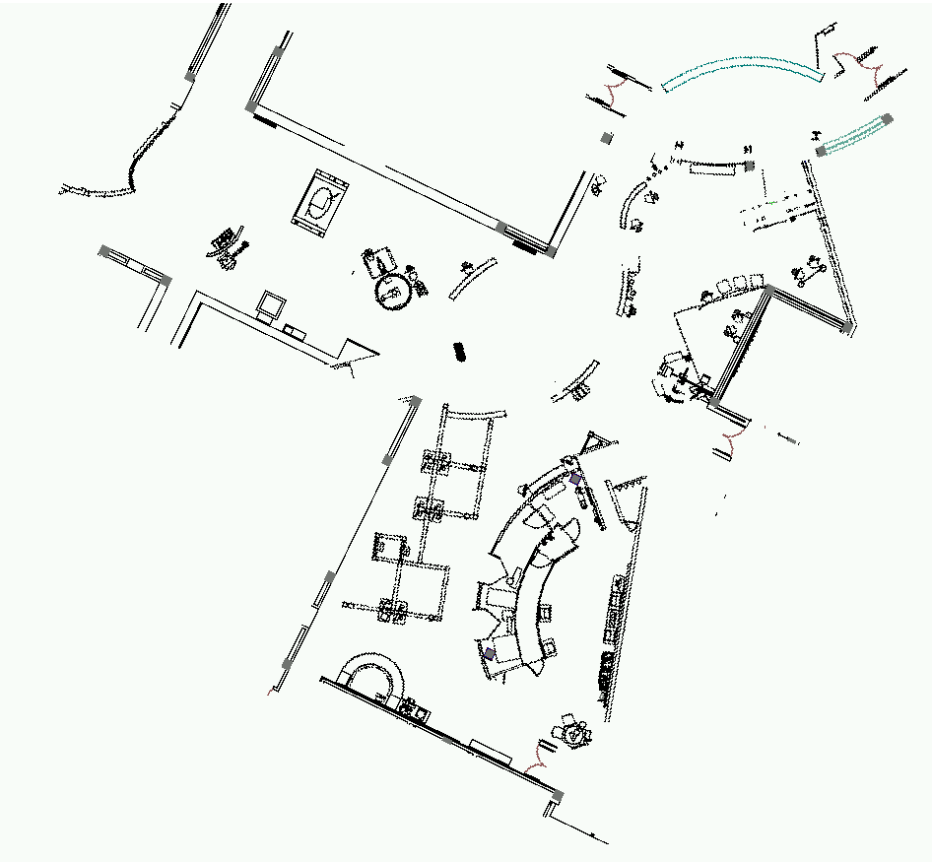
Example: MIT CSAIL 3rd Floor



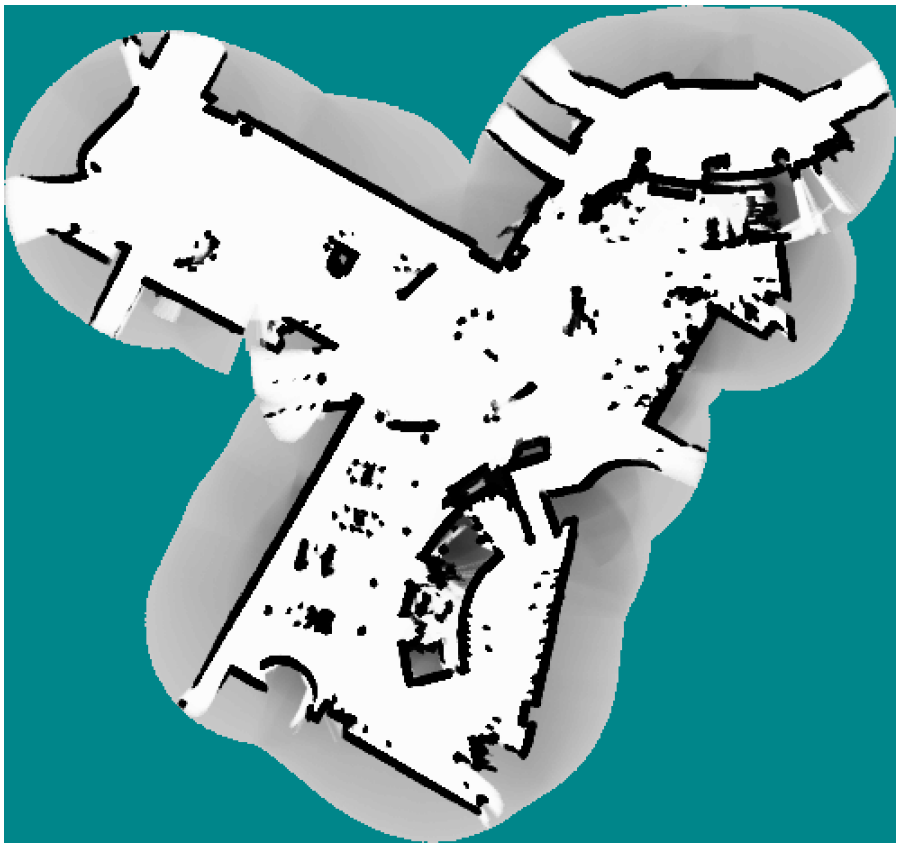
Uni Freiburg Building 106



Tech Museum, San Jose



CAD map



occupancy grid map

Summary

- Occupancy grid maps discretize the space into independent cells
- Each cell is a binary random variable estimating if the cell is occupied
- Static state binary Bayes filter per cell
- Mapping with known poses is easy
- Log odds model is fast to compute
- No need for predefined features

Alternative: Simple Counting

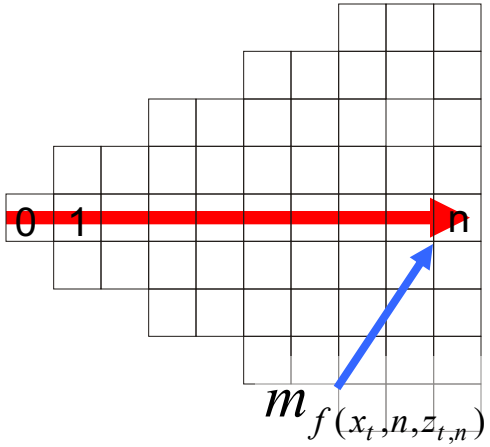
- For every cell count
  - hits(x,y): number of cases where a beam ended at <x,y>
  - misses(x,y): number of cases where a beam passed through <x,y>

$$Bel(m^{[xy]}) = \frac{\text{hits}(x,y)}{\text{hits}(x,y) + \text{misses}(x,y)}$$

- Value of interest: $P(reflects(x,y))$

The Measurement Model

1. pose at time *t*: x_t
2. beam *n* of scan *t*: $z_{t,n}$
3. maximum range reading: $\varsigma_{t,n} = 1$
4. beam reflected by an object: $\varsigma_{t,n} = 0$

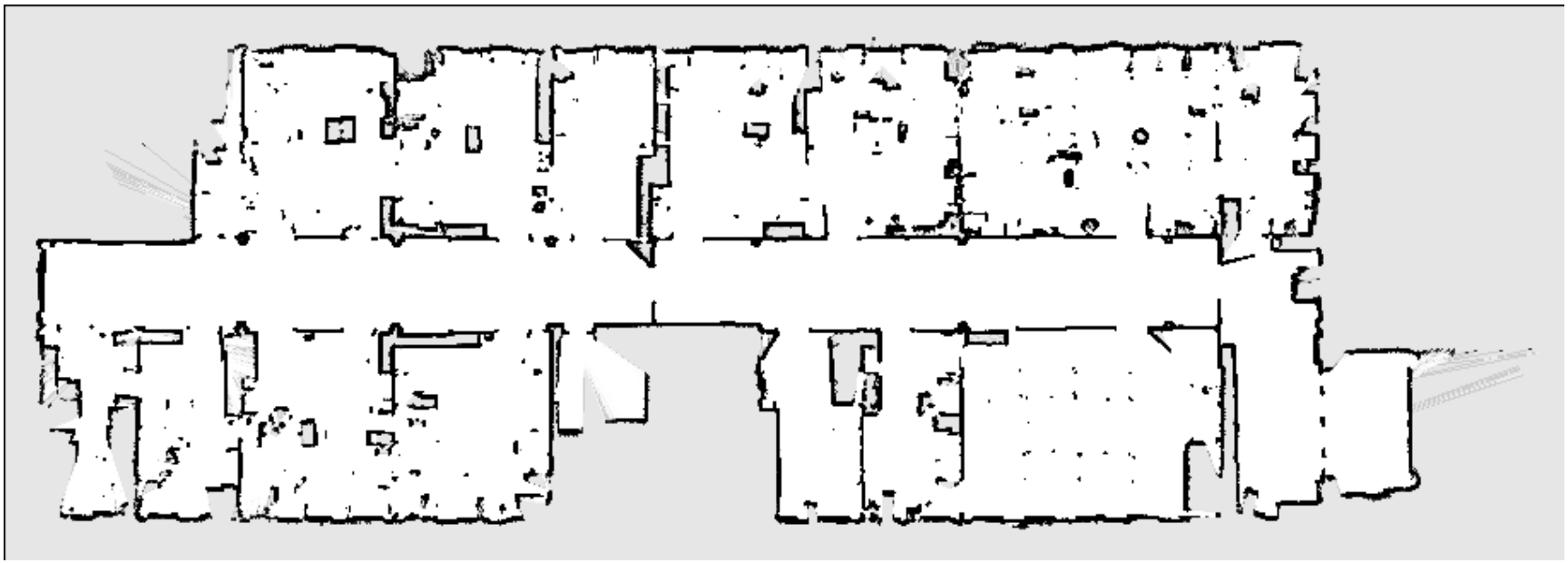


$$p(z_{t,n} \mid x_t, m) = \begin{cases} \prod_{k=0}^{z_{t,n}-1} (1 - m_{f(x_t,n,k)}) & \text{if } \varsigma_{t,n} = 1 \\ m_{f(x_t,n,z_{t,n})} \prod_{k=0}^{z_{t,n}-1} (1 - m_{f(x_t,n,k)}) & \text{if } \varsigma_{t,n} = 0 \end{cases}$$

Difference between Occupancy Grid Maps and Counting

- The counting model determines how often a cell reflects a beam.
- The occupancy model represents whether or not a cell is occupied by an object.
- Although a cell might be occupied by an object, the reflection probability of this object might be very small.

Example Occupancy Map



Example Reflection Map

Example

- Out of 1000 beams only 60% are reflected from a cell and 40% intercept it without ending in it.
- Accordingly, the reflection probability will be 0.6.
- Suppose *p(occ | z)* = 0.55 when a beam ends in a cell and *p(occ | z)* = 0.45 when a cell is intercepted by a beam that does not end in it.
- Accordingly, after n measurements we will have

$$\left(\frac{0.55}{0.45}\right)^{n*0.6} * \left(\frac{0.45}{0.55}\right)^{n*0.4} = \left(\frac{11}{9}\right)^{n*0.6} * \left(\frac{11}{9}\right)^{-n*0.4} = \left(\frac{11}{9}\right)^{n*0.2}$$

- Whereas the reflection map yields a value of 0.6, the occupancy grid value converges to 1.

Summary

- Occupancy grid maps are a popular approach to represent the environment of a mobile robot given known poses.
- In this approach each cell is considered independently from all others.
- It stores the posterior probability that the corresponding area in the environment is occupied.
- Occupancy grid maps can be learned efficiently using a probabilistic approach.
- Reflection maps are an alternative representation.
- They store in each cell the probability that a beam is reflected by this cell.
- We provided a sensor model for computing the likelihood of measurements and showed that the counting procedure underlying reflection maps yield the optimal map.

Literature

Static state binary Bayes filter

- Thrun et al.: “Probabilistic Robotics”, Chapter 4.2

Occupancy Grid Mapping

- Thrun et al.: “Probabilistic Robotics”, Chapter 9.1+9.2

Lecture 9

Markov Decision Processes (MDPs)

Reading

- (Thrun) Chapter 15
- (Sutton & Barto) Chapters 3–4
- Optional: (Bertsekas) Chapter 1 and 4

This is the beginning of Module 2, this module is about "how to act". The first module was about "how to sense". The prototypical problem in the first module was how to assimilate the information gathered by all the sensors into some representation of the world. In the next few lectures, we will assume that this representation is "good". It is accurate in terms of its geometry (small variance of the occupancy grid, small innovation in the Kalman filter etc.) and it has all the necessary semantics, e.g., objects are labeled as cars, buses, pedestrians etc (we will talk about how to do this in Module 4). The prototypical problem then is how to move around in this world, or affect the state of this world to achieve a desired outcome, e.g., drive a car from some place A to another place B.

9.1 Dynamic programming

Let us denote the state of a robot (and the world) by $x_k \in X \subset \mathbb{R}^n$ at the k^{th} timestep. We can change this state using a control input $u_k \in U \subset \mathbb{R}^p$ and this change is written as

$$x_{k+1} = f_k(x_k, u_k) \tag{9.1}$$

for $k = 0, 1, \ldots, T-1$ starting from some initial given state x_0. The function f_k is the dynamics. The time T is some time-horizon up to which we care about controlling the system. The state-space is X (which we will assume does not change with time k) and the control-space is U.

Question 1 (Expanding the state-space). Writing the dynamics as $f_k(x_k, u_k)$ is a big modeling assumption. The next state x_{k+1} could also depend on x_{k-1} for instance, which our dynamics does not allow. Does this mean that we are stuck solving a very restricted set of problems?

(9.1) tells us how the state evolves given a control input u_k at each timestep. How do we pick u_k? When you walk to school and turn right on Walnut, this control input is dependent on a lot of things, e.g., the place where you wish to end up at time T, the place you started at namely your home x_0, and certainly the spot that you are current at x_k. We will take a very general view and formalize the problem as follows. Consider a cost function

$$q_k(x_k, u_k) \in \mathbb{R}$$

which gives a scalar real-valued output for every pair (x_k, u_k). This models the fact that you do not want to walk more than you need to, i.e., we would like to minimize q_k. You also want to make sure you reach the lecture venue, let's write this down as another cost function to minimize $q_T(x_T)$. Altogether, you want to pick control inputs $(u_0, u_1, \ldots, u_{T-1})$ such that

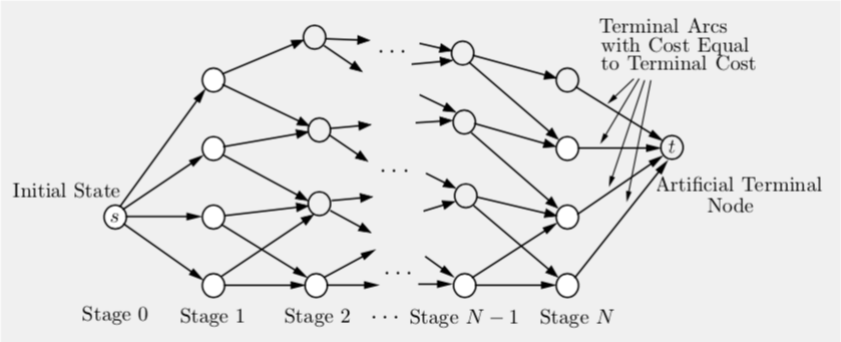
$$J(x_0; u_0, u_1, \ldots, u_{T-1}) = q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k) \tag{9.2}$$

is minimized. The cost $q_f(x_T)$ is called the terminal cost, it is high if x_T is not the lecture room and small otherwise. The cost q_k is called the run-time cost, it is high for instance if you have to use large control inputs, e.g., x_k is a climb. We want to find control *sequences* that minimize the total cost above, i.e., we want to solve

$$J^*(x_0) = \min_{u_k \in U, k=0,\ldots,T-1} J(x_0; u_0, \ldots, u_{T-1}) \tag{9.3}$$

9.1.1 Discrete, finite dynamic programming

If the state-space X and control-space U are discrete and finite sets, we can solve (9.3) as a shortest path problem using very fast algorithms. Consider the following picture. This is what would be called a transition graph for a deterministic finite-state dynamics.



The graph has one source state x_0. Each node in the graph is x_k, each edge depicts taking a certain control u_k. Depending on which control one picks, we transition to some other state x_{k+1} given by the dynamics $f_k(x_k, u_k)$. On each edge we write down the cost $q_k(x_k, u_k)$ and "close" the graph with a dummy terminal node with the cost $q(x_T)$ on every edge leading to a terminal time sink state T. Minimizing the cost in (9.3) is now the same as finding the shortest path in this graph from the source to the sink. The algorithm to do so is quite simple and is called Dijkstra's algorithm after Edsgar Dijkstra who used it around 1956 as a test program for a new computer named ARMAC (http://www-set.win.tue.nl/UnsungHeroes/machines/armac.html).

1. Q is a set of nodes that are unvisited; all nodes in the graph are added to it. S is an empty set. An array called $dist$ maintains the distance of every node in the graph from the source node x_0. Initalize $dist(t) = 0$ and $dist = \infty$ for all other nodes.

2. At each step, if Q is not empty, pop a node $v \in Q$ such that $v \notin S$ with the smallest $dist(v)$. Add v to S. Update the $dist$ of all nodes u connected to v. For each u, if

$$dist(u) > dist(v) + dist(u, v)$$

   udpate the distance of u to $dist(v) + dist(u, v)$, if this condition is not true do nothing.

The algorithm terminates when the set Q is empty.

You might know that there are many other variants of Dijkstra's algorithm, e.g., the A^* algorithm that are quicker to find shortest paths. We will look at these a few lectures from now.

Question 2. Shortest path algorithms do not work if there are cycles in the graph because the shortest path is not unique. Are there cycles in the above graph?

Question 3. What should one do if the state/control space is not finite? Can we still use Dijkstra's algorithm?

9.1.2 Dynamic programming

The principle of dynamic programming is a formalization of the idea behind Dijkstra's algorithm. It was discovered by Richard Bellman in the 1940s. The idea behind dynamic programming is quite intuitive: it says that the remainder of an optimal trajectory is optimal. Suppose that we find the optimal control sequence $(u_0^*, u_1^*, \ldots, u_{T-1}^*)$ for the problem in (9.3). There is a *unique* sequence of states $(x_0, x_1^*, \ldots, x_T^*)$ that this gives rise to. Each successive state is given by

$$x_{k+1}^* = f_k(x_k^*, u_k^*)$$

with $x_0^* = x_0$. The principle of optimality, or the principle of dynamic programming, states that if one starts from a state x_k^* at time k and wishes to minimize the "cost-to-go"

$$q_k(x_k^*, u_k) + \sum_{i=k+1}^{T-1} q_i(x_i, u_i) + q_f(x_T)$$

over the (now assumed unknown) sequence of controls $(u_k, u_{k+1}, \ldots, u_{T-1})$, then the optimal control sequence for this truncated problem is exactly $(u_k^*, \ldots, u_{T-1}^*)$. The proof of this assertion is easy: if the truncated sequence were not optimal starting from x_k^* there exists some other optimal sequence of controls for the truncated problem, say $(v_k^*, \ldots, v_{T-1}^*)$. If so, the solution of the original problem where one takes controls v_k^* from this new sequnces for timesteps $k, k+1, \ldots, T-1$ would have a lower cost. Hence the original sequence of controls would not be optimal.

The essense of dynamic programming is to solve the larger, original problem by sequentially solving the truncated sub-problems. At each iteration, Dijkstra's algorithm constructs the functions

$$J_T^*(x_T), J_{T-1}^*(x_{T-1}), \ldots, J_0^*(x_0)$$

starting from J_T* and proceeding backwards to $J_{T-1}^*, J_{T-2}^* \ldots$. The array $dist(v)$ at iteration k is really just $J_{T-k}^*(x_v)$. If we write down all this intuition mathematically, dynamic programming looks as follows.

Principle of dynamic programming.

1. Initialize $J_T^*(x) = q_f(x)$ for all $x \in X$.
2. For all times $k = 0, \ldots, T-1$, set

$$J_k^*(x) = \min_{u_k \in U} \left\{ q_k(x, u_k) + J_{k+1}^*(f_k(x, u_k)) \right\} \tag{9.4}$$

for all $x \in X$.

After running the above algorithm we have the optimal cost-to-go for each state $x \in X$ at each time $t \in \{0, \ldots, T\}$. We really only wanted $J_0^*(x_0)$ but had to do all this extra work of computing J_k^* for all the states. How much is the complexity of running this algorithm? Assume that the cost of the minimization over U is neglible, it is a bunch of comparisons between floats. The operations at each iteration at $|X|$ for setting the values $J_k^*(x)$ for all $x \in X$. So the total complexity is $\mathcal{O}(T\,|X|)$.

