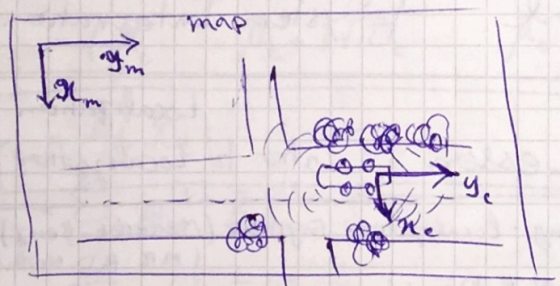
- Intuition: Reduce the uncertainty by measuring/sensing the world and comparing that measurement to the pre known model of the world and hence building a belief.

- Answer the question:
  where is the car in a given map with an accuracy of 10 cm or less.

- GPS ⇝ 3 - 50 meters (not usable



- Find the transformation between ~~local~~ local coordinate system of the car and the global coordinate system of the map by finding position of objects for whom we know their position in the map.

- 1D Baysian Filter in C++
- Motion models
- 2D Particle Filter in C++

20. May 12:00
22. May 20:00
~~3h~~

Lesson 2: Markov Localization

- Localization: Multiplication (Bayes Rule: sense/measurment) and Addition (motion (convolution) Total probability)
- Whole underling math behind the General localization Problem
  - → Derivation of the Bayes Localization Filter (Markov Localization) (one of the most common loc. frameworks)
  - → 1D Realization of Markov Loc. Filter in C++
  - → Motion & Observation Model.
- The project: Kidnapped Vehicle uses Particle Filters.
- Markov Localization is a generalized Filter for localization and all other loc. approaches are realizations of this approach.

- By Understanding the Markov Loc. Filter, we develop intuition and methods that help us solve any vehicle loc. tasks, including Particle Filters.
  - → Think of Vehicle position generally as a prob. dist.
  - → Each time we move ⇝ dist. becomes wider more uncertain
  - → Each time we feed the filter with data (map data, measurement, control) ⇝ the dist. becomes more ~~cert~~ certain more concentrated more narrow

What do we have?

- Map with Landmark positions in global co.sy.
- Observation from the onboard sensors in the local co.sy.
- information about how the car moves between two time steps.

Formally:

- vector $z_{1:t}$ — Observations (range meas., bearing, inge,)
- $u_{1:t}$ — Controls (yaw/pitch/roll rates, velocities, ~)
- m — Map (grid maps, global feature points PB, ~)

Known

- What we want to estimate: Transform between the local coord. sys. of the car and global coord. sys. of the map.

$\Rightarrow$ Pose of the car in the map.

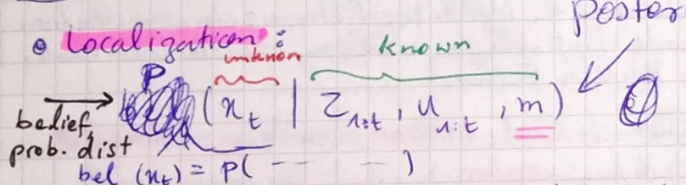
Unknow (position ($\vec{x}_t$ = (x,y) position, orientation ϕ)

- we can never know the pose exactly.
  What we want is a sufficiently accurate Belief

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

formulate in a probabilistic way.

- Localization:



- we assume the map is correct and does not change, so it is also considered given.
  If we wanted to also estimate the map, then we have to solve the SLAM problem.

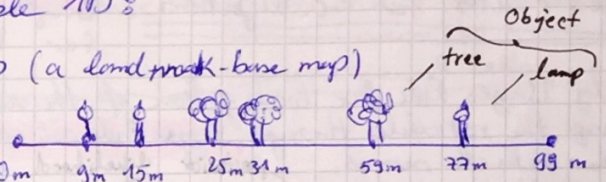
$$P(x_t, m | z_{1:t}, u_{1:t})$$

- Landmark-based map vs. grid-based map
  ↑ more sparse

→

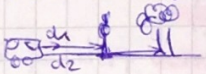
Example 1D:

- map (a landmark-base map)



map = [9, 15, 25, 31, 59, 77]
↳ vector of position of the objects

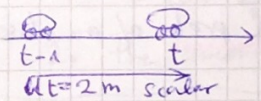
- Observation: the car measures the nearest k seen static objects in driving direction.



observation list $z_{1:t} = \{z_t, \dots, z_1\}$

observ. vector of distances to k nearest objects $z_t = [z_t^1, \dots, z_t^k]$

- Control vector: director move of the car between consecutive time stamps.