If Dijkstra's algorithm is run on a graph with n vertices and m edges, its computational complexity is $\mathcal{O}(n + m \log n)$. You might know that using a minor modification of Dijkstra's algorithm you can obtain the shortest path from any vertex to the goal vertex of a graph, with the same computational complexity. If we again ignore the cost of comparison of the distances in step 2 of the algorithm, the complexity is $\mathcal{O}(n)$ which is equal to $\mathcal{O}(T|X|)$ because that is the number of vertices in the transition graph. The term $|X|$ is often a hurdle: this is because the number of states $|X|$ is exponential in the dimensionality of the state-space. This was called the *curse of dimensionality* by Bellman.

Question 4 (Cost of dynamic programming is linear in the time-horizon T). Notice a very important difference between (9.4) and (9.3). The latter has a minimization over a sequence of controls $(u_0, u_1, \ldots, u_{T-1})$ while the former has a minimization over only the control at time k, u_k over T iterations. The former is much much easier to solve. Why?

A related question is: the principle of dynamic programming figures out a way to solve an optimization problem over a really large control-space (the space of all control trajectories) using a linear number of optmization problems. Can we do this for all optimization problems?

Question 5 (Is the head of the optimal trajectory optimal?). The tail of an optimal trajectory in dynamic programming is optimal. Perhaps the head is also optimal for something? Consider the following example: if there is a state x_k^* that lies on the optimal trajectory, one would expect that the trajectory from x_0 to x_k^* is optimal for the cost $\sum_{i=0}^{k-1} q_k(x_k, u_k)$. This is however not true. Imagine a situation where there are two paths to go to x_k^*, path A is longer but comes down a large slope and allows the agent to pick up a lot of speed and helps travel much further beyond x_k^* without using much control. Path B does not go down the slope, it is shorter and allows the agent to stop at x_k^* if desired. The optimal way to reach x_k^* if the horizon were simply k timesteps is therefore through path B; if the agent used path A it would have to use a large control to lose all the speed and come to a stop at x_k^*. If the horizon were T timesteps and we were interested in exploiting the speed gained on path A for future timesteps, that'd be the optimal trajectory that passes through x_k^*.

Question 6 (Non-additive cost). What if we are dealing with a non-additive cost, e.g., the runtime cost at time k given by q_k is a function of both x_k and x_{k-1}? How does the dynamic programming formulation change?

Remark 7 (Q-factors). The quantity

$$Q_k^*(x, u_k) = q_k(x, u_k) + J_{k+1}^*(f(x, u_k))$$

is called the optimal Q-factor. It is simply the expression that is minimized in the right-hand side of (9.4). This nomenclature was introduced by Watkins in his thesis. The Q-learning-style algorithms deal with Q-factors. Given the Q-factor, we can obtain the value function J_k^* as

$$J_k^*(x) = \min_{u_k \in U} Q_k^*(x, u_k). \tag{9.5}$$

The dynamic programming iteration can also be written completely in terms of the Q-factors

$$Q_k^*(x, u_k) = q_k(x, u_k) + \min_{u_{k+1} \in U} Q_{k+1}(f(x, u_k), u_{k+1}).$$

9.2 Markov Decision Processes (MDPs)

MDPs are a model for the scenario when we do not completely know the dynamics $f_k(x_k, u_k)$. This may happen for a number of reasons and it is important to appreciate them in order to understand the widespread usage of MDPs.

1. We did not do a good job of identifying the function $f : X \times U \to X$. This may happen when you are driving a car on an icy road, if you undertake the same control as you do on a clean road, you might reach a different future state x_{k+1}.
2. We did not use the correct state-space X. You could write down the state of the car as given by $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$ where x, y are the Euclidean co-ordinates of the car and θ is its orientation. This

is not a good model for studying high-speed turns, which are affect by other quantities like wheel slip, the quality of the suspension etc.

We may not even know the full state sometimes. This occurs when you are modeling how users interact with an online website like Amazon.com, you'd like to model the change in state of the user from "perusing stuff" to "looking stuff to buy it" to "buying it" but there are certainly many other variables that affect the user's behavior. As another example, consider the path that an airplane takes to go from Philadelphia to Los Angeles. This path is affected by the weather at all places along the route, it'd be cumbersome to incorporate the weather to find the shortest-time path for the airplane.

3. We did not measure the state correctly. This is the situation that we talked about in Module 1, namely your sensors may not be able to measure the state x accurately. There are multiple ways of talking about this, the Kalman filter estimates $\widehat{x}_t = \mathbb{E}[x_t \mid y_0, \ldots, y_t]$ as the optimal estimate of the true state and use the contorl input $u(\hat{x})$ in place of $u(x)$. You can also imagine that the true system lives within some envelope of your dynamical system.

   (picture here)

4. We did not use the correct control-space U for the controller. This is akin to the second point above. The gas pedal which one may think of as the control input to a car is only one out of the large number of variables that affect the running of the car's engine.

MDPs are a drastic abstraction of all the above situations. We write

$$x_{k+1} = f_k(x_k, u_k) + w_k \tag{9.6}$$

where the "noise" w_k is not under our control. The quantity w_k is not arbitrary however, we assume that

1. noise w_k is a random variable and we know its distribution. For example, you ran your car lots of times on icy road and measured how the state x_{k+1} deviates from similar runs on a clean road. The difference between the two is modeled as w_k. Note that the distribution of w_k may be a function of time k.

2. noise at different timesteps $\epsilon_1, \epsilon_2, \ldots, \epsilon_{T-1}$ is independent.

Instead of a deterministic transition from x_k to x_{k+1}, we now have

$$x_{k+1} \sim \mathbb{P}(x_{k+1} \mid x_k, u_k).$$

which is just another way of writing (9.6). The latter is a probability table of size $|X| \times |U| \times |X|$ akin to the transition matrix of a Markov chain except that there is a different transition matrix for every control $u \in U$. The former version (9.6) is more amenable to analysis. MDPs can be alternatively called stochastic dynamical systems, we will use either names for them in this course.

The moral of this section is to remember that as pervasive as noise seems in all problem formulations, it models different situations depending upon the specific problem. Understanding where noise comes from is important for real-world applications.

Question 8 (Where do we find MDPs in real-life?). There are lots of expensive robots in GRASP, e.g., a Kuka manipulator such as this https://www.youtube.com/watch?v=ym64NFCWORY costs upwards of $100,000. Is it a stochastic dynamical system?

Question 9 (Handling noise correlated in time). How will you handle the situation where noise at different timesteps is correlated? Say w_k is correlated with its immediate next noise input w_{k+1}.

9.3 Value iteration

Let us now think of solving the problem (9.3) for a stochastic dynamical system. Given the same control sequence $\overline{u} = (u_0, \ldots, u_{T-1})$ we may get many different state trajectories depending upon which noise sequence $\overline{\epsilon} = (\epsilon_0, \ldots, \epsilon_{T-1})$ as the realization on a particular run. One modification that we could make in (9.3) in this case is to look at the expected value of all realizations of $(w_0, \ldots, w_{T-1})$

$$\mathop{\mathbb{E}}_{(w_0,\ldots,w_{T-1})}\left[q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k)\right].$$

Remark 10 (A control policy). It is important to note what the controls u_k are at each time instant k. In stochastic dynamic programming we are interested in optimizing over the space of policies $\mu_k(x_k)$. This section therefore finds not a sequence of controls $(u_0, \ldots, u_{T-1})$ but a sequence of policies

$$\pi = \{\mu_0, \ldots, \mu_{T-1}\};$$

each $\mu_k : X \to U$ maps states to controls. Policies are more general objects than controls. The reason to use them in the stochastic setting is to allow the control $u_k = \mu_k(x_k)$ to depend on the realized state x_k instead of simply the time k. This is known as feedback control, without this the controller cannot adapt to unexpected states x_k that may occur due to the noise. This is an important distinction from the deterministic case.

Now, the optimization problem corresponding to (9.3) is therefore written as

$$\begin{aligned} J_\pi(x_0) &= \mathop{\mathbb{E}}_{(w_0,\ldots,w_{T-1})}\left[q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, \mu_k(x_k))\right] \\ J_{\pi^*}(x_0) &= \min_{\pi\in\Pi} J_\pi(x_0). \end{aligned} \tag{9.7}$$

The optimal policy is π^* that achieves this minimum. Remember that the optimal policy is a function of the state you start in x_0.

Dijkstra's algorithm no longer works if the edges in the graph are stochastic but we can use our formalization of the dynamic programming principle to write the solution for this modified problem. The idea remains the same, to compute a sequence of cost-to-go functions $J_T^*(x_T), J_{T-1}^*(x_{T-1}), \ldots, J_0^*(x_0)$ proceeding *backwards*.

Principle of dynamic programming for stochastic systems.

1. Initialize $J_T^*(x) = q_f(x)$ for all $x \in X$.
2. For all times $k = 0, \ldots, T-1$, set

$$J_k^*(x) = \min_{u_k \in U} \left\{ q_k(x, u_k) + \mathop{\mathbb{E}}_{w_k} \left[J_{k+1}^*(f_k(x, u_k) + w_k) \right] \right\} \tag{9.8}$$

for all $x \in X$.

This algorithm works in the same as that of the deterministic case. At each iteration, we solve a sub-problem. The key difference is that there is an expectation over noise w_k in step 2 and the truncated cost-to-go J_{k+1}^* is evaluated at the state $x' = f_k(x, u_k, w_k)$. Once we have solved the minimization on the right hand side of (9.8), we set the optimal policy to be

$$\mu_k^*(x) = u_k^*(x).$$

and the optimal policy is

$$\pi^* = \left\{ \mu_0^*, \ldots, \mu_{T-1}^* \right\}.$$

The update equation in (9.8) changes to

$$J_k^*(x) = \min_{u_k \in U} \left\{ q_k(x, u_k) + \mathop{\mathbb{E}}_{x' \sim \mathbb{P}(\cdot \,|x_k, u_k)} \left[J_{k+1}^*(x') \right] \right\}$$

if know the stochastic dynamical system as a table of transition probabilities $\mathbb{P}(x' |\, x_k, u_k)$. This helps understand the computational complexity of the updates: at each iteration we have to do $|X| \times |X| \times |U|$ work and there are T such iterations. The total complexity of dynamic programming for stochastic systems is $\mathcal{O}(T|X|^2|U|)$. The quadratic term in $|X|$ is an even bigger hurdle now because the number of states $|X|$ is exponential in the dimensionality of the state-space.

Question 11 (Risk-sensitive control). The objective in (9.7) is the most popular way to incorporate the noise $(w_0, \ldots, w_{T-1})$ into our formulation but it is not the only one. Can you think of any other objective we may wish to use?

9.3.1 Infinite horizon value iteration

Infinite horizon problems consider the case when $T \to \infty$. The problem is also stationary in such a setting, i.e.,

$$q(x, u) \equiv q_k(x, u),$$
$$f(x, u) \equiv f_k(x, u)$$

for all $x \in X$ and $u \in U$. For stochastic dynamical systems, we will also require that the statistics of the noise do not change with time. Note that the assumption of an infinite number of stages is never really satisfied in practice; it is a reasonable assumption of long, finite horizon problems. The assumption of stationarity *is* often satisfied in practice. System parameters, say the dynamics, may vary only slowly over time, e.g., degradation of the motors of your robot. Infinite horizon problems are popular because they give rise to elegant, insightful analysis.

Let us think of optimizing an objective of the form

$$J_\pi(x_0) = \lim_{T\to\infty} \mathop{\mathbb{E}}_{w_0,w_1,\ldots} \Big[\sum_{k=0}^{T-1} \gamma^k q(x_k, \mu_k(x_k)) \Big]. \tag{9.9}$$

The parameter $\gamma \in [0, 1)$ is called the "discount" factor. It puts more weight on costs incured in the early stages and puts less weight on costs incured later on. The policy is again $\pi = \{\mu_0, \mu_1, \ldots, \}$. Remember that the infinite horizon cost is given by the limit of finite horizon costs as the horizon goes to infinity; this will be the key to solving the problem.

There are two kinds of infinite horizon problems:

1. those with a terminating state and $\gamma = 1$ where we can keep taking some zero-cost control u to stay at the state, e.g., your robot has finished moving an object from one place to the other and it simply stops at the goal state with zero control. In this case the problem is really a finite-horizon problem masquerading as an infinite horizon problem, with the difference that we do not know the length of the horizon, it can effectively be random.

2. those with $\gamma < 1$ where there need not be a terminal state. It can be shown that all such discounted problems of this form can be converted to the previous form by creating a fake terminal state where the system transitions to with probability $1 - \gamma$ at each timestep. As a result we only need to think about the terminal sink state version of the infinite horizon problem.

As before, consider the situation where we computed the optimal cost of sovling the problem in k steps, this will be denoted by $J^k(x)$ for all $x \in X$. A dynamic programming principle is again valid for quantities $J^0, J^1, J^2, \ldots$

$$J^{k+1}(x) = \min_{u\in U} \mathop{\mathbb{E}}_{w} \Big[q(x, u) + J^k(f(x, u) + w) \Big] \tag{9.10}$$

Since the terminal cost q_f is zero in the infinite horizon setting, we have $J^0(x) = 0$ for all $x \in X$. This iteration (9.10) is called value iteration. Using the intuition that the infinite horizon cost is the limit of the finite-horizon costs, we can expect that

$$J^*(x) = \lim_{N\to\infty} J^N(x)$$

for all states $x \in X$. If someone claimed that J^* was the optimal infinite horizon optimal cost, it should also satisfy the equation (9.10), i.e.,

$$J^*(x) = \min_{u\in U} \mathop{\mathbb{E}}_{w} \Big[q(x, u) + J^*(f(x, u) + w) \Big] \tag{9.11}$$

for all states $x \in X$. Note that this is really a system of equations, one for each state x. The solution of this system of equations $J^*(x)$ is the optimal cost-to-go of the infinite horizon problem from a state x. This equation is called the Bellman equation and we will come to it again soon. The policy is again easily obtained as the control u that results in the minimization in the Bellman equation

$$\mu^*(x) = \underset{u \in U}{\operatorname{argmin}} \underset{w}{\mathbb{E}} \Big[q(x,u) + J^*(f(x,u) + w) \Big].$$

The optimal policy

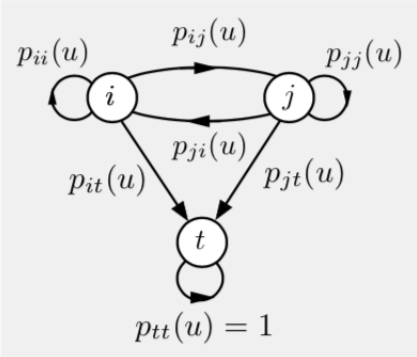
$$\pi^* = \{\mu^*, \mu^*, \ldots\}$$

is stationary now. This is quite intuitive: for an infinite horizon problem the future looks the same given the starting state x regardless of when we started.

Remark 12 (Order of minimization and expectation in the Bellman equation). The expression in (9.11) has a minimization over controls u outside the expectation over noise $\mathbb{E}_w$. This is the root of all trouble when one implements approximate versions of value iteration. If we do not compute the expectation over the noise accurately, we cannot also find the best control to take accurately.

9.3.2 Transition matrix version of the Bellman equation

Let us say that the states are $i \in \{1, \ldots, |X| = n\}$. Infinite horizon problems have a dummy terminal state where the system may remain indefinitely, let us call it t. We therefore have a transition graph of the following form.



Given the starting state i and the policy $\pi = \{\mu_0, \mu_1, \ldots, \}$, we are interested in finding the minimum of

$$J_\pi(i) = \lim_{T \to \infty} \underset{w_0, w_1, \ldots}{\mathbb{E}} \Big[\sum_{k=0}^{T-1} \gamma^k \, q(i_k, \mu_k(i_k)) \mid i_0 = i, \ \pi \Big]$$

$$J^*(i) = \min_\pi \ J_\pi(i)$$

for all states i. We will consider the version $x_{k+1} \sim \mathbb{P}(\cdot \mid x_k, u_k)$ version this time. The Bellman

equation (9.11) changes to

$$J^*(i) = \min_{u \in U} \Big[p_{it}(u) q(i,u) + \sum_{j=1}^{n} p_{ij}(u) \left(q(i,u) + J^*(j) \right) \Big]. \tag{9.12}$$

The term $p_{it}(u)q(i,u)$ is the expected cost if we terminate at the next instant by transitioning to state t. The term $\sum_{j=1}^{n} p_{ij}(u)q(i,u)$ is the expected cost of going to state j in the next instant while the third term $\sum_{j=1}^{n} p_{ij}(u)J^*(j)$ is the optimal cost-to-go from that state j.

The value iteration algorithm populates $J_0(1), J_0(2), \ldots, J_0(n)$ with arbitrary values and generates a sequence

$$J_{k+1}(i) = \min_{u \in U} \Big[p_{it}(u) q(i,u) + \sum_{j=1}^{n} p_{ij}(u) \left(q(i,u) + J_k(j) \right) \Big]. \tag{9.13}$$

9.3.3 Some theoretical results

We list down some very powerful theoretical results for value iteration. These results make it work for a large number of real-world problems and are at all the heart of all modern algorithms. The only assumption that these results need is that the set of policies Π over which we search is such that the system always reaches the terminal state after some finite number of timesteps.

1. **Convergence of value iteration.** Given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence generated by value iteration

   $$J_{k+1}(i) = \min_{u \in U} \Big[p_{it}(u)\ q(i,u) + \sum_{j=1}^{n} p_{ij}(u) \Big(q(i,u) + J_k(j) \Big) \Big]$$

   converges to the optimal cost $J^*(i)$ for each $i = 1, \ldots, n$.

2. **Unique solution of the Bellman equation.** The optimal cost function $J^* = (J^*(1), \ldots, J^*(n))$ satisfies the Bellman equation

   $$J^*(i) = \min_{u \in U} \Big[p_{it}(u)\ q(i,u) + \sum_{j=1}^{n} p_{ij}(u) \Big(q(i,u) + J^*(j) \Big) \Big]$$

   and is the unique solution of this equation. In other words, if we find some $J' = (J'(1), \ldots, J'(n))$ that satisfies the Bellman equation, we are guaranteed that this is indeed the optimal cost function.

3. **Bellman equation for a particular policy.** Consider a stationary policy $\pi = (\mu, \mu, \ldots,)$. The cost of executing this policy starting from different states $i = 1, \ldots, n$ given by $(J_\mu(1), J_\mu(2), \ldots, J_\mu(n))$ satisfies the equation

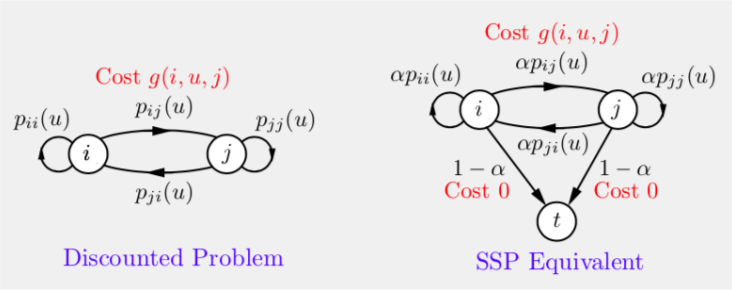
   $$J_\mu(i) = p_{it}(\mu(i))\ q(i, \mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i)) \ \Big(q(i, \mu(i)) + J_\mu(j) \Big)$$

and is the unique solution of this equation. Further, given any initial condition $(J_0(1), \dots, J_0(n))$ the sequence generated by

$$J_{k+1}(i) = p_{it}(\mu(i))\, q(i, \mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i)) \Big(q(i, \mu(i)) + J_k(j) \Big) \tag{9.14}$$

converges to the cost $J_\mu(i)$ for all $i = 1, \dots, n$. Running these set of updates to obtain the cost-to-go for a particular policy is called *evaluating a policy*.

Remark 13 (Discounted cost problems). All the above results remain conceptually similar in the case with discounted rewards.



The Bellman equation changes to

$$J^*(i) = \min_{u \in U} \sum_{j=1}^{n} p_{ij}(u) \Big(q(i,u) + \gamma J^*(j) \Big)$$

with the discount factor γ that multiplies J^* now. Value iteration

$$J_{k+1}(i) = \min_{u \in U} \sum_{j=1}^{n} p_{ij}(u) \Big(q(i,u) + \gamma J_k(j) \Big)$$

and still converges to $J^*(i)$ upon iteration.