$u_{1:t} = [u_t, \dots, u_1]$

- the true pose of the car is somewhere in the map area. Since the map is descrete the pose of the car could be any integer between 0-99

$\Rightarrow$ Belief of x_t: a vector of 100 elements.

$$bel(x_t) = [bel(x_t=0), \dots, bel(x_t=99)]$$

Bayes Filter for Localization: Bayes Filter for Localization

- Apply Bayes Rule for localization of the vehicle by passing the variable through Bayes Rule at each timestep, as the vehicle moves.

Bayes Rule: $P(a|b) = \frac{P(b|a)\,P(a)}{P(b)}$ (posterior, likelihood, prior, evidence (total prob.) (normalizing term))

In Case of Localization:

1. $P(a|b) \rightsquigarrow P(\text{location}|\text{observation})$: the normalize posterior prob. of a position, given an observation
2. $P(b|a) \rightsquigarrow P(\text{Observation}|\text{location})$ the probability of an observation given a location (the likelihood)
3. $P(a) \rightsquigarrow P(\text{location})$: prior prob. of a location
4. $P(b) \rightsquigarrow P(\text{Observation})$: total prob. of an obser.

* $P(\text{location})$ is determined by motion model: the prob. returned by the motion model is the product of the transition model probability (the prob. of moving from $x_{t-1} \xrightarrow{\text{to}} x_t$) and the probability of state x_{t-1}

$$\sum P(x_t|x_{t-1}) \cdot P(x_{t-1})$$

$$P(\text{loc.}|\text{Obs.}) = \frac{P(\text{Obs.}|\text{loc.})\,P(\text{loc.})}{P(\text{obs.})}$$

- Init Belief State: ◦ higher priority for landmarks the car is near to
  ◦ normalize all cells.

- **Data is Huge!**

Imagine:
- at each refresh rate the LIDAR sends 100,000 data points
- each observation contains 5 pieces of data (points id, range, 2 angles, and reflectivity)
- Each piece of data requires 4 bytes
- Drive for 6 hours
- Lidar refresh rate 10 Hz

$\Rightarrow z_{1:t} \approx 430\ GB \rightsquigarrow$ if the update step would take into account all historical observations

won't work!

$\Rightarrow$ two problems: $bel(x_t) = p(x_t | z_{1:t}, u_{1:t}, m)$

1. the localizer code must deal with a lot of data (100s of GB per update)
2. the amount of data increases over time

$\Rightarrow$ Does not work for a Realtime localizer at 10 Hz!

How to change $bel(x_t)$ so that our localizer works with

1. A little data (bytes per update)
2. Amount of data remains constant regardless of drive time.

instead of carrying all the obs. history over at every step

Bayes Localization Filter (or Markov Localization)

We need a recursive structure that allows us to estimate the current state only based on the previous state and

- Let's first break the $z_{1:t}$ into $z_{1:t-1}$ and z_t
- we then apply the Bayes Rule:

$$bel(x_t) = p(x_t \mid z_t, z_{1:t-1}, u_{1:t}, m)$$

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

- The likelihood term

likliood: Observation Model — prior: Motion Model

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

- pay attention that no current observation (z_t) is included in the motion model.
- to simplify, let's define:

$$\eta = \frac{1}{p(z_t \mid z_{1:t-1}, u_{1:t}, m)} = \frac{1}{\sum_i p(z_t \mid x_t^{(i)}, z_{1:t-1}, u_{1:t}, m)\, p(x_t^{(i)} \mid z_{1:t-1}, u_{1:t}, m)}$$

as sum of the products of observation and motion models over all possible states $x_t^{(i)}$. This means in order to estimate the belief we only have to define observation and motion models.

- Now let's focus on Motion model:

$p(x_t \mid z_{1:t-1}, u_{1:t}, m)$. The problem is, we have no information where was the car before (e.g. $\rightarrow$ (e.g state x_{t-1}). What can helps us?