9.4 Policy iteration

Question 14. Value iteration converges exponentially quickly, but asymptotically. The number of states $|X| = n$ is finite and so is the number of controls $|U|$. This seems very funny, one would expected that we should be able to find the optimal cost $J^*(i)$ in finite time if the problem is finite, after all we need to find n numbers $J^*(1), \dots, J^*(n)$. Where is the catch?

This section will discuss policy iteration which terminates in a finite number of iterations. Even for large problems where the policy and the cost function are approximately represented, policy iteration

works better than value iteration. The key idea of policy iteration is quite simple: given a stationary policy for an infinite horizon problem

$$\pi = (\mu, \mu, \dots,)$$

we can evaluate the policy to obtain the cost function $J_\mu(i)$ for all states $i = 1, \dots, n$. Given this cost function observe that if we find the feedback control

$$\tilde{\mu}(i) \in \operatorname*{argmin}_{u \in U} \sum_{j=1}^{n} p_{ij}(u)\Big(q(i,u) + \gamma J_\mu(j)\Big) \tag{9.15}$$

then the cost of the patched policy

$$\tilde{\pi}^1 = (\tilde{\mu}, \mu, \mu, \dots,)$$

which executes $\tilde{\mu}$ for one timestep and then executes μ. denoted by $J_{\tilde{\pi}^1}$ is better than J_μ

$$J_{\tilde{\pi}^1} \leq J_\mu.$$

The same thing is also true if we patch the policy for two timesteps

$$\tilde{\pi}^2 = (\tilde{\mu}, \tilde{\mu}, \mu, \dots,)$$

in which case we have

$$J_{\tilde{\pi}^2} \leq J_{\tilde{\pi}^1} \leq J_\mu.$$

We however known that the limit of patching for lots of timesteps is simply $J_{\tilde{mu}}$, i.e., the cost of taking the stationary policy $\tilde{\pi} = (\tilde{\mu}, \tilde{\mu}, \dots)$. We therefore have

$$J_{\tilde{\mu}}(i) \leq J_\mu(i), \quad \text{for all } i = 1, \dots, n.$$

Notice that by performing the policy improvement step (9.15), we have improved the cost-to-go for all states. We can start from any policy μ and iterate upon it this way.

Policy iteration.
For $k = 0, 1, \dots,$

1. Pick a policy μ^k and evaluate it using (9.14) to get $J_{\mu^k}(i)$ for all $i = 1, \dots, n$.
2. Perform a policy iteration step

$$\mu^{k+1}(i) \in \operatorname*{argmin}_{u \in U} \sum_{j=1}^{n} p_{ij}(u)\Big(q(i,u) + \gamma J_{\mu^k}(j)\Big)$$

to compute a new policy μ^{k+1}.

Policy iteration generates an improving sequence of policies

$$J_{\mu^{k+1}}(i) \leq J_{\mu^k}(i), \quad \text{for all } i \text{ and } k$$

and terminates with the optimal policy.

Remark 15 (Computational complexity of policy iteration). Policy iteration works really well in practice. In the worst case, the number of iterations is $|U|^{|X|}$, i.e., it is exponential in the number of states (Mansour and Singh, 2013). However, in practice it often converges in order $|X|$ iterations.

Bibliography

Mansour, Y. and Singh, S. (2013). On the complexity of policy iteration. *arXiv preprint arXiv:1301.6718*.

Lecture 11

Linear Quadratic Regulator (LQR)

Reading

- Todorov and Jordan (2002)
- Optional: Applied Optimal Control by Bryson & Ho, Chapter 4-5
- Optional: http://underactuated.csail.mit.edu/lqr.html, Lecture 3-4 at https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-323-principles-of-optimal-control-spring-2008/lecture-notes

---

The previous chapter gave two algorithms, namely value iteration and policy iteration, to solve dynamic programming problems for a finite number of states and a finite number of controls. Solving dynamic programming problems is difficult if the state/control space are infinite. This chapter discusses an important special case, that of the Linear Quadratic Regulator (LQR).

11.1 Discrete-time LQR

Imagine a dynamics that is linear in both the states and the controls

$$x_{k+1} = Ax_k + Bu_k; \quad \text{given } x_0$$

and the run-time cost is quadratic

$$q_k(x_k, u_k) = \frac{1}{2}\left(x_k^\top Q_k\, x_k + u_k^\top R_k\, u_k\right)$$

along with a quadratic terminal cost

$$q_f(x) = \frac{1}{2} x^\top Q_f\, x.$$

The matrices $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times p}$ are the autonomous and control parts of the dynamics f_k. The run-time cost is governed by our choices $Q_k \in \mathbb{R}^{n\times n}$ and $R_k \in \mathbb{R}^{p\times p}$: if we want to find optimal control trajectories such that some dimension of the state, say $x_k(i)$, does not have a large magnitude, we pick a large value on the diagonal entry Q_{ii} to achieve this. We will want that matrix Q is symmetric, positive semi-definite and R is symmetric, positive definite

$$Q = Q^\top \geq 0, \quad R = R^\top > 0.$$

This prevents the controller from driving down the cost-to-go to negative infinity by picking certain controls or staying at certain states. This problem is known as the Linear Quadratic Regulator (LQR).

Consider a deterministic, finite-horizon problem where the cost is given by

$$J(x_0) = \frac{1}{2} x_T^\top Q_f \, x_T + \sum_{k=0}^{T-1} \frac{1}{2} \left(x_k^\top Q_k \, x_k + u_k^\top R_k \, u_k \right).$$

As usual, our goal is to find a sequence of controls $(u_0, \ldots, u_{T-1})$ that minimizes the cost

$$J^*(x_0) = \min_{(u_0,\ldots,u_{T-1})} J(x_0; (u_0, \ldots, u_{T-1})).$$

We know the principle of dynamic programming now and can apply it to this problem. First set

$$J_T(x) = \frac{1}{2} x^\top Q_f \, x; \text{ for all } x.$$

$$\begin{aligned} J_{T-1}(x) &= \min_u \frac{1}{2} \Big\{ x^\top Q \, x + u^\top R \, u \Big\}_{T-1} + J_T(Ax + Bu) \\ &= \min_u \frac{1}{2} \Big\{ x^\top Q \, x + u^\top R \, u + (Ax + Bu)^\top Q_f (Ax + Bu) \Big\}_{T-1}. \end{aligned}$$

The $\{\}_{T-1}$ simply means that all quantities inside the curly brackets are sub-scripted at time $T-1$. We can now take the derivative of the right-hand-side with respect to the control u and set it to zero.

$$\begin{aligned} 0 &\equiv \frac{\mathrm{d}\,\text{RHS}}{\mathrm{d}u} \\ &= \frac{1}{2} \left\{ Ru + B^\top Q_f \, (Ax + Bu) \right\}_{T-1} \\ \Rightarrow u_{T-1}^* &= \left(R_{T-1} + B^\top Q_f B \right)^{-1} B^\top Q_f A \, x_{T-1} \\ &\equiv -K_{T-1} x_{T-1}. \end{aligned}$$

The second derivative is positive definite

$$\frac{\mathrm{d}^2\,\text{RHS}}{\mathrm{d}u^2} = R_{T-1} + B^\top Q_f B > 0$$

so u_{T-1}^* is a minimum of the RHS which is a convex function. The matrix K_{T-1} is called the Kalman gain. Notice that the optimal controller is linear in the state.

Observe that the optimal cost-to-go is a quadratic

$$\begin{aligned} J_{T-1}(x) &= \frac{1}{2}\left\{x^\top Q x + {u^*}^\top R u^* + (Ax + Bu^*)^\top Q_f\,(Ax + Bu^*)\right\}_{T-1} \\ &= \frac{1}{2}\, x_{T-1}^\top \left\{Q + K^\top R K + (A - BK)^\top Q_f\,(A - BK)\right\}_{T-1}\, x_{T-1} \\ &\equiv \frac{1}{2}\, x_{T-1}^\top P_{T-1} x_{T-1}. \end{aligned}$$

This suggests that at each timestep k we can assume

$$J_k(x) = \frac{1}{2} x^\top P_k\, x$$

and compute the optimal LQR controller via recursion as

$$\begin{aligned} &P_T = Q_f \\ &K_k = \left(R_k + B^\top P_{k+1}\, B\right)^{-1}\, B^\top P_{k+1}\, A \\ &P_k = Q_k + K_k^\top R_k\, K_k + (A - BK_k)^\top P_{k+1}\, (A - BK_k)\,, \\ &\text{for } k = T-1, T-2, \ldots, 0. \end{aligned} \tag{11.1}$$

These equations are similar to the update equations of the Kalman filter; in fact we will see shortly how spookily similar the two are. The key difference is that Kalman filter updates run forward in time starting from the initial covariance. It should not be surprising that because LQR is an optimal control problem, its update equations run backward in time.

LQR is among the most widely used controllers in the world (can you guess which *the* most widely used controller?). There are many amazing observations one can make about the LQR problem which makes it the cornerstone of linear optimal control theory.

1. The optimal controller
$$u_k^* = -K_k\, x_k$$
is linear in the state.

2. The matrices P_k and K_k do not depend on state x_k and can be computed offline, ahead of time.

3. For a finite-horizon problem K_k changes with k even if A, B, Q, R are all constant with time k.

11.1.1 Infinite horizon LQR

Similar to the case with infinite horizon dynamic programming the cost function P_k converges to a steady state P_{ss} as $k \to \infty$ if the dynamics (A, B) and the run-time cost (Q, R) does not change with time and if the dynamics (A, B) is stabilizable, i.e., if a certain control u cannot affect a state x_1 then the matrix A is such that the state converges to zero with time. The steady-state cost can be found by iterating

$$P_{ss} = Q + A^\top \left\{P_{ss} - P_{ss} B \left(R + B^\top P_{ss} B\right)^{-1}\, B^\top P_{ss}\right\}\, A. \tag{11.2}$$

This equation is called the Algebraic Riccati equation.

Question 1 (Linearization). Just like the extended Kalman filter, you can linearize a nonlinear dynamical system and execute the LQR controller for the linearization. If however you obtain a system of the form

$$x_{k+1} = A\, x_k + B\, u_k + c$$

for some constant c how can you use the LQR problem formulation? Similarly, if we have linear terms like

$$J = \frac{1}{2} \sum_{k=0}^{T-1} \left(x_k^\top Q\, x_k + u_k^\top R\, u_k + q^\top x_k + r^\top u_k \right).$$

how should one modify the LQR problem?

Question 2 (Abrupt changes in control). As we discussed, the control in our mathematical problem formulation (gas pedal in the car) may be only an abstract version of the actual control variables (the entire state of the engine). Drastically changing the control input may make take our dynamical model to states where it is not accurate, e.g, the noticeable lag in the car's response when you floor the gas pedal. We may wish to ensure that the optimal control u_k does not change quickly with time. How do we do this with LQR?

Question 3 (Tracking with LQR). A lot of problems in robotics will involve tracking a trajectory, e.g., you recorded a reference trajectory of states and corresponding controls $x_0^r, x_1^r, \ldots, x_T^r$ and $u_0^r, u_1^r, \ldots, u_{T-1}^r$ and would like the robot to repeat this trajectory if it starts from some other initial state. How should one modify the LQR problem to do this?

11.2 Hamilton-Jacobi-Bellman equation

An important abstraction one can make dynamic programming problems is assuming that the dynamics happens in continuous-time and is given by an ordinary differential equation

$$\dot{x} = f(x, u); \quad \text{for } x(0) = x_0.$$

You can think of this as the limit of the discrete-time dynamics $x_{k+1} = f(x_k, u_k)$ as the discretization time-interval goes to zero. Continuous-time dynamical systems are ordinary differential equations (ODEs https://en.wikipedia.org/wiki/Ordinary_differential_equation) and the trajectory of states and controls are functions of time t

$$\{x(t) :\ t \in \mathbb{R}_+\}, \quad \{u(t) :\ t \in \mathbb{R}_+\}.$$

Let us imagine that we want to find control sequences that minimize the objective

$$q_f(x(T)) + \int_0^T q(x(t), u(t))\, \mathrm{d}t;$$

this is again a sum of a terminal cost and a run-time cost. The run-time cost is expressed as an integral instead of a sum because we are in continuous-time. We want to solve for

$$J^*(x_0) = \min_{u(t),\ t\in\mathbb{R}_+} \left\{ q_f(x(T)) + \int_0^T q(x(t), u(t))\ \mathrm{d}t \right\}. \tag{11.3}$$

Notice that the minimization is over a function of time $u(t) : t \in \mathbb{R}_+$ as opposed to a sequence of controls $(u_0, \ldots, u_{T-1})$ that we had in the discrete-time case.

The dynamic programming principle is still valid, if we have an optimal control trajectory $u^*(t) : t \in \mathbb{R}_+$ we can chop it up into two parts at time intermediate time $t \in [0, T]$ and claim that the tail is optimal. We write this down as follows.

$$\begin{aligned} J^*(x(t), t) &= \min_{u(s),\ t\le s\le T} \left\{ q_f(x(T)) + \int_t^T q(x(s), u(s))\ \mathrm{d}s \right\} \\ &= \min_{u(s),\ t\le s\le T} \left\{ q_f(x(T)) + \int_t^{t+\Delta t} q(x(s), u(s))\ \mathrm{d}s + \int_{t+\Delta t}^T q(x(s), u(s))\ \mathrm{d}s \right\} \\ &= \min_{u(s),\ t\le s\le T} \left\{ J^*(x(t+\Delta t), t+\Delta t) + \int_t^{t+\Delta t} q(x(s), u(s))\ \mathrm{d}s \right\}. \end{aligned}$$

We can now take the Taylor approximation of the term $J^*(x(t + \Delta t), t + \Delta t)$ and substitute back in this equation to get

$$0 = \partial_t\ J^*(x(t), t) + \min_{u(t)\in U} \left\{ q(x(t), u(t)) + f(x(t), u(t))\ \partial_x J^*(x(t), t) \right\}. \tag{11.4}$$

The term $\partial_t J^*(x(t), t)$ is partial derivative of the function $J^*(x, t)$ with respect to its second argument evaluated at time t. Similarly the expression $\partial_x J^*(x(t), t)$ is the partial derivative evaluated with respect to its first argument at $x(t)$. Notice that the minimization in (11.4) is only over one control input $u(t) \in U$, the one we should take at time t.

This equation is called the Hamilton-Jacobi-Bellman (HJB) equation. It is the continuous-time version of the Bellman equation

$$J_k^*(x) = \min_{u_k\in U} \left\{ q_k(x, u_k) + J_{k+1}^*(f_k(x, u_k)) \right\}.$$

Just like the Bellman equation is solved backwards in time starting from T with $J_k^*(x) = q_f(x)$, the HJB equation is solved backwards in time by setting

$$J^*(x, T) = q_f(x).$$

11.3 Continuous-time LQR

Let us consider a linear continuous-time dynamics given by

$$\dot{x} = A\,x + B\,u; \quad x(0) = x_0.$$

We are interested in finding a control trajectory that minimizes, as usual, a cost function that is quadratic in states and controls

$$\frac{1}{2}\,x(T)^\top Q_f\,x(T) + \frac{1}{2}\int_0^T x(t)^\top Q\,x(t) + u(t)^\top R\,u(t)\,\mathrm{d}t.$$

This is a nice setup for using the HJB equation from the previous section. Let us use our intuition from the discrete-time LQR problem and say that the optimal cost is quadratic in the states, namely,

$$J^*(x,t) = \frac{1}{2}x^\top P(t)\,x(t);$$

notice that as usual the optimal cost-to-go is a function of the states x and the time t because is the optimal cost of the continuous-time LQR problem if the system starts at a state x at time t and goes on until time $T \geq t$. We want to check if this J^* satisfies the HJB equation

$$-\partial_t J^*(x,t) = \min_{u\in U}\left\{\frac{1}{2}\left(x^\top Q x + u^\top R\,u\right) + (A\,x + B\,u)^\top\,\partial_x J^*(x,t)\right\}$$

from (11.4). Notice that the minimization is over the control input that we take *at time* t. Also notice the partial derivaties

$$\partial_x J^*(x,t) = P(t)\,x.$$
$$\partial_t J^*(x,t) = \frac{1}{2}x^\top \dot{P}(t)\,x.$$

It is convenient in this case to see that the minimization can be performed using calculus, we differentiate with respect to u and set it to zero.

$$\begin{aligned} 0 &= \frac{\mathrm{d}\,\{\}}{\mathrm{d}u}\Big|_{u^*(t)} \\ \Rightarrow u^*(t) &= -R^{-1}\,B^\top P(t)\,x(t) \\ &\triangleq -K(t)\,x(t). \end{aligned} \tag{11.5}$$

where $K(t) = R^{-1}\,B^\top P(t)$ is the Kalman gain. The controller is again linear in the states $x(t)$ and the expression for the gain is very simple in this case, much simpler than discrete-time LQR. Since $R \succ 0$, we also know that $u^*(t)$ computed here is the global minimum. If we substitute this value of $u^*(t)$ back into the HJB equation we have

$$\{\}\,\Big|_{u^*(t)} = \frac{1}{2}x^\top\left\{PA + A^\top P + Q - PBR^{-1}B^\top P\right\}\,x.$$

If order to satisfy the HJB equation, we must have that the expression above is equal to $-\partial_t J^*(x,t)$. We therefore have the differential Riccati equation

$$-\dot{P} = PA + A^\top P + Q - PBR^{-1}B^\top P. \tag{11.6}$$

This is an ordinary differential equation in the matrix P. The derivative $\dot{P} = \frac{\mathrm{d}P}{\mathrm{d}t}$ stands for differentiating every entry of P individually with time t. The terminal cost is $\frac{1}{2}x(T)^\top Q_f\, x(T)$ which gives the boundary value for the ODE as

$$P(T) = Q_f.$$

Notice that the ODE for the $P(t)$ travels backwards in time.

11.4 Stochastic LQR

Let us now consider a stochastic LQR problem. We will exploit the continuous-time equations we just showed. The dynamics looks as

$$\dot{x} = Ax + Bu + B_w w; \quad x(0) = x_0.$$

The "noise" is a standard Gaussian random variable $w \sim N(0, I)$ and uncorrelated in time, i.e., $w(t)$ is independent of $w(t+s)$ for all times t, s. This is a "linear" Markov Decision Process.

Remark 4. The above notation is not very precise. Observe that $x(t)$ changes in infinetisimal increments and a different noise vector w at each time instant means that the trajectory $x(t)$ of the ODE written above is not continuous. The most appropriate way of writing the above equation is

$$x(t) = x_0 + \int_0^t \big(Ax(s) + Bu(s) + B_w w(s)\big)\, \mathrm{d}s$$

where $w(s) \sim N(0, I)$ is a Gaussian random variable at each instant s. The integral of Gaussian noise is continuous and everything is well-defined mathematically. Handling such notation is the subject of stochastic calculus and we will not worry about it here.

We are interested in minimizing cost of the form

$$\frac{1}{2} \mathop{\mathbb{E}}_{w(t):t\in[0,T]} \left[x(T)^\top Q_f\, x(T) + \frac{1}{2} \int_0^T x(t)^\top Q\, x(t) + u(t)^\top R\, u(t)\, \mathrm{d}t \right].$$

We want to find a control trajectory $u^*(t) :\ t \in [0,T]$ that minimizes the cost of the LQR problem when averaged over all realizations of the noise $w(t)$.

It is an *extremely surprising and non-intuitive fact* that the optimal control $u^*(t)$ is the same as that of (11.5)

$$u^*(t) = -K(t)\, x(t).$$

In other words, the optimal controller one should use for the stochastic LQR problem does not change from the deterministic LQR problem. This result is true only if the noise is Gaussian and the dynamics is linear, it is not true in general. The total cost incurred by this controller is

$$\begin{aligned} J^*(x_0, 0) &= \mathop{\mathbb{E}}_{w(t):t\in[0,T]} \left(\frac{1}{2} x(T)^\top Q_f \; x(T) + \int_0^T \ldots \; \mathrm{d}t \right) \\ &= \frac{1}{2} x_0^\top \; P(0) \; x_0 + \frac{1}{2} \int_0^T \text{ trace } \left[P(t) B_w B_w^\top \right] \; \mathrm{d}t. \end{aligned} \tag{11.7}$$

The first term is the same as that of the deterministic LQR problem. The second term is the penalty we incur for having a noisy dynamical system. This is the minimal cost achievable for continuous LQR but it is not the same as that of the deterministic LQR. Imagine if B_w is diagonal matrix, i.e., noise acts on all states. If there are certain states x_{ii} where $P_{ii}(t)$ is large and if B_{wii} is also non-zero, then those states contribute a lot to the stochastic LQR cost.

11.5 Linear Quadratic Gaussian

The previous section was an MDP where we have observations of the entire state $x(t)$ at each instant in time t, even if this state $x(t)$ changes stochastically and we do not really know the future noise vector $w(t)$. We said that the optimal control is $u^*(t) = -K(t) \; x(t)$. What should one do if we cannot observe the state exactly?

Imagine that we receive observations of the form

$$y(t) = Cx(t) + Dv.$$

where $v \sim N(0, I)$ is standard Gaussian noise that corrupts our observations y. Again the most appropriate way of writing this equation is $y(t) = y_0 + \int_0^t \; (Cx(s) + Dv) \; \mathrm{d}s$ but we will not worry about it. The Kalman filter is interested in estimating the expected value of the state $x(t)$ at time t given all the observations preceeding it

$$\widehat{x}(t) = \mathop{\mathbb{E}}_{v(s):s\in[0,t]} \Big[x(t) \mid y(s) : s \in [0, t] \Big].$$

The notation above is also a difficult object to define because the observation noise changes continuous, to be precise one should condition on the "filtration" $\mathcal{Y}(t)$; we will not worry about this.

There exists a continuous-time version of the Kalman filter, this is called the Kalman-Bucy filter. If the covariance of the estimate is

$$\Sigma(t) = \mathbb{E} \left[x(t) \; x(t)^\top \mid \mathcal{Y}(t) \right],$$

the Kalman-Bucy filter updates $\Sigma(t)$ using the differential equation

$$\dot{\Sigma} = B_w B_w^\top + A\Sigma + \Sigma A^\top - \Sigma \, C^\top \left(DD^\top \right)^{-1} \; C \, \Sigma; \quad \text{given } \Sigma(0).$$

This equation is very close to the Kalman filter equations you saw in Lecture 3. You can read more at https://en.wikipedia.org/wiki/Kalman_filter.

We can now plug in the Kalman-Bucy filter estimate $\widehat{x}(t)$ into the optimal controller for LQR

$$u^*(t) = -K(t)\,\widehat{x}(t). \tag{11.8}$$

This is called the Linear Quadratic Gaussian (LQG): it is just a Kalman/Kalman-Bucy filter operating in tandem with an LQR/stochastic-LQR controller.

It is very surprising that this choice, namely plugging in $\widehat{x}(t)$ if we do not know $x(t)$ exactly, is optimal. Again, this result is true for the case of linear dynamics, linear observations and Gaussian dynamics and observation noise; the result is not true for other cases. This choice is known as "certainty equivalence", a name has been borrowed from risk theory in finance.

11.6 Duality between the Kalman filter and the LQR

Given a dynamical system

$$\begin{aligned}\dot{x} &= Ax + Bu + B_w w\\ y &= Cx + Dv\end{aligned}$$

where w, v are Gaussian noise with identity covariance, we know that the LQR problem solves for the optimal controller for the stochastic system that minimizes

$$\mathop{\mathbb{E}}_{w}\left[\frac{1}{2}\int_0^T x(t)^\top Q x(t) + u(t)^\top R u(t)\,\mathrm{d}t\right]$$

and the Kalman filter solves for the optimal state estimate (with $u = 0$)

$$\mathop{\mathbb{E}}_{v}\Big[x(t) \mid y(s) : s \in [0,t]\Big]$$

conditioned on all past observations. The update equation for the covariance matrix Σ of the Kalman filter in continuous time is

$$\dot{\Sigma} = B_w B_w^\top + A\Sigma + \Sigma A^\top - \Sigma C^\top \left(DD^\top\right)^{-1} C\,\Sigma. \tag{11.9}$$

This version of the Kalman filter is called the Kalman-Bucy filter and you can read more about it at https://en.wikipedia.org/wiki/Kalman_filter. The matrix Σ is initialized to some reasonable value, this initialization is our estimate of the initial covariance of the state before the system starts producing observations. The LQR problem has a cost-to-go function

$$J^*(x,t) = \frac{1}{2}x^\top P(t)\,x$$

where the positive semi-definite matrix evolves from its terminal value $P(T) = Q_f$ as

$$-\dot{P} = PA + A^\top P + Q - PBR^{-1}B^\top P \tag{11.10}$$

using the differential Riccati equation.

Let us write the Kalman-Bucy filter's updates in terms of the inverse of Σ

$$\Sigma^{-1} = S.$$

The matrix S is called the information matrix. Use the fact that

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\Sigma^{-1}\right) = -\Sigma^{-1}\,\dot{\Sigma}\,\Sigma^{-1}$$

and rewrite (11.9) as

$$\begin{aligned} \dot{\Sigma} &= B_w B_w^\top + AS^{-1} + S^{-1}\,A^\top - S^{-1}C^\top\left(DD^\top\right)^{-1}\,C\,S^{-1} \\ \Rightarrow \dot{S} &= C^\top\left(DD^\top\right)^{-1}C - A^\top S - S\,A - SCC^\top S. \end{aligned} \tag{11.11}$$

This form is called the information-matrix form of the Kalman filter.

The two equations (11.10) and (11.11) look quite similar.

LQR	Kalman-Bucy filter
P	Σ^{-1}
A	$-A$
$BR^{-1}B^\top$	$B_w B_w^\top$
Q	$C^\top\left(DD^\top\right)^{-1}C$
t	$T-t$

This is a deep and surprising result that optimal control problems and state estimation problems are equivalent for linear dynamical systems with linear observations with Gaussian dynamics and observation noise. This result is not true in general. Effectively, you can solve for the Kalman filter using the DARE command in MATLAB/Python for solving for the LQR.

Let us analyze this equivalence. Notice that the inverse of the Kalman filter is like the cost matrix of LQR. The "dynamics" of the Kalman filter is the reserve of the dynamics of the LQR problem, this shows that the P matrix is updated backwards in time while the covariance Σ is updated forwards in time. The next identity

$$BR^{-1}B^\top = B_w B_w^\top$$

is very interesting. Imagine a situation where we have a fully-actuated system with $B = I$ and B_w being a diagonal matrix. This identity suggests that larger the control cost R_{ii} of a particular actuator i, lower is the noise of using that actuator $(B_w)_{ii}$, and vice-versa. This is how muscles in your body have evolved: muscles that are cheap to use (low R) are also very noisy in what they do whereas muscles that are expensive to use (large R) which are typically the biggest muscles in the body are also the least noisy and most precise. You can read more about this in the assigned reading Todorov and Jordan (2002). The next identity

$$Q = C^\top\left(DD^\top\right)^{-1}\,C$$

is related to the quadratic state-cost in LQR. Imagine the situation where both Q, D are diagonal matrices. If the noise in the measurements D_{ii} is large, this is equivalent to the state-cost matrix Q_{ii} being small; roughly there is no way we can achieve a low state-cost $x^\top Q x$ in our system that consists of LQR and a Kalman filter (this combination is known as Linear Quadratic Gaussian LQG as saw before) if there is lots of noise in the state measurements. The final identity

$$t = T - t$$

is the observation that we have made many times before: dynamic programming travels backwards in time and the Kalman filter travels forwards in time.

Remark 5. The duality between linear controls and linear state-estimation problems is powerful. A version of this duality was identitified by Kalman in the 60s and for the longest time it was believed that this duality is unique to linear systems. However, there is a large class of non-linear problems that also show such duality. You can also read Todorov (2008) to expand upon this section. This material—quite surprisingly—will form the basis for what is called soft-Q learning in Reinforcement Learning.

Remark 6. The Kalman duality was discovered and is usually presented in the following way. If we directly compare (11.10) and (11.9) we get the equivalence

LQR	Kalman-Bucy filter
P	Σ
A	$A^\top$
B	$C^\top$
R	$DD^\top$
Q	$B_w B_w^\top$
t	$T - t$

While the equations indeed match with these substitutions, if you follow the discussion above these equivalences do not really make sense, this version of the Kalman duality is a strong artifact of the linear setting and does not generalize as discussed in the previous remark.

11.7 Iterative LQR

Remark 7 (Tracking using LQR).

Question 8 (Does iLQR converge?).

Question 9 (Is the iLQR problem convex?).

Remark 10 (Differential dynamic programming).

11.7.1 Receeding horizon LQR

Remark 11 (Model Predictive Control).

Bibliography

Todorov, E. (2008). General duality between optimal control and estimation. In *2008 47th IEEE Conference on Decision and Control*, pages 4286–4292. IEEE.

Todorov, E. and Jordan, M. I. (2002). Optimal feedback control as a theory of motor coordination. *Nature neuroscience*, 5(11):1226–1235.

Lecture 13

Sampling-based Motion Planning

Reading

- Lavalle Chapter 5, 14, 15.3
- Kuffner and LaValle (2000)

Building systems in robotics

- A Perception-Driven Autonomous Urban Vehicle: http://acl.mit.edu/papers/LeonardJFR08.pdf
- MIT's Entry in the DARPA Robotics Challenge: https://ocw.mit.edu/resources/res-9-003-brains-minds-and-machines-summer-course-summer-2015/unit-8.-robotics/lecture-8.1-russ-tedrake-mits-entry-in-the-darpa-robotics-challenge/
- Mine Tunnel Exploration Using Multiple Quadrupedal Robots: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8990018

We will use the course notes hosted at https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-decision-making-fall-2010/lecture-notes/MIT16_410F10_lec15.pdf from Emilio Frazzoli (ETH/MIT) for this lecture. Please use the handwritten notes for the following sections. https://github.com/pratikac/smpl is a fast C++ library that implements the RRT* algorithm, you are encouraged to study this implementation. In practice you can use libraries like https://ompl.kavrakilab.org/ which implement a host of motion-planning algorithms.

13.1 Steering function

13.2 Nearest neighbor algorithms

13.3 Obstacles

13.4 Receeding horizon planning

13.5 Multiple agents

13.6 RRTs for stochastic systems

13.7 Some special dynamical systems

13.7.1 Dubins car

13.7.2 Reeds-Shepp car

13.7.3 Planning for quadrotor

13.7.4 Manipulator

13.8 Robotics as a system

Bibliography

Kuffner, J. J. and LaValle, S. M. (2000). Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 995–1001. IEEE.

Lecture 16

Imitation Learning

Reading

- An Algorithmic Perspective on Imitation Learning (https://arxiv.org/pdf/1811.06711.pdf)
- The DAGGER algorithm (https://www.cs.cmu.edu/ sross1/publications/Ross-AIStats11-NoRegret.pdf)
- https://www.youtube.com/watch?v=TUBBIgtQL_k

16.1 Background

This is the beginning of Module 3 of the course. The previous two modules have been about how to to estimate the state of the world around the robot (Module 1) and how to move the robot (or the world) to a desired state (Module 2). Both of these required that we maintain a model of the dynamics of the robot; this model may be inaccurate and we fudged over this inaccuracy by modeling the remainder as "noise" in Markov Decision Processes.

The next few lectures introduce different aspects of what is called Reinforcement Learning (RL). This is a very large field and you can think of using techniques from RL in many different ways.

1. **Dynamic programming with function approximation.** If you are doing dynamic programming, you can think of writing down the optimal cost-to-go $J^*(x,t)$ as a function of some parameters, e.g., the cost-to-go is

$$J_\varphi(x,t) = \frac{1}{2}x(t)^\top \underbrace{\Big(\text{some function of } A, B, Q, R\Big)}_{\text{function of } \varphi} x(t)$$

   for LQR. We know the stuff inside the brackets to be exactly $P(t)$ but, if we did not, it could be written down as some generic function of parameters φ. We know that any cost-to-go that satisfies the Bellman equation is the optimal cost-to-go, so we can now "fit" the candidate function $J_\theta(x,t)$

to satisfy the Bellman equation. Similarly, one may also approximate the policy π using some parameters θ.

2. **Learning from data.** It may happen that we do not know very much about the dynamical system, e.g., we do not know a good model for what drives customers as they buy items in an online merchandise platform, or a robot traveling in a crowded area may not have a good model for how large crowds of people walk around it. One may collect data from these systems fit some model of the form $\dot{x} = f(x, u)$ to the data and then go back to the techniques of Module 2. It is typically not clear how much data to collect. RL gives a suite of techniques to learn the cost-to-go in these situations by collecting and assimilating the data itself. These techniques go under the umbrella of policy gradients, on-policy methods etc. One may also simply "memorize" the data provided by an expert operator, this is called Imitation Learning.

We will look at a number of these methods in the coming lectures.

Imitation Learning is also called "learning from demonstrations". This is one of the earliest examples of using a neural network for doing robotics. The Autonomous Land Vehicle in a Neural Network (ALVINN) project at CMU by Dean Pomerleau in 1988 (https://www.youtube.com/watch?v=2KMAAmkz9go) used a two-layer neural network with 5 hidden neurons, about 1000 inputs from the pixels of a camera and 30 outputs. It successfully drove across the US and Germany. Imitation learning has also been responsible for numerous other early-successes in RL, e.g., acrobatic maneuvers on an RC helicopter (http://ai.stanford.edu/ acoates/papers/Abbeel-CoatesNg_IJRR2010.pdf).

Imitation Learning seeks to record data from experts, e.g., humans, and reproduce these desired behaviors on robots. The key questions we should ask are as follows:

1. Who should demonstrate (experts, amateurs, or novices) and how should we record data (what states, controls etc.)?

2. How should we learn from this data? e.g., fit a supervised regression model for the policy. How should one ignore bad behaviors in non-expert data?

3. And most importantly, what can we do if the robot encounters a situation which was not in the dataset. This is called covariate shift in machine learning.

16.2 A tour of Supervised Learning

Let us first see standard supervised learning where it is Nature that gives us data. On this data, it provides us with a prediction problem:

$$\underbrace{\text{input data}}_{X} \to \underbrace{\text{predictions}}_{Y}.$$

Nature often does not tell us what property of a datum $x \in X$ results in a particular prediction $y \in Y$. We would like to learn to imitate nature: predict y given x.

What does such learning mean? It is simply a notion of being able to identify patterns in the input data without explicitly programming a computer for prediction. We are often happy with a learning process that simply identifies correlations: if we learn correlations on a few samples $(x_1, y_1), \ldots, (x_n, y_n)$, we may be able to predict the output for a new datum x_{n+1}. We may not need to know *why* the label of x_{n+1} was predicted to be so and so.

Let us say that nature possesses a probability distribution $\mathcal{P}$ over (X, Y). It draws n iid samples from this distribution

$$D_{\text{train}} = \{(x_i, y_i)\}_{i=1}^{n}$$

where

$$(x_i, y_i) \sim \mathcal{P} \quad \forall i \leq n$$

and hands D_{train} to us as the "training set". If we wish to test our predictor on new data then nature gives us a "test set"

$$D_{\text{test}}.$$

The assumption that the distribution $\mathcal{P}$ generates both D_{train} and D_{test} is very important. This distribution provides coherence between past and future samples: past samples that were used to train and future samples that we will wish to predict upon.

What is the task in machine learning? Imagine D_{train} consists of $n = 50$ RGB images of size 100×100 of two kinds, ones with an orange inside them and ones without. 10^4 is a large number of pixels, each of possible intensities 255^3. It might happen that we discover one particular pixel, say at location $(25, 45)$, which takes distinct values in all images inside D_{train}. Here is a predictor: given a test datum the predictor outputs the label of the first image in the training set whose pixel $(25, 45)$ matches that of the test datum. Observe that this predictor achieves zero error on D_{train}. Why do you think this will not work well for images outside D_{train}? For instance, the lighting conditions or the viewpoint of future images might be different than the ones in D_{train}.

> Designing a predictor that is accurate on D_{train} is trivial. A hash function that memorizes the data is sufficient. This is NOT our task in machine learning. We want predictors that generalize to new data outside D_{train}.

How to generalize? If we never see data from outside D_{train} why should we hope to do well on it? The key is the distribution $\mathcal{P}$. It is to constrain the class of predictors we entertain in our search. If this class is too big we risk finding rules similar to the one constructed above: they are good on the training data but cannot generalize. If this class is too small we never even predict on the training data well, surely the performance on the test data will be equally bad. Finding the right class of functions to fit the data is known as the model selection problem. Regularization in machine learning is a technique that will help constrain large model classes.

16.3 Behavior Cloning

Let us imagine that we are given access to n trajectories each of length $T+1$ time-steps from an expert demonstrator. We write this as a dataset

$$D = \left\{(x_t^i, u_t^i)_{t=0,1,\ldots,T}\right\}_{i=1,\ldots,n}$$

At each step, we record the state x_t^i and the control the expert took at that state u_t^i.

How to represent the controller? We would like to learn a deterministic feedback control for the robot that is parametrized by parameters θ

$$u_\theta(x) : X \mapsto U.$$

This may represent many different families of controllers. For example, $u_\theta(x) = \theta x$ where $\theta \in \mathbb{R}^{d\times p}$ is a linear controller much like the control for LQR. We could think of some other complicated function like

$$u_\theta(x) = \theta_2\, \sigma\left(\theta_1 x\right)$$

where $\theta_1 \in \mathbb{R}^{p\times m}$ and $\theta_2 \in \mathbb{R}^{m\times p}$ and $\sigma : \mathbb{R}^m \mapsto \mathbb{R}^m$ is some nonlinear function, say a sigmoid. The parameters are the union of the two parameters on each layer $\theta = \{\theta_1, \theta_2\}$. This is really a two-layer neural network.

How to fit the controller? Given our chosen model (also known as the architecture) for $u_\theta(x)$, fitting the controller involves finding the best value for the parameters θ such that $u_\theta(x_t^i) \approx u_t^i$ for data in our dataset. There are many ways to do this, e.g., we can solve the following optimization problem

$$\widehat{\theta} = \operatorname*{argmin}_\theta \frac{1}{n(T+1)} \sum_{t=0}^{T} \sum_{i=1}^{n} \|u_t^i - u_\theta(x_t^i)\|_2^2 \tag{16.1}$$

The difficulty of solving the above problem depends upon how difficult the model $u_\theta(x)$ is, for instance, if the model is linear $\theta\, x$, we can solve (16.1) using ordinary least squares. If the model is a neural network, one would have to use iterative methods to solve the optimization problem above.

Question 1 (Data in BC is not iid). Standard supervised learning makes the assumption that Nature gives training data that is independent and identically distributed from the distribution $\mathcal{P}$. Is the data from an expert iid? So is the objective in (16.1) the correct way to fit a policy? Can you suggest ways of fixing correlations in the data?

Question 2 (Generalization performance of BC). If we fit the model $u_{\widehat{\theta}}$ well, it will have small prediction error on states x_t^i in the dataset. Can we use this model to run a real robot?

16.3.1 Behavior cloning with a stochastic controller

We have always considered feedback controllers that are deterministic as yet, i.e., there is a single value of control u that is taken at the state x. Going forward, we will also talk about stochastic controllers. They will be denoted by

$$u \sim u_\theta(\cdot \mid x);$$

in this case, $u_\theta(\cdot \mid x)$ is a probability distribution on the control space U that depends on the state x, and in this case the parameters θ. The control taken at a state x is a sample drawn from this probability distribution. The deterministic controller is a special case of this setup where

$$u_\theta(\cdot \mid x) = \delta_{u_\theta(x)}$$

is a Dirac-delta distribution at $u_\theta(x)$. If the control space U is discrete, $u_\theta(\cdot \mid x)$ could be a categorical distribution, if the control space U is continuous, you may wish to think of the controls being sampled from a Gaussian distribution with some mean $\mu_\theta(x)$ and variance $\sigma^2_\theta(x)$.