$\Rightarrow$ Law of total Probability

- **Motion Model**

$$p(x_t \mid z_{1:t-1}, u_t, m)$$

- **Total Probability**

$$P(B) = \sum_{i=1}^{\infty} P(B \mid A_i)\, P(A_i)$$

or

$$P(B) = \int P(B \mid A_i)\, P(A_i)$$

- Let's assume a state x_{t-1} is given:

$$p(x_t \mid z_{1:t-1}, u_{1:t}, m) = \int p(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}, m) \cdot p(x_{t-1} \mid z_{1:t-1}, u_{1:t}, m)\, dx_{t-1}$$

- **Markov Property**: memoryless property of an stochastic process.

A stochastic process has Markov Property if the conditional ~~Property~~ Probability of future states (conditioned on all past and the present state) only depends upon the present state and not on the sequence of events that preceed it. Markov assumption is used to describe a model where this property holds. Such model is called Markov Process, such as a HMM.

$$p(x_t \mid x_{1:t-1}) = p(x_t \mid x_{t-1})$$

~~To calculate the state~~ $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots$

we also needed a correctly initialized initial state x_0

- In localization we can have the following two assumptions (Markov Assumptions):

(a) Since we (hypothetically) know in which state the system is at timestep $t-1$, the past observations $z_{1:t-1}$ and control $u_{1:t-1}$ would not provide us any additional information to estimate the posterior x_t; because they were already used to estimate x_{t-1} $\Rightarrow$

$\Rightarrow$ we can simplify $P(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}, m)$ to

$$P(x_t \mid x_{t-1}, u_t, m)$$

(b) since u_t is "in the future" w.r.t. x_{t-1}, then u_t does not tell us anything about x_{t-1}. So we can simplify $P(x_{t-1} \mid z_{1:t-1}, u_{1:t}, m)$ to

$$P(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}, m)$$

$$\Rightarrow P(x_t \mid z_{1:t-1}, u_{1:t}, m) = \int P(x_t \mid x_{t-1}, \cancel{z_{1:t-1}}, u_{1:t}, m) \cdot P(x_{t-1} \mid z_{1:t-1}, u_{1:t}, m)\, dx_{t-1}$$

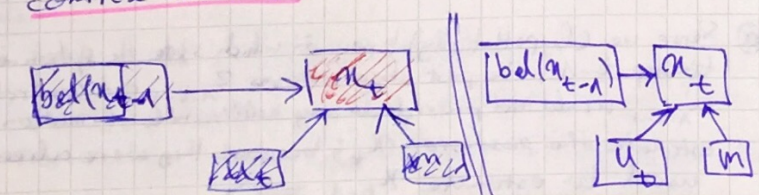
1st & 2nd Assumption

$$= \int \underbrace{P(x_t \mid x_{t-1}, u_t, m)}_{\text{Transition Model}} \cdot \underbrace{P(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}, m)}_{bel(x_{t-1})}\, dx_{t-1}$$

aka (System Model)

$\Rightarrow$ By applying Markov assumptions we achieved a **recursive** (Pay attention that $P(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}, m)$ describes exactly the belief at x_{t-1}) structure that let's us estimate the state at x_t based only on state at x_{t-1} and the observation. [the observation part is not handled yet though]

- A very important step in Bayes localization filter:
  - we use the estimated state from the previous time step for prediction of the current one. Independent from the whole observation and control history.



Summary for Implementation of Prediction Step

- **Motion Model** (discretized) (calculates the prob. that the vehicle is now at a given location x_t)

$$p(x_t \mid z_{1:t-1}, u_{1:t}, m) = \sum_i p(x_t \mid x_{t-1}^{(i)}, u_t, m)\, bel(x_{t-1}^{(i)})$$

Assumption for initial Belief Bel (X=0):
The car knows that it was parked at a tree or a street lamp ± 1 m at the very beginning.