Let's pick a particular stochastic controller, say a Gaussian. How should we fit the parameters θ for this? We would like to find parameters θ that make the expert's data in our dataset very likely. The log-likelihood of each datum is

$$\log u_\theta(u^i_t \mid x^i_t)$$

and maximizing the log-likelihood of the entire dataset amounts to solving

$$\widehat{\theta} = \underset{\theta}{\text{argmin}} \ \frac{1}{n(T+1)} \sum_{t=0}^{T} \sum_{i=1}^{n} - \log u_\theta(u^i_t \mid x^i_t). \tag{16.2}$$

Remark 3 (Fitting BC with a Gaussian controller). Notice that if we use a Gaussian distribution

$$u_\theta(\cdot \mid x) = N\left(\mu_\theta(x),\ 1\right)$$

as our stochastic controller, the objective in (16.2) is the same as that in (16.1). To elaborate upon this, if

$$u_\theta(\cdot \mid x) = N\left(\mu_\theta(x), \sigma^2_\theta(x)\right)$$

we have that

$$- \log u_\theta(u \mid x) = \frac{\|\mu_\theta(x) - u\|_2^2}{\sigma^2_\theta(x)} \ + p \ \log \sigma_\theta(x).$$

Remark 4 (KL-divergence form of BC). We can slightly generalize the development in this section by thinking of the expert as using a controller $u_{\theta^*}(\cdot \mid x)$. The data is drawn by running this controller for n trajectories, $T+1$ time-steps on the system. This dataset is therefore a sample from

$$p_{u_{\theta^*}}(x, u)$$

which is the joint distribution on states and controls. The candidate distribution is $p_{u_\theta}(x, u)$ and the objective in (16.2) can also be written as

$$\widehat{\theta} = \underset{\theta}{\text{argmin}} \ \text{KL}\left(p_{u_{\theta^*}} \ || \ p_{u_\theta}\right) \tag{16.3}$$

Written this way, BC can be understood as finding a controller $\widehat{\theta}$ whose distribution on the states and controls is close to the distribution of states and controls of the expert.

Some background on KL-divergence. For two distributions $p(x)$ and $q(x)$ supported on a discrete

set X, the Kullback-Leibler divergence between them is given by

$$\mathrm{KL}(p \,||\, q) = \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)}. \tag{16.4}$$

This formula is well-defined only if for all x where $q(x) = 0$, we also have $p(x) = 0$. The KL divergence is a measure of the distance between two distributions. Notice that it is not symmetric

$$\mathrm{KL}(q \,||\, p) = \sum_{x \in X} q(x) \log \frac{q(x)}{p(x)} \quad \neq \mathrm{KL}(p \,||\, q).$$

Therefore the KL divergence is not a metric. It is always positive however which you can show using an application of Jensen's inequality. For distributions with continuous support, we integrate over the entire space X and define KL divergence as

$$\mathrm{KL}(p \,||\, q) = \int_{\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x.$$

Remark 5 (Worst-case performance of BC). Performance of Behavior Cloning can be quite bad in the worst case. The authors in Ross and Bagnell (2010) show under certain stylistic models that if the learnt controller $u_{\widehat{\theta}}$ differs from the control taken by the expert controller u_{θ^*} with a probability ϵ at each time-step, over a horizon of length T time-steps, it can be $\mathcal{O}(T^2\epsilon)$ off from the cost-to-go of the expert *as averaged over states that the learnt controller visits*. This is because once the robot makes a mistake and goes away from the expert's part of the state-space, the future trajectories of the robot and the expert can be very different.

Remark 6 (Model-free nature of BC). Observe that we have learnt a controller $u_{\widehat{\theta}}(\cdot \mid x)$ that is, as usual, a feedback controller and works for entire state-space X. We do not have any idea about the dynamics of the system however and did not need to know the dynamics in order to arrive at the controller. The data from the expert is conceptually the same as the model $\dot{x} = f(x, u)$ of the dynamics, and you can learn controllers from both. Do you notice a catch?

16.4 DAgger: Dataset Aggregation

The dataset in Behavior Cloning determines the quality of the controller learnt. If we collected very few trajectories from the expert, they may not cover all parts of the state-space and the behavior cloning controller has no data to fit the model in those parts.

Let us design a simple algorithm, of the same spirit as iterative-LQR. We start with a candidate controller, say $u_{\theta_0}(x)$; you can also start with a stochastic controller $u_{\theta_0}(\cdot \mid x)$ if you wish.

DAgger: Initialize dataset D to be the data of the expert. Initialize $u_{\theta_0} = u_{\widehat{\theta}}$ to be the BC controller for the expert's data. At iteration k

1. Pick a controller $u(x)$ which uses the expert's controller u_{θ^*} for fraction p of the time-steps

and uses u_{θ_k} for the other time-steps

$$u \sim p\, \delta_{u_{\theta^*}(x)} + (1-p)\, \delta_{u_{\theta_k}(x)}.$$

2. Use $u(x)$ to collect a dataset $D_k = \left\{(x_t^i, u_t^i)_{t=0,\dots,T}\right\}_{i=1,\dots,n}$
3. Set the new dataset to be $D \leftarrow D \cup D_k$
4. Fit a controller $u_{\theta_{k+1}}$ to the dataset D

This algorithm iteratively updates the BC controller $u_{\widehat{\theta}}$ by drawing new data from the system. The algorithm bootstraps off the expert's data. At each iteration, it runs the controller on the real system and adds the states and the controls to the dataset. In the beginning we may wish to be close to the expert's data and use a large value of p, as the fitted controller $u_{\theta_{k+1}}$ becomes good, we can reduce the value of p and rely less on the expert.

Remark 7. DAgger is an iterative algorithm which expands the controller to handle larger and larger parts of the state-space. Therefore, the cost-to-go of the controller learnt via DAgger is $\mathcal{O}(T)$ off from the cost-to-go of the expert *as averaged over states that the learnt controller visits.*

Question 8. What criterion can we use to stop these iterations? We can stop when the incremental dataset collected D_k is not that different from the cummulative dataset D, we know that the new controllers are not that different. We can also stop when the parameters $\theta_{k+1} \approx \theta_k$.

Remark 9 (DAgger with expert annotations at each step). We can also cook up a version of DAgger where we start with the BC controller $u_{\theta_0} = u_{\widehat{\theta}}$ and at each step, we run the controller on the real system and ask the expert to relabel the data, i.e., record the states x_t^i for our trajectories in step 2 of DAgger and put $u_{\theta^*}(x_t^i)$ in the dataset instead of our chosen control u_t^i. The dataset D collected by the algorithm expands at each iteration and although the states x_t^i are those visited by our controller, their annotations are those given by the expert.

Bibliography

Ross, S. and Bagnell, D. (2010). Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 661–668.

Lecture 17

Policy Gradient

Reading

- Sutton & Barto, Chapter 9–10, 13
- Simple random search provides a competitive approach to reinforcement learning at https://arxiv.org/abs/1803.07055
- Proximal Policy Optimization Algorithms https://arxiv.org/abs/1707.06347
- Are Deep Policy Gradient Algorithms Truly Policy Gradient Algorithms? https://arxiv.org/abs/1811.02553
- Asynchronous Methods for Deep Reinforcement Learning http://proceedings.mlr.press/v48/mniha16.pdf

---

This lecture discusses methods to learn the controller that minimizes a given cost functional over trajectories of an unknown dynamical system. These methods use what is called the "policy gradient" which will be the main section of this lecture. Recall from the previous class that we were able to fit stochastic controllers of the form $u_\theta(\cdot \mid x)$ that is a probability distribution on the control-space U for each $x \in X$. We fitted u_θ using data from the expert in imitation learning. We did not learn the cost-to-go for the fitted controller, like we did in the lectures on dynamic programming. This is a clever choice: it is often easier to learn the controller in a typical problem than to *learn* the cost-to-go as a parametric function $J_\theta(x)$.

Question 1. Can you give an instance when we have computed a controller previously in the class without coming up with its cost-to-go?

17.1 Setup

In this and the next few lectures we will always consider discrete-time stochastic dynamical systems with a stochastic controller

$$\begin{aligned} x_{k+1} &\sim p(\cdot \mid x_k, u_k) \text{ with noise denoted by } \xi_k \\ u_k &\sim u_\theta(x_k). \end{aligned}$$

We are interested in maximizing the expected value of the cummulative rewards over an infinite-horizon over trajectories of the stochastic system

$$J(\theta) = \mathop{\mathbb{E}}_{\xi_0,\, \xi_1, \ldots} \left[\sum_{k=0}^{\infty} \gamma^k \, r(x_k, u_k) \;\middle|\; x_0, u_k \sim u_\theta(\cdot \mid x_k) \right].$$

This is the same setup as that in the module on dynamic programming where we thought of the cost-to-go instead of the cummulative rewards. You can think of the reward as simply the negative of the cost and vice versa. Note that the expectation is only a function of the parameters of the controller θ. Finding the best controller therefore amounts to solving for

$$\widehat{\theta} = \operatorname*{argmax}_{\theta} J(\theta). \tag{17.1}$$

Question 2. Given the best policy with parameters $\widehat{\theta}$ how will you compute $J(\theta)$ numerically? We can sample n trajectories from the system and compute the empirical estimate of the expectation

$$\widehat{J}(\theta) \approx \frac{1}{n} \sum_{i=0}^{n} \sum_{k=0}^{T} \gamma^k r(x_k^i, u_k^i)$$

for some large time-horizon T. Contrast this with the complexity of policy evaluation which was simply a system of linear equations; evaluating the policy without having access to the dynamical system is harder.

17.1.1 Trajectory space

We know that the probability of the next state x_{k+1} given x_k is $p(x_{k+1} \mid x_k, u_k)$. The probability of taking the control u_k at state x_k is $u_\theta(u_k \mid x_k)$. Let us denote an infinite trajectory by

$$\tau = x_0, u_0, x_1, u_1, \ldots.$$

The probability of observing this entire trajectory is then

$$p(\tau) = \prod_{i=0}^{\infty} p(x_{k+1} \mid x_k, u_k) \; u_\theta(u_k \mid x_k).$$

or the log-likelihood of the trajectory is

$$\log p(\tau) = \sum_{i=0}^{\infty} \Big(\log p(x_{k+1} \mid x_k, u_k) + \log u_\theta(u_k \mid x_k) \Big).$$

The summation

$$R(\tau) = \sum_{k=0}^{\infty} \gamma^k \, r(x_k, u_k)$$

is called the **discounted return of the trajectory** τ. Sometimes we will also talk of the undiscounted return of the trajectory which is the sum of the rewards up to some fixed finite horizon T without the

discount factor pre-multiplier.

Remark 3. Observe what is probably the most important point in policy-gradient based reinforcement learning: the probability of trajectory is an infinite product of terms all of which are smaller than 1 (they are probabilities), so it is essentially zero even if the state-space and the control-space are finite (even if they are small). Any given infinite (or long) trajectory is very rare under the controller. Policy-gradient methods sample lots of trajectories from the system and average the returns across these trajectories. Since the set of trajectories of even a small MDP is so large, sampling lots of trajectories, or even the most likely ones, is very hard. This is a key challenge in getting RL algorithms to work.

17.2 Cross-entropy Method

Let us first consider a simple method to compute the best controller. Remember that if we are solving

$$\widehat{\theta} = \underset{\theta}{\text{argmax}}\, J(\theta)$$

we can do so using gradient descent and run a sequence of updates

$$\theta_{k+1} = \theta_k + \eta\, \nabla J(\theta).$$

where the step-size is $\eta > 0$ and $\nabla J(\theta)$ is the gradient of the objective $J(\theta)$ with respect to the parameters θ. In lieu of computing $\nabla J(\theta)$ which we will do in the next section, let us simply compute the gradient using a finite-difference approximation. The i^{th} entry of the gradient is

$$\begin{aligned}(\widehat{\nabla} J(\theta))_i &= \frac{J(\theta + \epsilon e_i) - J(\theta - \epsilon e_i)}{2\epsilon} \\ &\approx \frac{\widehat{J}(\theta + \epsilon e_i) - \widehat{J}(\theta - \epsilon e_i)}{2\epsilon}.\end{aligned}$$

where $e_i = [0, 0, \ldots, 0, 1, 0, \ldots]$ with 1 on the i^{th} entry. We compute all entries of the objective using this approximation and update the parameters using

$$\theta_{k+1} = \theta_k + \eta \widehat{\nabla}\, J(\theta_k).$$

Question 4. How many trajectories does finite-differences take? Can you do something more clever to estimate the gradient? Instead of picking perturbations e_i along the cardinal directions, let us sample them from a Gaussian distribution

$$e \sim N(0, \sigma^2 I)$$

for some chosen σ. We can however no longer use the finite-difference formula to compute the derivative because the noise e is not aligned with the axes. We can however use a Taylor series approximation to see that

$$J(\theta + e) \approx J(\theta) + \langle \nabla J(\theta),\, e \rangle$$

where $\langle \cdot, \cdot \rangle$ is the inner product. Given n samples $e^1, \ldots, e^n$ observe that

$$\begin{aligned}
\widehat{J}(\theta + e^1) &= \widehat{J}(\theta) + \left\langle \nabla J(\theta),\ e^1 \right\rangle \\
\widehat{J}(\theta + e^2) &= \widehat{J}(\theta) + \left\langle \nabla J(\theta),\ e^2 \right\rangle \\
&\vdots \\
\widehat{J}(\theta + e^n) &= \widehat{J}(\theta) + \left\langle \nabla J(\theta),\ e^n \right\rangle .
\end{aligned}$$

which is a linear system of equations in $\nabla J(\theta)$. The quantities $\widehat{J}$ are estimated as before using trajectories drawn from the system. We solve this linear system to get an estimate of the gradient $\widehat{\nabla} J(\theta)$.

The Cross-Entropy Method is a more crude but simpler to implement version of the above least squares formulation. At each iteration it updates the parameters using the formula

$$\theta_{k+1} = \mathop{\mathbb{E}}_{\theta \sim N(\theta_k, \sigma^2 I)} \left[\theta \, \mathbf{1}_{\left\{ \widehat{J}(\theta) > \widehat{J}(\theta_k) \right\}} \right]. \tag{17.2}$$

The CEM therefore samples a few parameters of the policy θ from a Gaussian (or really any other distribution) centered around the current parameters θ_k and updates the parameters in the direction if it leads to an increase $\widehat{J}(\theta) > \widehat{J}(\theta_k)$ to be maximized.

Question 5. The CEM seems to be a particularly bad method to maximize $J(\theta)$. It is reasonable to expect that it does not work well for all problems. What problems do you expect it to work well for? The assigned reading for today's lecture is a paper which demonstrates a very fast implementation of the CEM.

17.3 Likelihood-ratio-based methods

Consider the case where we are solving an optimization problem of the form

$$\min_{\theta} \mathop{\mathbb{E}}_{\xi} \left[f(\theta, \xi) \right]$$

where the function to be minimized depends on two arguments (θ, ξ); the first argument are some parameters θ and the second argument is ξ which is a stochastic quantity beyond our control (think of the noise in the MDP). The expectation on the outside is over the argument ξ which makes the entire objective only a function of the parameter θ. If we were to solve this problem using gradient descent, we could imagine running

$$\theta_{k+1} = \theta_k - \eta \, \nabla_\theta \left\{ \mathop{\mathbb{E}}_{\xi} \left[f(\theta_k, \xi) \right] \right\}.$$

Expectation is a linear operator and we can push the gradient ∇_θ inside the expectation to write the second term as

$$\mathop{\mathbb{E}}_{\xi} \left[\nabla_\theta f(\theta_k, \xi) \right]$$

which is simply the average of the standard gradient $\nabla_\theta f(\theta, \xi)$ over the stochastic variable in the problem ξ.

Imagine if the problem were instead to minimize

$$\min_\theta \mathop{\mathbb{E}}_{\xi \sim p_\theta(\cdot)} [f(\xi)]$$

where there is only one variable in the problem $\xi \sim p_\theta(\cdot)$ whose distribution is parametrized by the parameters θ. We can no longer push the gradient into the expectation easily because the distribution over which the expectation is computed also depends on the parameters θ. Essentially, we would like to do the chain rule of calculus but where one of the functions in the chain is an expectation. The likelihood-ratio trick described next allows us to take such derivatives.

Here is how the computation goes

$$\begin{aligned}
\nabla \mathop{\mathbb{E}}_{\xi \sim p_\theta(\cdot)} [f(\xi)] &= \nabla \int f(\xi)\, p_\theta(\xi)\, \mathrm{d}\xi \\
&= \int f(\xi)\, \nabla p_\theta(\xi)\, \mathrm{d}\xi \quad \text{(move the gradient inside, integral is over samples } \xi \text{ which do not depend on } \theta) \\
&= \int f(\xi)\, p_\theta(\xi)\, \frac{\nabla p_\theta(\xi)}{p_\theta(\xi)}\, \mathrm{d}\xi \\
&= \int f(\xi)\, p_\theta(\xi)\, \nabla \log p_\theta(\xi)\, \mathrm{d}\xi \\
&= \mathop{\mathbb{E}}_{\xi \sim p_\theta(\xi)} \left[f(\xi)\, \nabla \log p_\theta(\xi) \right].
\end{aligned} \tag{17.3}$$

This is called the likelihood-ratio trick to compute the policy gradient. It simply multiplies and divides by the term $p_\theta(\xi)$ and rewrites the term $\frac{\nabla p_\theta}{p_\theta} = \nabla \log p_\theta$.

Remark 6 (Variance of policy gradient). How will you implement the policy gradient on a computer? Can you notice any problems with this implementation? Observe

$$\begin{aligned}
\nabla \mathop{\mathbb{E}}_{\xi \sim p_\theta(\cdot)} [f(\xi)] &= \mathop{\mathbb{E}}_{\xi \sim p_\theta(\xi)} \left[f(\xi)\, \frac{\nabla\, p_\theta(\xi)}{p_\theta(\xi)} \right] \\
&\approx \frac{1}{n} \sum_{i=1}^{n} f(\xi_i)\, \frac{\nabla\, p_\theta(\xi_i)}{p_\theta(\xi_i)}.
\end{aligned}$$

High variance of the computing the gradient using samples ξ_i for the policy gradient is a characteristic of the likelihood-ratio trick. The denominator is culprit. Trajectories that have low likelihood better have low returns or low gradient under the model class. If none of these conditions are true, an estimator for the gradient will have high variance because the integrand takes values in a large range. This phenomenon is very common in machine learning, evaluating the log-partition function for probabilistic models is hard because the integrand may take values across multiple magnitudes.

Question 7. Do you know any other way of computing gradients of expressions where the parameters are involved in the expectation?

17.3.1 Expression for the policy gradient

Let us get back to reinforcement learning. We can write the objective $J(\theta)$ in short as

$$J(\theta) = \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \Big[R(\tau) \Big].$$

Observe carefully, this is very bad notation but is unfortunately rampant in the literature on reinforcement learning. The expectation is being taken over the space of trajectories. We have indicated that this distribution of trajectories of the system depends on the parameters of the stochastic controller θ that we wish to optimize. The integrand is $R(\tau)$ which is the discounted return of the infinite-horizon trajectory. We can now easily use the expression for the gradient in the previous expression to get

$$\begin{aligned}
\nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \Big[R(\tau) \, \nabla \log p(\tau;\theta) \Big] \\
&= \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \left[R(\tau) \, \nabla \left(\sum_{k=0}^{T} \log u_{\theta_k}(u_k \mid x_k) + \log p(x_{k+1} \mid x_k, u_k) \right) \right] \\
&= \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \left[R(\tau) \, \left(\sum_{k=0}^{T} \nabla \log u_{\theta_k}(u_k \mid x_k) \right) \right] \\
&= \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \left[\left(\sum_{k=0}^{T} \gamma^k r(x_k, u_k) \right) \, \left(\sum_{k=0}^{T} \nabla \log u_{\theta_k}(u_k \mid x_k) \right) \right].
\end{aligned} \tag{17.4}$$

The gradient of the dynamics with respect to θ is zero because the dynamics does not depend on θ. Note that for things to be well-defined we have eschewed the infinite-horizon expressions and replaced them with the finite-horizon version; if you don't think like this think of our entire RL setup as being a finite-horizon formulation with $\gamma = 1$.

We can again estimate this gradient using n trajectories from the system

$$\tau^i = x_0^i, u_0^i, x_1^i, u_1^i, \ldots.$$

using the expression

$$\widehat{\nabla} J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{k=0}^{T} \gamma^k r(x_k^i, u_k^i) \right) \, \left(\sum_{k=0}^{T} \nabla \log u_{\theta_k}(u_k^i \mid x_k^i) \right)$$

and update our parameters θ_k using

$$\theta_{k+1} = \theta_k + \eta \, \widehat{\nabla} J(\theta).$$

17.4 Variance reduction of the policy gradient

Remark 8 (Why do we need variance reduction). Consider an optimization problem

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} f(\theta; \xi_i)$$

where ξ_i is the i^{th} datum in a dataset of n samples and we are minimizing the loss (say, the classification error) $f(\theta; \xi_i)$. This is the empirical version of the stochastic optimization problem where we were averaging over a random variable ξ. Gradient descent updates the parameters using

$$\begin{aligned}\theta_{k+1} &= \theta_k - \eta \, \nabla_\theta \left(\frac{1}{n} \sum_{i=1}^{n} f(\theta; \xi_i) \right) \\ &= \theta_k - \eta \, \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta f(\theta; \xi_i).\end{aligned}$$

Updating the parameters θ_k at each iteration therefore involves computing the gradient of the classifier $f(\theta_k; \xi_i)$ on every datum ξ_i and averaging these gradients. The step-size η is typically chosen as a hyper-parameter in machine learning (advanced optimization methods such as BFGS will do a clever job of picking η but they are slow if n is large).