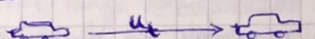
- **Transition (System) Model**: (describes how the system transitions from state x_{t-1} to state x_t)

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

- The transition model is only controlled by x_{t-1} and u_t
- we assume that the Transition Model is independent from the map $\Rightarrow$

$$p(x_t \mid x_{t-1}^{(i)}, u_t, m) = p(x_t \mid x_{t-1}^{(i)}, u_t)$$

- u_t is (in our case) defined as the direct move pointed at driving direction.



$$u_{1:t} = [u_1, \ldots, u_t] \qquad \sigma_{u_t} = 1\ m$$

- the transition model is defined by the 1D normal dist with mean u_t and σ_{u_t}

$$p(x_t \mid x_{t-1}^{(i)}, u_t) = \mathcal{N}(x_t - x_{t-1}^{(i)}; u_t, \sigma_{u_t})$$

and we have to evaluate at position $x_t - x_{t-1}^{(i)}$

- **State Space Range**
  - From 0 to 99 meters with 1m step resolution.

Observation Model

- Reminder: the target function

$$p(x_t \mid z_t, z_{1:t-1}, u_{1:t}, m) = \frac{\overbrace{p(z_t \mid x_t, z_{1:t-1}, u_{1:t}, m)}^{\text{OBSERVATION MODEL}} \overbrace{p(x_t \mid z_{1:t-1}, u_{1:t}, m)}^{\text{MOTION MODEL}}}{p(z_t \mid z_{1:t-1}, u_{1:t}, m)}$$

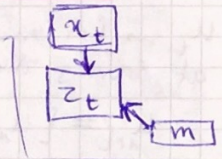
$$= \eta \, p(z_t \mid x_t, z_{1:t-1}, u_{1:t}, m) \, p(x_t \mid z_{1:t-1}, u_{1:t}, m)$$

with η as normalizer

- Again using the Markov Assumption we can simplify the Observation model.
  Since we assume we know the state x_t, it does not really matter what the car observes or how it moved before x_t.
  These values ($z_{1:t-1}, u_{1:t}$) were already used to compute x_t.

z_t only depends on x_t and the map.

$$\Rightarrow \text{Observation model: } p(z_t \mid x_t, z_{1:t-1}, u_{1:t}, m) = p(z_t \mid x_t, m)$$



- Now remember the observation:

$$z_{1:t} = \{z_1, \dots, z_t\}$$

$$z_t = [z_t^1, \dots, z_t^K]$$ ← ranges to the K nearest objects

$$p(z_t \mid x_t, m) = p(z_t^1, \dots, z_t^K \mid x_t, m)$$

- Now we assume the measurements $z_t^1, \dots, z_t^K$ are independent

$$\Rightarrow p(z_t^1, \dots, z_t^K \mid x_t, m) = \prod_{k=1}^{K} p(z_t^k \mid x_t, m)$$

→

← We represent the observation model as the product of individual prob. dists of each single range measurement.

- Now the question is:
  How does should we define the observation model for a single range measurement?
- Observation model depends on:
  - Sensor type (lidar, radar, camera, ultrasound, ...)
  - Different noise behaviors
  - Map type (dense 2D or 3D grid maps, sparce feature-based maps)

- In our 1D example, we assume our sensor measures the range to n closest objects in driving direction. Objects are landmarks represented on our map.
- we also assume range noise is a Gaussian with $\sigma_{z_t} = 1\,m$
- we also assume the range spectrum of the sensor is 1-100 m
- and only forward measurements are included in our observation model.
- we use x_t, and map m to estimate pseudo ranges z_t^*
- pseudo ranges represent the true range value under the assumption the car would be standing at the specific position x_t in the map
- Based on theese assumptions the observation model for a single range measurement is defined by the probability of a normal dist. defined by the mean z_t^{*k} and std. dev. σ_{z_t}

$$p(z_t^k \mid x_t, m) \sim \mathcal{N}(z_t^k; z_t^{*k}, \sigma_{z_t})$$

the highest Π will be where corresponding to your current position is

- For each position x:
  - creat a vector of pseudo ranges (distance from x to every landmark in forward direction) and sort that list: list of z_t^{*k}
  - for every measurment z_t^k find a matching z_t^{*k} (since both lists are sorted, just find the matching indices)
  - calculate $p(z_t^k \mid x_t, m) = \mathcal{N}(z_t^k; z_t^{*k}, \sigma_{z_t})$ and the the Π of all of them