Stochastic gradient descent (SGD) makes the observation that it may not be necessary to compute the gradient on all the n data because we don't know the best step-size to use anyway. SGD samples an index $\omega_t \in \{1, \ldots, n\}$ uniformly randomly at each iteration and updates the parameters using

$$\theta_k - \eta \, \nabla_\theta f(\theta; \, \xi_{\omega_t}).$$

It thereby requires only one gradient evaluation per iteration; each iteration is much faster than an iteration of gradient descent. Typically you do mini-batch SGD where you sample $\omega_t^1, \ldots, w_t^{\mathscr{b}} \in \{1, \ldots, n\}$ uniformly randomly and update

$$\theta_k - \eta \, \frac{1}{\mathscr{b}} \sum_{i=1}^{\mathscr{b}} \nabla_\theta f(\theta; \, \xi_{\omega_t^i}).$$

for some chosen mini-batch size $\mathscr{b} > 0$.

There is no free lunch, SGD is much slower to converge than GD; it takes many more steps to minimize the objective. Roughly speaking, if the size of the dataset n is small, GD is better and if n is large, SGD requires fewer CPU cycles. The inefficiency of SGD depends on the variance of the gradient

$$D = \operatorname*{Var}_{\omega_t} \left(\nabla_\theta f(\theta; \, \xi_{\omega_t}) - \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta f(\theta; \xi_i) \right);$$

this is the expression for $\mathscr{b} = 1$. If this is large, the gradient used in SGD is very inaccurate as compared to its mean gradient which is the gradient of GD.

Accelerating SGD therefore relies on making its stochastic gradient more accurate. Variance reduction

techniques are a mechanism to accelerate SGD and are of critical importance when implementing policy gradient-based RL.

17.4.1 Control variates

You will perhaps appreciate that computing the accurate policy gradient is very hard in practice. This is a general concept from the literature on Monte-Carlo integration. It is typically introduced as follows. Say we have a random variable X and we would like to guess its expected value $\mu = \mathbb{E}[X]$. Note that X is an unbiased estimator of μ but it may have a large variance. If we have another random variable Y with known expected value $\mathbb{E}[Y]$, then

$$\widehat{X} = X + c(Y - \mathbb{E}[Y]) \tag{17.5}$$

is also an unbiased estimator for μ for any value of c. The variance of $\widehat{X}$ is

$$\text{Var}(\widehat{X}) = \text{Var}(X) + c^2\,\text{Var}(Y) + 2c\;\text{Cov}(X, Y).$$

which is minimized for

$$c^* = -\frac{\text{Cov}(X, Y)}{\text{Var}(Y)}$$

to

$$\begin{aligned}\text{Var}(\widehat{X}) &= \text{Var}(X) - c^{*2}\;\text{Var}(Y) \\ &= \left(1 - \left(\frac{\text{Cov}(X, Y)}{\text{Var}(Y)}\right)^2\right)\text{Var}(X).\end{aligned}$$

By subtracting $Y - \mathbb{E}[Y]$ from our observed random variable X, we have reduced the variance of X if the correlation between X and Y is non-zero. Most importantly, note that no matter what Y we plug into the above expression, we can never increase the variance of X; the worst that can happen is that we pick a Y that is completely uncorrelated with X and end up achieving nothing. Control variates are at the heart of a number of algorithms in machine learning and statistics.

17.4.2 Building a baseline

The simplest baseline one can build is to subtract a constant from the return. Consider the PG given by

$$\begin{aligned}\nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim p(\tau;\theta)}\Big[R(\tau)\;\nabla \log p(\tau;\theta)\Big] \\ &= \mathop{\mathbb{E}}_{\tau\sim p(\tau;\theta)}\Big[\,(R(\tau) - b)\;\nabla \log p(\tau;\theta)\Big].\end{aligned}$$

This is so because

$$\begin{aligned}\mathop{\mathbb{E}}_{\tau\sim p(\tau;\theta)}\big[b\,\nabla\log p(\tau;\theta)\big] &= \int \mathrm{d}\tau\; b\; p(\tau;\theta)\nabla\log p(\tau;\theta)\\ &= \int \mathrm{d}\tau\; b\;\nabla p(\tau;\theta)\\ &= b\,\nabla\int \mathrm{d}\tau\; p(\tau;\theta)\\ &= b\,\nabla 1\\ &= 0.\end{aligned}$$

Remark 9. What is the simplest b we can cook up? Let us write the mini-batch version of the policy gradient

$$\widehat{\nabla}J(\theta) = \frac{1}{\cancel{b}}\sum_{i=1}^{\cancel{b}}\;\big[R(\tau^i)\;\nabla\log p(\tau^i;\theta)\big].$$

We can set

$$b = \frac{1}{\cancel{b}}\sum_{i=1}^{\cancel{b}}\;R(\tau^i)$$

can use the variance-reduced gradient

$$\widehat{\nabla}J(\theta) = \frac{1}{\cancel{b}}\sum_{i=1}^{\cancel{b}}\;\big[\;\big(R(\tau^i)-b\big)\;\nabla\log p(\tau^i;\theta)\big].$$

This is a one-line change in your code for policy gradient so there is no reason not to do it.

Remark 10. What is the **best** constant b we can use? This involves a similar computation that we did for the control variate example. You minimize the variance of the gradient estimate

$$\begin{aligned}\mathrm{Var}\left(\widehat{\nabla}_{\theta_i}J(\theta)\right) &= \mathop{\mathbb{E}}_{\tau}\big[\;((R(\tau)-b_i)\nabla_{\theta_i}\log p(\tau;\theta))^2\;\big] - \left(\mathop{\mathbb{E}}_{\tau}\big[\;((R(\tau)-b_i))\,\nabla_{\theta_i}\log p(\tau;\theta)\big]\right)^2\\ &= \mathop{\mathbb{E}}_{\tau}\big[\;((R(\tau)-b_i)\nabla_{\theta_i}\log p(\tau;\theta))^2\;\big] - \left(\widehat{\nabla}_{\theta_i}J(\theta)\right)^2.\end{aligned}$$

Set

$$\frac{\mathrm{Var}\left(\widehat{\nabla}_{\theta_i}J(\theta)\right)}{\mathrm{d}b_i} = 0$$

in the above expression to get

$$b_i = \frac{\mathbb{E}_t\left[\left(\nabla_{\theta_i}\log p(\tau;\theta)\right)^2 R(\tau)\right]}{\mathbb{E}_\tau\left[\left(\nabla_{\theta_i}\log p(\tau;\theta)\right)^2\right]}$$

which is the baseline you should subtract from the gradient of the i^{th} parameter θ_i to result in the largest variance reduction. This expression is just the expected return but it is weighted by the magnitude of the gradient, this again 1–2 lines of code.

17.5 An alternative expression for the policy gradient

Let us define an important quantity that helps us think of RL algorithms.

Definition 11 (Discounted state visitation frequency). Given a stochastic controller $u_\theta(\cdot \mid x)$ the discounted state visitation frequency for a discrete-time dynamical system is given by

$$d^\theta(x) = \sum_{k=0}^{\infty} \gamma^k \, \mathbb{P}(x_k = x \mid x_0, u_k \sim u_\theta(\cdot \mid x_k)).$$

The distribution $d^\theta(x)$ is the probability of visiting a state x computed over all trajectories of the system that start at the initial state x_0. If $\gamma = 1$, this is the steady-state distribution of the Markov chain underlying the Markov Decision Process where at each step the MDP choses the control $u_k \sim u_\theta(x_k)$. The fact that we have defined the discounted distribution is a technicality; this version is seen in the policy gradient expression. You will also notice that $d^\theta(x)$ is not a normalized distribution. The normalization constant is difficult to characterize both theoretically and empirically and we will not worry about it here; RL algorithms do not require it.

The policy gradient that we saw before can be written in terms of the q -factor as follows.

$$\widehat{\nabla} J(\theta) = \mathop{\mathbb{E}}_{x \sim d^\theta} \; \mathop{\mathbb{E}}_{u \sim u_\theta(\cdot \mid x)} \left[q^\theta(x,u) \, \nabla_\theta \log u_\theta(u \mid x) \right]. \tag{17.6}$$

The function q^θ is what we have seen before as the cost-to-go in Module 2:

$$q^\theta(x,u) = \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \left[R(\tau) \mid x_0 = x, u_0 = u \right]; \tag{17.7}$$

it is the infinite-horizon discounted cummulative reward (return) if the system starts at state x and takes the control u in the first step and runs the controller $u_\theta(\cdot \mid x)$ for all steps thereafter. We will make the dependence of q^θ on the parameters θ of the controller explicit. This function is also often called the action-value function.

The derivation is easy although tedious, you can find it in the Appendix of the paper "Policy gradient methods for reinforcement learning with function approximation" at https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf.

Remark 12. Compare the above formula for the policy gradient with the one we had before in (17.4)

$$\begin{aligned} \widehat{\nabla} J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \left[R(\tau) \, \nabla \log p(\tau;\, \theta) \right] \\ &= \mathop{\mathbb{E}}_{\tau \sim p(\tau;\theta)} \left[\left(\sum_{k=0}^{T} \gamma^k r(x_k, u_k) \right) \left(\sum_{k=0}^{T} \nabla \log u_\theta(u_k \mid x_k) \right) \right]. \end{aligned}$$

This is an expectation over trajectories; (17.6) is an expectation over states x sampled from the discounted state visitation frequency. The control u_k in (17.6) is sampled from the stochastic controller at each time-step k and so is the one in the above expression except that it is implicit as shown in the equality via (17.4). The most important distinction is that (17.6) involves the expectation of the action-value function q^θ weighted by the gradient of the log-likelihood of picking

each control action. There are numerous hacky ways of deriving (17.6) from (17.4) but remember that they are fundamentally different expressions of the same policy gradient.

This expression allows understanding of a number of properties of reinforcement learning.

1. While the algorithm collects the data, states that are unlikely under the distribution d^θ contribute little to (17.6). In other words, the policy gradient is insensitive to such states. The policy update will not consider these unlikely states.

2. The opposite happens for states which are very likely. For two controls u_1, u_2 at the same state x, the policy increases the log-likelihood of taking the controls weighted by their values $q^\theta(x, u_1)$ and $q^\theta(x, u_2)$. This is the "definition" of reinforcement learning. In the expression (17.4) the gradient was increasing the likelihood of trajectories with high returns, here it deals with states and controls.

17.5.1 Implementing the new expression

Suppose we have a stochastic control that is a Gaussian

$$u_\theta(u \mid x) = \frac{1}{(2\pi\sigma^2)^{p/2}} \, e^{-\frac{\|u-\theta^\top x\|^2}{2\sigma^2}}$$

where $\theta \in \mathbb{R}^{d\times p}$ and $u \in \mathbb{R}^p$; the variance σ can be chosen by the user. We can easily compute the $\log u_\theta(u \mid x)$ in (17.6). How should one compute $q^\theta(x, u)$ in (17.7)? We can again estimate it using sample trajectories from the system; each of these trajectories would have to start from a state x and the control at the first step would be u, with the controller u_θ being used thereafter. Note that we have one such trajectory, namely the remainder of the trajectory where we encountered (x, u) while sampling trajectories for the policy gradient in (17.6). In practice, we do not sample trajectories a second time, we simply take this trajectory, let us call it $\tau_{x,u}$ and set

$$q^\theta(x, u) = \sum_{k=0}^{T} \gamma^k r(x_k, u_k)$$

where $(x_0, u_0) = (x, u)$ and the summation is evaluated for (x_k, u_k) that lie on the trajectory $\tau_{x,u}$. Effectively, we are evaluating (17.7) using one sample trajectory, a highly erroneous estimate of q^θ.

17.5.2 Actor-Critic methods

We can of course do better. We know the q-function satisfies the Bellman equation (dynamic programming equation) from Module 2. This means

$$q^\theta(x, u) = r(x, u) + \gamma \mathop{\mathbb{E}}_{x'\sim P(\cdot \mid x,u)} \left[q^\theta(x', u_\theta(x'))\right]. \tag{17.8}$$

We do not know a model for the system so we cannot evaluate the expectation over $x' \sim \mathbb{P}(\cdot \mid x, u)$. But we do have trajectories τ^i from the system that we drew to evaluate the policy gradient. Let's

say (x_k^i, u_k^i) lie on τ^i at time-step k. We can then estimate the expectation over $\mathbb{P}(\cdot \mid x_k^i, u_k^i)$ using simply x_{k+1}^i. We can write

$$q^\theta(x_k^i, u_k^i) \approx r(x_k^i, u_k^i) + \gamma\, q^\theta(x_{k+1}^i, u_{k+1}^i) \quad \text{for all } i \leq n, k \leq T.$$

This is a nice constraint on the values of the q-function. If this were a discrete-state, discrete-control MDP, it is a set of linear equations for the q-values. If we are dealing with a continuous state/control-space, we can think of parameterizing the q-function using parameters φ

$$q_\varphi^\theta(x, u) : X \times U \to \mathbb{R}.$$

The parameterization is similar to the parameterization of the controller, e.g., just like

$$u_\theta(x) = \theta^\top x$$

we can think of a linear q-function of the form

$$q_\varphi^\theta(x, u) = \varphi^\top \begin{bmatrix} x \\ u \end{bmatrix}, \quad \varphi \in \mathbb{R}^{p+d}$$

which is a linear q-function in the states and controls. You can also think of using something like

$$q_\varphi^\theta(x, u) = \begin{bmatrix} 1 & x & u \end{bmatrix} \varphi \begin{bmatrix} 1 \\ x \\ u \end{bmatrix} \quad \varphi \in \mathbb{R}^{(p+d+1)\times(p+d+1)}.$$

which is quadratic in the states and controls. You can now "fit" the parameters of the q-function by solving the problem

$$\widehat{\varphi} = \operatorname*{argmin}_\varphi \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \|q_\varphi^\theta(x_k^i, u_k^i) - r(x_k^i, u_k^i) + \gamma\, q_\varphi^\theta(x_{k+1}^i, u_{k+1}^i)\|^2. \tag{17.9}$$

If the q-function is linear in $[x, u]$ this is a least squares problem, if it is quadratic the problem is a quadratic optimization problem which can also be solved efficiently. Such a q-function is called the "critic", it evaluates the controller u_θ which is called the "actor". This particular version of the policy gradient where one fits the parameters of both the controller and the q-function are called Actor-Critic methods.

Remark 13. We will be pedantic and always write the q-function as q_φ^θ. The superscript θ denotes that this is the q-function corresponding to the θ-parameterized controller u_θ. The subscript denotes that the q-function is parameterized by parameters φ.

Question 14. Remember that we draw n trajectories from the system before each update to θ and φ using the policy gradient. The mathematics suggests that we should fit the q-function afresh before each update. Of the the q-function that evaluates u_{θ_k} given by q^{θ_k} is different from the one that evaluates $u_{\theta_{k+1}}$ given by $q^{\theta_{k+1}}$. If we are minimizing (17.9) using stochastic gradient descent, we really have an approximate solution for $\widehat{\varphi}_k$. We can solve for $\widehat{\varphi}_{k+1}$ using $\widehat{\varphi}_k$ as an initialization instead of solving from scratch.

17.5.3 Advantage function

The new expression for the policy gradient also has a large variance; this should be no surprise it is equal to the old expression. We can perform variance reduction on this using the value function.

Question 15. Show that any function that only depends on the state x can be used as a baseline in the policy gradient. This is known as reward shaping.

Question 16. The above question suggests a powerful baseline that we can use for the policy gradient expression in (17.6). What is a function that we often saw in dynamic programming that we expect to be correlated with the q-function and it is also only a function of the state x?

The answer is given by the value function

$$v^\theta(x) = \mathop{\mathbb{E}}_{\tau\sim p(\tau \mid \theta)} \left[R(\tau) \mid x_0 = x, u_k \sim u_\theta(x_k)\right].$$

The value function again satisfies the dynamic programming principle/Bellman equation

$$v^\theta(x) = \mathop{\mathbb{E}}_{u\sim u_\theta(x)} \left[r(x,u) + \gamma \mathop{\mathbb{E}}_{x'\sim \mathbb{P}(\cdot \mid x,u)} \left[v^\theta(x')\right]\right].$$

We again parameterize the value function using parameters

$$v^\theta_\psi(x) : X \to \mathbb{R}$$

and fit it using the data like (17.9) to get

$$\widehat{\psi} = \operatorname*{argmin}_\psi \frac{1}{n(T+1)} \sum_{i=1}^n \sum_{k=0}^T \|v^\theta_\psi(x^i_k) - r(x^i_k, u^i_k) + \gamma\, v^\theta_\psi(x^i_{k+1})\|^2. \tag{17.10}$$

Using this baseline will modify the policy gradient to be

$$\widehat{\nabla} J(\theta) = \mathop{\mathbb{E}}_{x\sim d^\theta} \mathop{\mathbb{E}}_{u\sim u_\theta(\cdot \mid x)} \left[\Big(\underbrace{q^\theta_\varphi(x,u) - v^\theta_\psi(x)}_{a^\theta_{\varphi,\psi}(x,u)} \Big) \nabla_\theta \log u_\theta(u \mid x) \right]. \tag{17.11}$$

where each of the functions q^θ_φ and v^θ_ψ are themselves fitted using (17.9) and (17.10) respectively. The difference

$$\begin{aligned} a^\theta_{\varphi,\psi}(x,u) &= q^\theta_\varphi(x,u) - v^\theta_\psi(x) \\ &= q^\theta_\varphi(x,u) - \mathop{\mathbb{E}}_{u\sim u_\theta(x)} \left[q^\theta_\varphi(x,u)\right] \end{aligned} \tag{17.12}$$

is called the advantage function. It measures how much better the particular control u is for a state x as compared to the average return of controls sampled from the controller at that state. The form (17.11) is the most commonly implemented form in research papers.

Question 17. The advantage function is very useful while doing theoretical work on RL algorithms. But it is also extremely useful in practice. It imposes a constraint upon our estimate q^θ_φ and the

estimate v_ψ^θ. If we are not solving (17.9) and (17.10) to completion, we may benefit by imposing this constraint of the advantage function. Can you think of a way?

17.6 Discussion

This brings to an end the discussion of policy gradients. They are a notoriously complicated suite of algorithms to implement; you will see some of this complexity when you implement the controller for something as simple as the simple pendulum. The key challenges with implementing policy gradients come from the following.

1. Need lots of data, each parameter update requires fresh data from the systems. Typical problems may need a million trajectories, most robots would break before one gets this much data from them if one implements these algorithms naively.
2. The gradient is very inaccurate (example of grabbing a cup of coffee) so may need lots of samples even in one iteration.
3. The log-likelihood ratio trick has a high variance due to $u_\theta(\cdot \mid x)$ being in the denominator of the expression, so we need variance reduction techniques.
4. Fitting the q-function and the value function in the expression is not easy, each parameter update of the policy ideally requires you to solve the entire problems (17.9) and (17.10). In practice, we only perform a few steps of SGD to solve the two problems and reuse the solution of k^{th} controller update as an initialization of the $k+1^{\text{th}}$ update. This is known as "warm start" in the optimization literature and reduces the complexity of fitting the q/value-functions from scratch each time.
5. The q/value-function fitted in iteration k may be poor estimates of the q/value at iteration $k+1$ for the new policy $u_{\theta_{k+1}}$. If the controller parameters change quickly, θ_{k+1} is very different from θ_k and so are $q^{\theta_{k+1}}$ and $v^{\theta_{k+1}}$. There is a very fine balance between training quickly and retaining the efficiency of warm start; and tuning this in practice is quite difficult. A large number of policy gradient algorithms like TRPO (https://arxiv.org/abs/1502.05477) and PPO (https://arxiv.org/abs/1707.06347) try to alleviate this with varying degrees of success.

Lecture 18

Q-Learning

Reading

- Sutton & Barto, Chapter 6, 11

The previous chapter looked at on-policy methods, these are methods where the current controller u_{θ_k} is used to draw fresh data from the system and used to complete the update to paramters θ_k. The key inefficiency in on-policy methods is that this data is thrown away in the next iteration. We need to draw a fresh set of trajectories from the system for $u_{\theta_{k+1}}$. This lecture will discuss off-policy methods which are a way to reuse past data. These methods require much fewer data than on-policy methods (10–100$\times$ less).

18.1 Tabular Q-Learning

Recap of tabular version of Bellman iteration

18.1.1 How to explore?

Fact: Tabular Q-Learning converges to the **optimal** q-function even if the controller gathering data is sub-optimal. This result relies on visiting all states in the MDP infinitely often and having a learning rate that does not decay to zero too quickly, i.e., we are making non-vanishing updates to the q-function always.

18.2 Function approximation

Tabular methods are really nice but they do not scale. The grid-world in the midterm exam had 64 states, a typical game of Tetris has about 10^{60} states (the number of atoms in the known universe is

about 10^{80}). The number of different states in a typical Atari game is more than 10^{300}. We would therefore like to learn a parametrized q-function q_φ. This q-function may not be exact for all states and controls (x, u) by so long as the controller computing using this q-function is not too different from the optimal controller, we can use it.

Remark 1. We can also use function approximation for a tabular MDP from the previous section.

18.2.1 Fitted Q-Iteration

18.3 Deep Q-Networks

18.4 Q-Learning for continuous control-spaces

Lecture 19

Q-Learning

Reading

- Sutton & Barto, Chapter 6, 11
- Human-level control through deep reinforcement learning, https://www.nature.com/articles/nature14236.

The previous chapter looked at on-policy methods, these are methods where the current controller u_{θ_k} is used to draw fresh data from the system and used to complete the update to paramters θ_k. The key inefficiency in on-policy methods is that this data is thrown away in the next iteration. We need to draw a fresh set of trajectories from the system for $u_{\theta_{k+1}}$. This lecture will discuss off-policy methods which are a way to reuse past data. These methods require much fewer data than on-policy methods (10–100$\times$ less).

19.1 Tabular Q-Learning

Recall the tabular version of Bellman iteration for discrete (and finite) state and control spaces. The q-function estimate at the k^{th} iteration is given by

$$\begin{aligned} q_{k+1}(x,u) &= \sum_{x' \in X} \mathbb{P}(x' \mid x,u) \ \left(r(x,u) + \gamma \max_{u'} q_k(x',u') \right) \\ &= \mathop{\mathbb{E}}_{x' \sim \mathbb{P}(\cdot \mid x,u)} \left[r(x,u) + \gamma \max_{u'} q_k(x',u') \right]. \end{aligned}$$

In the simplest possible instantiation of q-learning, the expectation above is replaced by samples draw from the environment. Imagine a grid-world in which the robot roams around using some arbitary controller $u_e(\cdot \mid x)$. We will call this the "exploration controller". We maintain the value $q_k(x,u)$ for all states $x \in X$ and controls $u \in U$ and update these values to get q_{k+1} after each step of the robot. We know that any q-function that satisfies the above equation is the optimal q-function; we would

therefore like our q-function to satisfy

$$q(x_k, u_k) \approx r(x_k, u_e(x_k)) + \gamma \max_{u'} \; q(x_{k+1}, u').$$

Let us imagine the robot travels for n trajectories each of length T time-steps. We can now solve for q by minimizing the objective

$$\min_q \underbrace{\frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \| q(x_k^i, u_e(x_k^i)) - r(x_k^i, u_e(x_k^i)) - \gamma \max_{u' \in U} \; q(x_{k+1}^i, u') \|_2^2}_{:= \ell(q)}. \tag{19.1}$$

on the data collected by the robot. Notice a few important things about this expression.

1. It involves a maximization over $u' \in U$, it is not simply $u_e(x_{k+1}^i)$ which is the control the robot took in the next step.

2. How would this equation change if state the robot reached x_{k+1}^i was a terminal state. What is the value of the terminal state?

3. Every entry $q(x, u)$ for all $x \in U$ and $u \in U$ is a variable of this objective. We can solve for it iteratively as
$$q(x, u) \; \leftarrow \; q(x, u) - \eta \, \nabla_{q(x,u)} \, \ell(q). \tag{19.2}$$
Note that only (x, u) in the data collected by the robot
$$D = \left\{ (x_k^i, u_k^i)_{k=0,\ldots,T} \right\}_{i=1,\ldots,n}$$
get a non-zero gradient; the q-function $q(x, u)$ at all others remains unchanged.

4. We can also interleave the updates to the q-function with the states visited by the robot, i.e., if the robot took a transition (x, u, x') and obtained a reward r, we update
$$q(x, u) \; \leftarrow \; (1 - \eta) \, q(x, u) + \eta \left(r(x, u, x') + \gamma (1 - \mathbf{1}_{\{x' \text{ is terminal}\}}) \max_{u'} q(x', u') \right)$$
You will notice that expression is simply the gradient of the objective in (19.2) at $q(x, u)$.

Question 1. What should you initialize $q(x, u)$ with for tabular q-learning?

Remark 2 (What is the controller). After we perform many updates to the q-function and are done with the training, we can use the controller

$$u(x) = \operatorname*{argmax}_{u'} q(x, u')$$

at test time. This is the optimal controller if $q(x, u)$ is the optimal q-function.

19.1.1 How to explore?

Fact: Tabular Q-Learning converges to the **optimal** q-function even if the controller gathering data is sub-optimal. This result relies on visiting all states in the MDP infinitely often and having a learning rate that does not decay to zero too quickly, i.e., we are making non-vanishing updates to the q-function always. It is important to understand this fact because it is powerful. Effectively, you can learn the optimal q-function in spite of having an arbitrary stochastic controller that gathers data.

Question 3. How should we design the exploratory controller $u_e(\cdot \mid x)$?

Remark 4 (ϵ-greedy exploration). We can have the robot use the q-function itself to gather data. A nice deterministic policy to use is the greedy policy

$$u_e(x) := \operatorname*{argmax}_{u'} q(x, u').$$

This controller has a problem: it repeatedly keeps taking the controls we have taken in the past (why?) and we do not get any new information from the system. The ϵ-greedy is a popular heuristic to explore the control space, it sets the stochastic exploratory controller to be

$$u_e(u \mid x) := \begin{cases} \operatorname{argmax}_{u'} q(x, u') & \text{with probability } 1 - \epsilon \\ \text{uniform}(U) & \text{with probability } \epsilon. \end{cases}$$

This controller uses the greedy policy of the current q-function most of the time and takes a random action with probability ϵ.

Question 5. Why should we use ϵ-greedy exploration? Why not simply set $\epsilon = 1$? This leads to the idea of a replay buffer.

Remark 6. You can also think of tabular q-learning as happening in two stages. In the first stage, the robot gathers a large amount of data

$$D = \left\{ (x_k^i, u_k^i)_{k=0,\dots,T} \right\}_{i=1,\dots,n}.$$

It then fits the model for the system, i.e., it learns the MDP, and uses value iteration to find the q-function for this MDP. Given this data and a discrete state/control space, how can we learn the MDP? Learning an MDP means that for all states x, controls u and next states x' we would like to estimate the probability

$$\mathbb{P}(x' \mid x, u).$$

This is easy to do using the empirical frequency counts. We estimate

$$\mathbb{P}(x' \mid x, u) \approx \frac{1}{N} \sum_{i=0}^{N} \mathbf{1}_{\{x' \text{ was reached from } x \text{ using } u \text{ in the dataset}\}}$$

where N is the total number of times in the dataset where the robot took control u at state x. Once we have the transition probability, we can simply perform value/policy iteration on this MDP to get the optimal value function

$$q_{k+1}(x, u) = \mathop{\mathbb{E}}_{x' \sim \mathbb{P}(\cdot \mid x, u)} \left[r(x, u) + \gamma \max_{u'} q_k(x', u') \right]$$

The success of this two-stage approach depends upon how accurate the estimate of $\mathbb{P}(x' \mid x, u)$ is. This in turn depends on how much the robot explored the domain and the size of the dataset it collected, both of these need to be large.

You can therefore think of q-learning as interleaving the two stages, it learns the dynamics/model and the q-function for that dynamics/ simultaneously, but does not really contain a representation of the dynamics/ in the algorithm.

19.2 Function approximation (Deep Q Networks)

Tabular methods are really nice but they do not scale. The grid-world in the midterm exam had 64 states, a typical game of Tetris has about 10^{60} states (the number of atoms in the known universe is about 10^{80}). The number of different states in a typical Atari game is more than 10^{300}. We would therefore like to learn a parametrized q-function q_φ. This q-function may not be exact for all states and controls (x, u) but so long as the controller computed using this q-function is not too different from the optimal controller, we can use it.

We use the same idea of parameterizing the q-function using parameters φ from the previous lecture

$$q_\varphi(x, u) : X \times U \to \mathbb{R}.$$

Let us for the moment consider a discrete state/control space. We again have an exploratory policy $u_e(\cdot \mid x)$ that gathers data from the environment. Fitting the q-function now involves enforcing the Bellman equation on the collected data

$$q_\varphi(x, u) = r(x, u, x') + \gamma \max_{u'} q_\varphi(x', u') \quad \text{for all } (x, u, x', r) \in D.$$

While there are many ways of doing so, one popular method is to solve

$$\operatorname*{argmin}_{\varphi} \mathop{\mathbb{E}}_{(x,u,x',r)\sim D} \left[\left(q_\varphi(x, u) - \underbrace{r(x, u, x') - \gamma(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}) \max_{u'} q_\varphi(x', u')}_{\text{target } (x';\, \varphi)} \right)^2 \right]. \tag{19.3}$$

We can again update the parameters φ using gradient descent or stochastic gradient descent as

$$\varphi_{k+1} = \varphi_k - \eta \nabla_\varphi \Big(q_{\varphi_k}(x, u) - \text{target}(x';\, \varphi_k) \Big)^2 \tag{19.4}$$

A few important points:

1. **Off-policy learning still interacts with the environment.** The dataset D is known as the replay buffer. Recall that the current q-function being trained is typically used to obtain an ϵ-greedy exploration policy $u_e(\cdot \mid x)$. The transitions (x, u, x', r) that are collected by the robot are stored in the replay buffer whose size increases as the training progresses. Off-policy does not mean offline, the robot still interacts with the environment. We will discuss this a bit deeper in the lecture.

2. **Setting a good value for exploration is critical.** Towards the beginning of training, we want a large value for ϵ because the estimates of the q-function are erroneous. As training progresses, we want to reduce ϵ because presumably we have a few good control trajectories to maximize the reward and focus on improving those trajectories.

3. **Prioritized experience replay** Sample transitions that have high Bellman error instead of sampling uinformly from the replay buffer in SGD.

4. **The above version of q-learning has a very big problem.** The Bellman equation guarantees that if we find a q-function that satisfies the Bellman equation

$$q(x,u) = \mathop{\mathbb{E}}_{x' \sim \mathbb{P}(\cdot \mid x,u)} \Big[r(x,u) + \gamma \max_{u'} q(x',u') \Big]$$

at all (x,u) this is the optimal q-function. This however says nothing about

(i) how to satisfy these constraints; we simply used quadratic loss between the left-hand and right-hand sides; there may be better ways of finding this fixed point, e.g., the Huber loss

$$\text{huber}_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta \\ \delta\left(|a| - \frac{1}{2}\delta\right) & \text{otherwise.} \end{cases}$$

is a good candidate to replace the quadratic loss.

(ii) what happens when we use a parametric q-function. The reason we use parameters is because the state/control space is too large to visit/represent everything and we hope that the function approximator gives reasonable values at those (x,u). The Bellman equation need not be satisfied for those (x,u). Finding optimal q-function using function approximation is a very (very) tricky business and you can easily construct simply tabular settings where q-learning breaks down.

5. **Delayed target.** In practice, this problem shows up as q_φ fitting the data in (19.3) almost perfectly but being completely wrong, i.e.,

$$q_\varphi(x,u) \approx \text{target}(x';\ \varphi) \quad \text{but}$$
$$q_\varphi(x,u) \neq \mathop{\mathbb{E}}_{\text{trajectories } \tau} \Big[R(\tau) \mid x_0 = x, u_0 = u, u_k = \operatorname*{argmax}_{u'} q(x,u') \Big]$$

You will often see inordinately large values for all $q_\varphi(x,u)$ when this hppens.

The above problem is mitigated (not really eliminated) by modifying the updates in (19.5) as

$$\varphi_{k+1} = \varphi_k - \eta\, \nabla_\varphi \Big(q_{\varphi_k}(x,u) - \text{target}(x';\ \varphi_{k'}) \Big)^2 \tag{19.5}$$

where k' is an iterate much older than k, say $k' = k - 100$. This idea is known as delayed target. The rationale of doing so is very interesting: even if the estimates of q_{φ_k} may be wrong, the delayed target forces q_{φ_k} to underestimate the return at state (x,u) and prevents q_{φ_k} from blowing up.

6. **There is no correction mechanism in Q-learning if the value estimate is wrong**. Compare the execution of the policy gradient with what is happening in Q-learning.

7. **Double Q learning.** The max operator in standard Q-learning or the one with delayed target uses the same q-function both to select and to evaluate a control, i.e., the target has a term

$$\max_{u'} q_{\varphi_{k'}}(x', u').$$

This makes it more likely to select controls with overestimated q-values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation. This is achieved by doing

$$\text{target}(x';\ \varphi_{k'}) := r(x, u, x') + \gamma(1 - \mathbf{1}_{\{x' \text{ is terminal}\}})\ q_{\varphi_{k'}}(x', \operatorname*{argmax}_{u'} q_{\varphi_k}(x', u')).$$

Remark 7 (Fitted Q-Iteration).

19.3 Q-Learning for continuous control-spaces

Lecture 22

Continuous Q-Learning, Inverse Reinforcement Learning

Reading

- Deterministic Policy Gradient Algorithms, http://proceedings.mlr.press/v32/silver14.html
- Addressing Function Approximation Error in Actor-Critic Methods https://arxiv.org/abs/1802.09477
- An Application of Reinforcement Learning to Aerobatic Helicopter Flight, https://papers.nips.cc/paper/3151-an-application-of-reinforcement-learning-to-aerobatic-helicopter-flight
- Algorithms for Inverse Reinforcement Learning https://ai.stanford.edu/ ang/papers/icml00-irl.pdf
- Maximum Entropy Inverse Reinforcement Learning, https://www.aaai.org/Papers/AAAI/2008/AAAI08-227.pdf

---

22.1 Q-Learning for continuous spaces

22.1.1 Deterministic policy gradient

22.1.2 Twin-Delayed DDPG

22.2 Inverse RL

22.2.1 LQR

22.2.2

Lecture 23

Inverse and Model-based Reinforcement Learning

Reading

- An Application of Reinforcement Learning to Aerobatic Helicopter Flight, https://papers.nips.cc/paper/3151-an-application-of-reinforcement-learning-to-aerobatic-helicopter-flight
- Algorithms for Inverse Reinforcement Learning https://ai.stanford.edu/ãng/papers/icml00-irl.pdf
- Apprenticeship Learning via Inverse Reinforcement Learning, https://ai.stanford.edu/ ang/papers/icml04-apprentice.pdf
- Maximum Entropy Inverse Reinforcement Learning, https://www.aaai.org/Papers/AAAI/2008/AAAI08-227.pdf
- PILCO: A Model-Based and Data-Efficient Approach to Policy Search, http://mlg.eng.cam.ac.uk/pub/pdf/DeiRas11.pdf
- Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images https://arxiv.org/abs/1506.07365
- Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models https://arxiv.org/abs/1805.12114

23.1 Fitting LQR dynamics and rewards

23.2 IRL with a known model

23.3 IRL from sampled trajectories

23.4 Maximum Entropy IRL

23.5 Model-based RL

23.5.1 Fitting neural networks for the dynamics

23.5.2 PILCO

See a preliminary but great tutorial on Gaussian Processes at https://distill.pub/2019/visual-exploration-gaussian-processes

This is a complicated algorithm to implement but you can see the source code by the original authors at https://mloss.org/software/view/508

23.5.3 Ensemble methods in practice

Lecture 25

Meta-Learning

Reading

- Learning To Learn: Introduction (1996), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.3140
- Learning Many Related Tasks at the Same Time with Backpropagation https://papers.nips.cc/paper/959-learning-many-related-tasks-at-the-same-time-with-backpropagation
- Prototypical Networks for Few-shot Learning http://papers.nips.cc/paper/6996-prototypical-networks-for-few-shot-learning
- Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks https://arxiv.org/abs/1703.03400
- A Baseline for Few-Shot Image Classification https://arxiv.org/abs/1909.02729
- Meta-Q-Learning https://arxiv.org/abs/1910.00125

The human visual system is proof that we do not need lots of images to learn to identify objects or lots of experiences to learn about concepts. Consider the mushrooms shown in the image below.

The one on the left is called *Muscaria* and you'd be able to identify the bright spots on this mushroom very easily after looking at one image. The differences between an edible one in the center (*Russala*)

and the one on the right (*Phalloides*) may sometimes be subtle but a few samples from each of them are enough for humans to learn this important distinction.

The world of control comes with more dramatic examples of this phenomenon. You touched a hot stove once as a child and have forever learnt not to do it. You learnt to ride a bike as a child and only need a few minutes on a completely new bike to be able to ride it these days. At the same time, you could start learning to juggle today and will be able to juggle 3 objects with a couple of days of practice.

The hallmark of human perception and control is the ability to generalize. This generalization comes in two forms. The first is the ability to do a task better if you see more samples from the same task; this is what machine learning calls generalization. The second is the ability to mix-and-match concepts from previously seen tasks to do well on new tasks; doing well means obtaining a lower error/higher reward as well as learning the new task quickly with few days. This second kind is the subject of what is called "learning to learn" or meta-learning.

$$\text{Standard machine learning} \Rightarrow \text{generalization across samples}$$

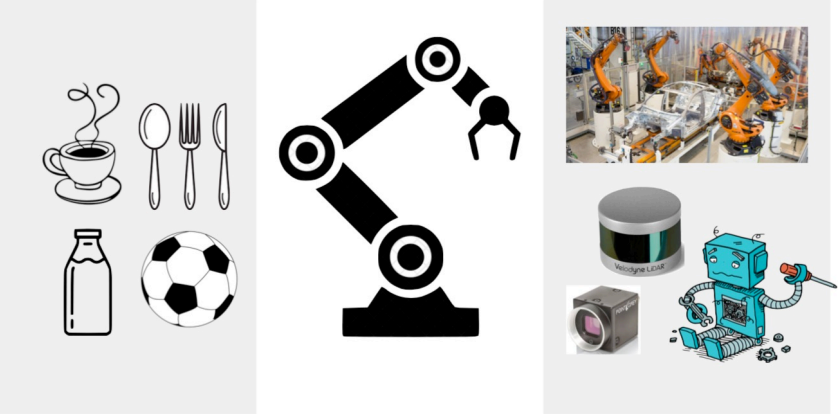
$$\text{Meta-Learning} \Rightarrow \text{generalization across tasks}$$

Remark 1 (What is a task?). If we are going to formalize meta-learning, we better define what a task is. This is harder than it sounds. Say we are doing image classification, classifying cats vs. dogs could be considered Task 1; Task 2 could be classifying apples vs. oranges. It is reasonable to expect that learning low-level features such as texture, colors, shapes etc. while learning Task 1 could help us to do well on Task 2.

This is not always the case, two tasks can also fight each other. Say, you design a system to classify ethnicities of people using two kinds of features. Task 1 uses the shape of the nose to classify Caucasians (long nose) vs. Africans (wide nose). Task 2 uses the kind of hair to classify Caucasians (straight hair) vs. Africans (curly hair). An image of a Caucasian person with curly hair clearly results in two tasks fighting each other.

The difficulty in meta-learning begins with defining what a task is.

Remark 2 (Tasks in robotic systems). While understanding what a task is may seem cumbersome but doable for image classification, it is even harder for robotics systems.



We can think of two different kinds of tasks.

1. The first, on the left, is picking up different objects like a soccer ball, a cup of coffee, a bottle of water etc. using a robot manipulator. We may wish to learn how to pick up a soccer ball quickly given data about how to pick up the bottle.

2. The second kind of tasks is shown on the right. You can imagine that after building/training a model for the robot in your lab you want to install it in a factory. The factory robot might have 6 degrees-of-freedom whereas the one in your lab had only 5; your policy better adapt to this slightly different state-space. An autonomous car typically has about 10 cameras and 5 LIDARs, any learning system on the car better adapt itself to handle the situation when one of these sensors breaks down. The gears on a robot manipulator will degrade over the course of its life, our policies should adapt to this degrading robot.