Now Let's finalize the Bayes Localization Filters (Markov Localization)

$$bel(x_t) = P(x_t \mid z_t, z_{1:t-1}, u_{1:t}, m)$$

$$= \eta \underbrace{P(z_t \mid x_t, z_{1:t-1}, u_{1:t}, m)}_{\text{Observation Model}} \cdot \underbrace{P(x_t \mid z_{1:t-1}, u_{1:t}, m)}_{\text{Motion Model}}$$

η: normalizer

- **Observation Model:** (simplified using the Markov Assumption)

$$P(z_t \mid x_t, z_{1:t-1}, u_{1:t}, m) = p(z_t \mid x_t, m)$$

- **Motion Model:** (simplified using M.A.; using the Law of Total Prob.)

$$P(x_t \mid z_{1:t-1}, u_{1:t}, m) = \int \underbrace{p(x_t \mid x_{t-1}, u_t, m)}_{\text{Transition Model (System Model)}} bel(x_{t-1})\, dx_{t-1}$$

$\Rightarrow$

$$bel(x_t) = P(x_t \mid z_t, z_{1:t-1}, u_{1:t}, m)$$

$$= \eta \underbrace{p(z_t \mid x_t, m)}_{\text{Observation Model}} \cdot \underbrace{\int p(x_t \mid x_{t-1}, u_t, m)\, bel(x_{t-1})\, dx_{t-1}}_{\text{Motion Model}}$$

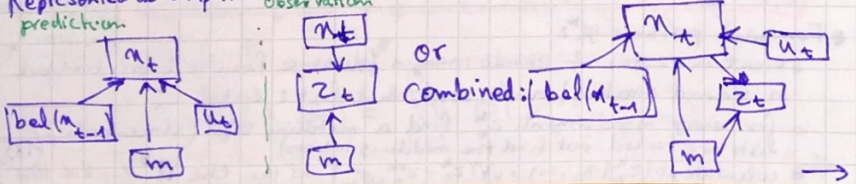
the Motion model is also called Prediction Step and shown with $\overline{bel(x_t)}$

$\Rightarrow$

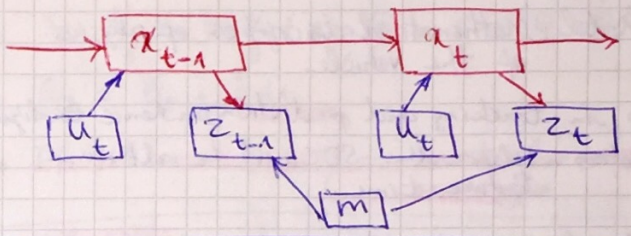
The formula for General Bayes Filter for Localization (Markov Localization)

$$bel(x_t) = \eta\, p(z_t \mid x_t, m) \cdot \overline{bel(x_t)}$$

Represented as Graph:



- It is the common practice to neglegt the map in the Motion Model. Also in the literature the graph is shown without $bel(x_{t-1})$

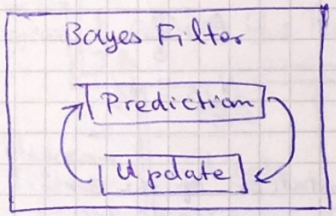


Bayes Localization Filter Summary:

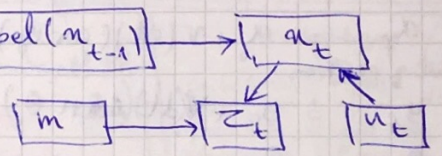
- A general framework for recursive state estimation:
  - → use the previous state to estimate new state
  - → using only current observation and controls. (and not the whole history of data)

MOTION MODEL → Prediction Step of the filter

OBSERVATION MODEL → Update Step to estimate new state probabilities



- Kalman Filters, Particle Filters, ... are all realizations of this general probabilistic filter framework.
  - → probabilistic reasoning
  - → recursive estate estimation