Almost all the current meta-learning/meta-reinforcement learning literature focuses on developing methods to do the first set of tasks. The second suite of tasks are however more important in practice. In the remainder of this chapter, we will discuss two canonical algorithms to tackle these two kinds of tasks.

Question 3 (Meta-learning vs. multi-task learning). We have talked about adaptation as the way to handle new tasks in the previous remark. Consider the following situation: in standard machine learning, we know that larger the size of the training data we collect better the performance on the test data; a large number of images help capture lots of variability in the data, e.g., dogs of different shapes, sizes and colors. You can imagine then that in order to do well on lots of different tasks, i.e., meta-learn, we should simply collect data from lots of different tasks. This is an idea known as multi-task learning which is the second assigned reading today.

Can you think as to why mere multi-task learning may not work well for meta-learning?

25.1 Problem formulation for image classification

The image classification formulation thinks of each class/category as a "task".

Consider a supervised learning problem with a dataset $D = \left\{(x^i, y^i)\right\}_{i=1,\ldots,N_b}$. The labels $y^i \in \{1, \ldots, K\}$ for some large K and there are $\frac{N_b}{K}$ samples in the dataset for each class. Think of this as a large dataset of cars, cats, dogs, airplanes etc. all objects that are very frequent in nature and for which we can get lots of images. This training set is called the meta-training set with K "base tasks".

Say, we were simply interested in obtaining a machine learning model to classify this data. Let us denote the parameters of this model by θ. If this model predicts the probability of the input x belonging to each of these known classes we can think of maximizing the log-likelihood of the data under the model

$$\widehat{\theta} = \underset{\theta}{\text{argmax}} \ \frac{1}{N_b} \sum_{i=1}^{N_b} \log p_\theta(y^i \mid x^i). \tag{25.1}$$

This is the standard multi-class image classification setup. Since we like to think of one task as one

category, this is also the multi-task learning setup. The model

$$p_{\widehat{\theta}}(\cdot \mid x)$$

after fitting on the training data will be good at classifying some new input image x as to whether it belongs to one of the K training classes. Note that we have written the model as providing the probability distribution $p_{\widehat{\theta}}(\cdot \mid x)$ as the output, one real-valued scalar per candidate class $\{1, \ldots, K\}$.

Question 4. Say, we are interested in classifying images from classes that are different than those in the training set. The model has only K outputs, effectively the universe is partitioned into K categories as far as the model is concerned and it does not know about any other classes. How should one formalize the problem of meta-learning then?

We will consider the following setup. We are given a "few-shot dataset" that has w new classes and s labeled samples per class, a total of $N_n = ws$ new samples

$$D^n = \left\{(x^i, y^i)\right\}_{i=N_b+1,\ldots,N_b+N_n}; \quad y^i \in \{K+1, \ldots, K+w\}.$$

The words few-shot simply mean that s is small, in particular we are given much fewer images per class than the meta-training dataset,

$$\frac{N_n}{w} = s \ll \frac{N_b}{K}.$$

This models the situation where the model is forced to classify images from rare classes, e.g., the three kinds of strawberries grown on a farm in California after being trained on data of cars/cats/dogs/planes etc.

25.1.1 Fine-tuning

We would like to adapt the parameters $\widehat{\theta}$ using this labeled few-shot data. Here is one solution, we simply train the model again on the new data. This looks like solving another optimization problem of the form

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \frac{1}{N_n} \sum_{i=N_b+1}^{N_b+N_n} \log p_\theta(y^i \mid x^i) - \frac{1}{2\lambda}\|\theta - \widehat{\theta}\|_2^2. \tag{25.2}$$

The new parameters θ^* can potentially do well on the new classes even if s is low because the training is initialized using the parameters $\widehat{\theta}$. We write down this initialization using the second term

$$\frac{1}{2\lambda}\|\theta - \widehat{\theta}\|_2^2$$

which keeps the parameters being optimized θ closed to their initialization using a quadratic spring-force controlled by the parameter λ. We can expect the new model p_{θ^*} to perform well on the new classes if the initialization $\widehat{\theta}$ was good, i.e., if the new tasks and the base tasks were close to each other. This method is called fine-tuning, it is the easiest trick to implement to handle new classes.

Question 5. Think of a multi-layer neural network $\widehat{\theta}$ that has K outputs. The new network should now produce w outputs, how should we modify this network?

25.1.2 Prototypical networks

The cross-entropy objective used in (25.1) to train the model $p_{\widehat{\theta}}$ simply maximizes the log-likelihood of the labels given the data. It is reasonable to think that since the base classes are not going to show up as the few-shot classes, we should not be fitting to this objective. The idea behind a propotypical loss is to train the model to be a good discriminator among w classes.

Let us imagine the features of the model, e.g., the activations of the last layer in the neural network,

$$z = \varphi_\theta(x)$$

for a particular image x. Note that the features z depend on the parameters θ as well. During standard cross-entropy training, there is a linear layer on top of these features and one has

$$p_\theta(y \mid x) = \frac{e^{w_y^\top z}}{\sum_{y'} e^{w_{y'}^\top z}}$$

where $w_y \in \mathbb{R}^{\dim(z)}$. This is called the softmax operator and the vectors w are the weights of the last layer of the network; when we wrote (25.1) we implicitly folded those parameters into the notation θ. Prototypical networks train the model to be a discriminator.

1. Each mini-batch during training time consists of a few-shot task created out of the meta-training set by sub-sampling.

$$\begin{aligned} D^{\text{episode}} &= D^{\text{support}} \cup D^{\text{query}} \\ &= \left\{(x^i, y^i);\ y^i \in \{1, \dots, K\}\right\}_{i \in \{1,\dots,N_b\}}; \\ &\quad \cup \left\{(x^i, y^i);\ y^i \in \{1, \dots, K\}\right\}_{i \in \{1,\dots,N_b\}}. \end{aligned}$$

   with $|D^{\text{support}}| = ws$ and $|D^{\text{query}}| = wq$. This is called an "episode" in this literature. Each episode comes with some more data from the same classes called the "query-shot" in this literature. The query-shot is akin to the data from the new classes that the model is forced to predict during adaptation time. Let us have q query-shot per class in each episode.

2. We know the labels of the $N_n = ws$ labeled data and can compute the prototypes

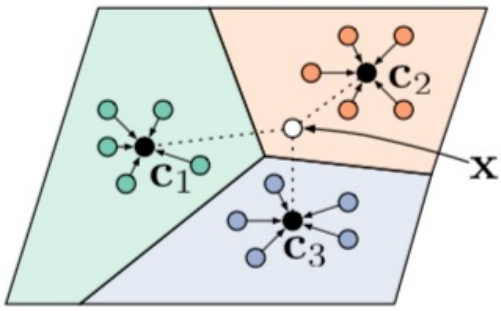
$$c^y = \frac{1}{s}\left(\sum_{y^i = y,\ (x^i, y^i) \in D^{\text{episode}}} \varphi_\theta(x^i)\right);$$

   these are simply the centroids of the features of class y.

3. You can now impose a clustering loss to force the query samples to be labeled correctly, i.e., maximize

$$p_{\theta,w}(y^i \mid x^i) = \frac{e^{-\|\varphi_\theta(x) - c^{y^i}\|_2}}{\sum_{y'} e^{-\|\varphi_\theta(x) - c^{y'}\|_2}}$$

   for all (x^i, y^i) in the query-set of the episode/mini-batch.



(a) Few-shot

4. The objective maximized at each mini-batch is

$$\frac{1}{qs}\left(\sum_{(x^i,y^i)\in \text{query}(D^{\text{episode}})} \log p_{\theta,w}(y^i \mid x^i)\right).$$

   Note that the gradient of the above expression is both on the weights w of the top layer and the weights θ of the lower layers.

5. We can now use the trained model for classifying new classes by simply feeding the new images through the network, computing the prototypes using the few labeled data and computing the clustering loss on the unlabeled query data at test time to see which prototype the feature of a particular query datum is closest to.

Remark 6 (Discussion). Prototypical loss falls into the general category of metric-based approaches to few-shot learning. We make a few remarks next.

1. It is a very natural setting for learning representations of the data for classification that can be transferred easily. If the model is going to be used for new classes, it seems reasonable that the prototypes of the new classes should be far away from each other and the zs of the query samples should be clustered around their correct prototypes.

2. Prototypical networks perform well if you can estimate the prototypes accurately, this requires that you have about 10 labeled data per new class.

3. We used the ℓ_2 metric $\|\cdot\|_2$ in the z-space to compute the affinities of the query samples. This may not be a reasonable metric to use for some problems, so a large number of approaches try to devise/learn new metrics.

4. A key point of prototypical networks is that there is no gradient-based learning going on upon the new categories; we simply compute the prototypes and the affinities and use those to classify the new samples.

25.1.3 Model-agnostic meta-learning (MAML)

We will next look at a simple algorithm for gradient-based adaptation of the model on the new categories. The key idea is to update the model using the same objective in (25.1) but avoid overfitting the model on the meta-training data so that the model can be quickly adapted using the few-shot data via gradient-updates.

Here we consider an episode $D^{\text{episode}} = D^{\text{support}}$, i.e., there are no query shots in the episode. Let us define

$$\ell(\theta; D^{\text{support}}) = \frac{1}{N_n} \sum_{(x^i,y^i)\in D^{\text{support}}} \log p_\theta(y^i \mid x^i);$$

this is the same objective as that in (25.1) so if we maximized the objective

$$\ell(\theta;\ D)$$

we will perform standard cross-entropy training. At each mini-batch/episode, the MAML algorithm instead maximizes the objective

$$\ell^{\text{MAML}}(\theta;\ D^{\text{support}}) = \ell\Big(\theta + \alpha\nabla\ell(\theta; D^{\text{support}});\ D^{\text{support}}\Big). \tag{25.3}$$

In other words, MAML uses a "look ahead" gradient: the gradient of $\ell(\theta;\ D^{\text{support}})$ is not in the steepest ascent direction of $\ell(\theta; D^{\text{support}})$ but in the steepest ascent direction after one update of the parameters $\theta + \alpha\nabla\ell(\theta;\ D^{\text{support}})$.

Adaptation on the few-shot data: One we have a model training using MAML

$$\widehat{\theta} = \operatorname*{argmax}_\theta \ell^{\text{MAML}}(\theta;\ D)$$

we can update it on new data simply by maximizing the standard cross-entropy objective again, i.e.,

$$\theta^* = \operatorname*{argmax}_\theta \frac{1}{N_n} \sum_{i=N_b+1}^{N_b+N_n} \log p_\theta(y^i \mid x^i) - \frac{1}{2\lambda}\|\theta - \widehat{\theta}\|_2^2. \tag{25.4}$$

The adaptation phase is exactly the same as standard cross-entropy training.

Remark 7. (draw a picture of a situation where the lookahead helps)

MAML is not specific to few-shot learning. We can use the MAML objective for any other standard supervised learning problem, is this going to help? Indeed it will, gradient descent/stochastic gradient descent are myopic algorithms because they update the parameters only in the direction of the steepest gradient, you can potentially do better by computing the lookahead gradient. The caveat is that is computationally difficult to compute the lookahead gradient. Observe that

$$\begin{aligned}
\ell^{\text{MAML}}(\theta) &= \ell(\theta + \alpha\nabla\ell(\theta)) \\
&\approx \ell(\theta) + \alpha\left(\nabla\ell(\theta)\right)^\top \nabla\ell(\theta) \\
\Rightarrow \nabla\ell^{\text{MAML}}(\theta) &= \nabla\ell(\theta) + 2\alpha\nabla^2\ell(\theta)\ \nabla\ell(\theta).
\end{aligned}$$

So MAML is secretly a second-order optimization method, computing the gradient of the MAML

objective requires having access to the Hessian of the objective $\nabla^2 \ell(\theta)$. For large models such as neural networks this is very expensive to compute.

Remark 8. Let us consider a meta-training set with two mini-batches/episodes/tasks, $D = D^1 \cup D^2$. The MAML algorithm uses the gradient

$$\begin{aligned}\nabla \ell^{\text{MAML}}(\theta; D) &= \nabla \frac{1}{2} \sum_{i=1}^{2} \ell^{\text{MAML}}(\theta; D^i) \\ &= \frac{1}{2} \sum_{i=1}^{2} \nabla \ell(\theta; D^i) + 2\alpha \nabla^2 \ell(\theta; D^i) \, \nabla \ell(\theta; D^i)\end{aligned}$$

Observe now that if there exist parameters θ that have $\nabla \ell(\theta; D^i)$ for all the episodes D^i then the MAML gradient is also zero. In other words, if there exist parameters θ that work well for all tasks then MAML may find such parameters. However, in this case, the simple objective

$$\ell^{\text{multi-task}}(\theta; D) = \frac{1}{2} \sum_{i=1}^{2} \ell(\theta; D^i) \tag{25.5}$$

that sums up the losses of all the mini-batches/episodes/tasks will also find these parameters. This objective known as the multi-task learning objective is much simpler than MAML's because it requires only the first-order gradient.

What happens if the two tasks are different (as is likely to be the case) in which case there don't exist parameters that work well for all the tasks?

(draw a picture)

25.2 Problem formulation for RL

One mathematical formulation of meta-RL is as follows. Let k denote a task and there is an underlying (unknown) dynamics for this task given by

$$x_{t+1}^k = f^k(x_t^k, u_t^k, \xi_t)$$

We will assume that all the tasks have a shared state-space $x_t^k \in X$ and a shared control-space $u_t^k \in U$. The reward function of each task is different $r^k(x, u)$ but we are maximizing the same infinite-horizon discounted objective for each task. The q-function is then defined to be

$$q^{k,\theta^k}(x, u) = \mathop{\mathbb{E}}_{\xi(\cdot)} \Big[\sum_{t=0}^{\infty} \gamma^t r^k(x_t, u_t) \mid x_0 = x, u_0 = u, u_t = u_{\theta^k}(x_t) \Big].$$

where $u_{\theta^k}(x_t)$ is a deterministic controller for task k. Given all these meta-training tasks, our objective is to learn a controller that can solve a new task $k \notin \{1, \ldots, K\}$ upon being presented a few trajectories from the new task. Think of you learning to pick up different objects during training time and then adapting to picking up a new object not in the training set.

Let us consider the off-policy Q-learning setting and learn separate controllers for all the tasks for now. As usual, we want the q-function to satisfy the Bellman equation, i.e., if we are using parameters φ^k to approximate the q-function, we want to find parameters φ^k such that

$$\underset{\varphi^k}{\text{argmin}} \underset{(x,u,x')\in D^k}{\mathbb{E}} \left[\left(q^{k,\theta^k}_{\varphi^k}(x,u) - r^k(x,u) - \gamma \, q^{k,\theta^k}_{\varphi^k}(x', u_{\theta^k}(x')) \right)^2 \right] \tag{25.6}$$

where the dataset D^k is created using some exploratory policy for the task k. The controllers u_{θ^k} are trained to behave like the greedy policy for the particular q-function $q^{k,\theta^k}_{\varphi^k}$

$$\underset{\theta^k}{\text{argmax}} \underset{(x,u)\in D^k}{\mathbb{E}} \left[q^{k,\theta^k}_{\varphi^k}(x, u_{\theta^k}(x)) \right]. \tag{25.7}$$

The above development is standard off-policy Q-learning and we have seen it in earlier lectures. The different controllers u_{θ^k} do not learn anything from each other, they are trained independently on their own datasets. We can now construct a multi-task learning objective for meta-RL, in this we will learn a single q-function and a single controller for all tasks. We modify (25.6) and (25.7) to simply work on all the datasets together

$$\begin{aligned} &\underset{\varphi}{\text{argmin}} \underset{(x,u,x')\in D^1\cup D^2 \ldots}{\mathbb{E}} \left[\left(q^{\theta}_{\varphi}(x,u) - r^k(x,u) - \gamma \, q^{\theta}_{\varphi}(x', u_{\theta}(x')) \right)^2 \right] \\ &\underset{\theta}{\text{argmax}} \underset{(x,u)\in D^1\cup D^2 \ldots}{\mathbb{E}} \left[q^{\theta}_{\varphi}(x, u_{\theta}(x)) \right] \end{aligned} \tag{25.8}$$

This is the multi-task learning objective for RL. This is unlikely to work well because depending upon the task, the controllers for the different tasks will conflict with each other, it is unlikely that there is a single set of parameters for the controller and the q-function that works well for all tasks.

(draw a picture of a planning task with multiple goals)

Remark 9. How to fix this? You can use MAML certainly to under-fit the controller and the q-function to all the tasks and then adapt them using some data from the new task using gradient updates.

25.2.1 A context variable

Reinforcement Learning offers a very interesting way to solve the few-shot/meta-learning problem. We can append the state-space to include a context variable that is a representation of the particular task. Think of the way we constructed features for a trajectory in inverse reinforcement learning, we considered a set of basis functions

$$\{\phi_1(x,u,r), \phi_2(x,u,r), \ldots, \phi_m(x,u,r)\}$$

and created a feature by mixing them linearly. We will do the same in this case, we construct a variable $\mu^k(\tau)$ for a trajectory $\tau_{0:t} = x_0, u_0, \ldots, x_t, u_t$ from task k as

$$\mu(\tau_{0:t}) = \sum_{s=0}^{t} \sum_{i=1}^{m} \gamma^t \; \alpha_i \; \phi_i(x_s^k, u_s^k, r^k(x_s^k, u_s^k)).$$

Note that the mixing coefficients α_i are shared across all the tasks. We would like to think of this feature vector $\mu(\tau)$ as a kind of indicator of whether a trajectory τ belongs to the task k or not. We now learn a q-function and controller that also depend on $\mu(\tau)$

$$\begin{aligned} &q_\varphi^\theta(x_t, u_t, \mu(\tau_{0:t})) \\ &u_\theta(x_t, \mu(\tau_{0:t})). \end{aligned} \tag{25.9}$$

Including a context variable like $\mu(\tau)$ allows the q-function to detect the particular task that it is being executed for using the past t time-steps of the trajectory $\tau_{0:t}$. This is similar to learning independent q-functions $q_{\varphi_k}^{k,\theta^k}$ and controllers u_{θ^k} but there is parameter sharing going on in (25.9). We will still solve the multi-task learning problem like (25.8) but also optimize the parameters α_is.

$$\begin{aligned} &\underset{\varphi,\alpha_i}{\text{argmin}} \sum_{i=1}^{K} \mathop{\mathbb{E}}_{(x,u,x')\in D^k} \Big[\big(q_\varphi^\theta(x,u,\mu(\tau)) - r^k(x,u) - \gamma \, q_\varphi^\theta(x', u_\theta(x', \mu(\tau)))\big)^2 \Big] \\ &\underset{\theta,\alpha_i}{\text{argmax}} \sum_{i=1}^{K} \mathop{\mathbb{E}}_{(x,u)\in D^k} \Big[q_\varphi^\theta(x, u_\theta(x, \mu(\tau)), \mu(\tau)) \Big]. \end{aligned} \tag{25.10}$$

The parameters α_i of the context join the q-functions and the controllers of the different tasks together but also allow the controller the freedom to take different controls depending on which task it is being trained for.

Adapting the meta-learnt controller to a new task: Suppose we trained on K RL tasks using the above setup and have the parameters $\widehat{\theta}, \widehat{\varphi}, \{\widehat{\alpha_i}\}_{i=1,\ldots,m}$ in our hands. How should we adapt to a new task? This is easy, we can run an exploration policy on the new task to collect some data and update our off-policy Q-learning parameters $\widehat{\theta}, \widehat{\varphi}$ on this data using (25.6) and (25.7) while keeping the results close to our meta-trained parameters using penalties like

$$\frac{1}{2\lambda}\|\theta - \widehat{\theta}\|_2^2 \quad \text{and} \quad \frac{1}{2\lambda}\|\varphi - \widehat{\varphi}\|_2^2.$$

We don't update the context parameters α_is during such adaptation.

Question 10. Does adaptation always improve performance on the new task?

25.2.2 Discussion

This brings an end to the chapter on meta-learning and Module 4. We focused on adapting learning-based models for robotics to new tasks. This adaptation can take the form of learning a reward (inverse RL), learning the dynamics (model-based RL) or learning to adapt (meta-learning). Adaptation to new data/tasks with few samples is a very pertinent problem because we want learning-based methods

to generalize a variety of different tasks than the ones they have been trained for. Such adaptation also comes with certain caveats, adaptation may not always improve the performance on new tasks; understanding when one can/cannot adapt forms the bulk of the research on meta-learning.