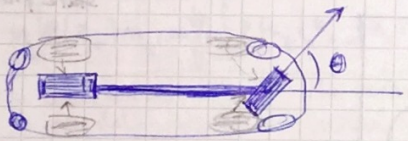


24. May 20:30
24. May 21:15

30 min

Lesson 3: Motion Models

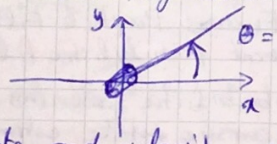
- **Motion Model**: Mathematical description of physics of the vehicle.
  - help in tracking and prediction and hence localization
  - question: where the SDC will be at some future time.
- **Bicycle Model**: Estimate the position of the car, given sensor data, such as num. of wheel turns, velocity, and/or the yaw rate.
- **Assumpsstions**: To apply Bicycle model to a car.
  - ignor vertical dynamics of the car => motion in 2D
  - Front wheels turn together => like 1 wheel
  - Front wheels and rear wheels are connected by using a fix rigid distance.
  - car is controld like a bicycle with steering angle θ and some longitudenal velocity in the heading dir.



$\theta = \frac{\pi}{8}$

$V = 3\ m/s$

- **Car Heading (yaw angle)**: orientation of the car measured from x-axis in map coordinates



- Assuming constant turn rate and velocity: New pos. of the car is

$\dot{\theta} = 0$ yaw rate

$$x_f = x_0 + v(dt)(\cos\theta_0)$$ final x position

$$y_f = y_0 + v(dt)(\sin\theta_0)$$ final y position

$$\theta_f = \theta_0$$

- if yaw rate $\dot{\theta} \neq 0$

$$x_f = x_0 + \frac{v}{\dot{\theta}}\left(\sin(\theta_0 + \dot{\theta}\,dt) - \sin\theta_0\right)$$

$$y_f = y_0 + \frac{v}{\dot{\theta}}\left(\cos\theta_0 - \cos(\theta_0 + \dot{\theta}\,dt)\right)$$

$$\theta_f = \theta_0 + \dot{\theta}\,dt$$

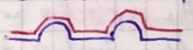
- If the road is hilly => must also consider pitch beside yaw

- **Odometry**: number of wheel turns. (circumference of the wheel)

$$x_f = x_0 + \#\text{turns} \cdot (\cos\theta) \cdot c_{wheel}$$

$$y_f = y_0 + \#\text{turns} \cdot \sin\theta \cdot c_{wheel}$$

- Odometry is erroneous
  - wheel slipping on wet roads and when breaking
  - bumpy roads



24. May 21:30
25. May 20:30

Lesson 4: Particle Filter

3 h

	State Space	Belief	Efficiency	Use in Robotics
Histogram Filters	discrete	multimodal	✗ exponential in number of dim of state space	(exact or) approximate
Kalman Filters	continous	unimodal	quadratic i.e. only if μ_{t+1} and Σ_{t+1} is given.	approximate
Particle Filters	continous	multimodal	?	approximate

But: the most important advantage of PF:

Easy to Program

dim of state space
- can be terrible for anything larger than 4
- sometimes, if used wrong, can lead to very bad performance, but for tracking usually very good.

- it is unknown
- depends on the problem

- **Global Localization:** The robot has no clue where it is, and has to find out where it is just by sensor measurement.

- Each red dot is a discrete guess where the robot might be $(x, y, \text{heading}) \rightarrow$ a single guess.
  - A single guess is not the filter
  - It is the set of several of such guesses that together comprizes an approximate representation of the posterior of the robot.
- At the begining the particles are uniformly distributed. The particle filter makes the survive in proportion of how consistent any of the particles is with sensor measurement.
- Survival of the fittest. Particles that are more consistent with the measurment, are more likely to survive
  $\Rightarrow$ Places of high probability will collect more particles.

- function measurment_prob: A key part of SotF rule in PF, accepts a measurement and tells you how plausible this measurement is.
  - if the measurement is very improbable from $\rightarrow$ [from point of view of the random dummy particle robot]
  Point of the random particle robot, then this particle is at a very wrong place.
  For the next iteration, we only choose those particles that adhere to the measurment.
  That is, only those for whome this value of measurment would make sense.
  The probability of choosing particles for the next iteration is proportional to this weight.

- Particle filters are actually a meta heuristic (like evolutionary algorithms) method:
  - create a large random population of state estimates
  - apply the motion model to them.
  - Do observation, and give each particle a weight proportionate to how much it adheres to the observation. (Importance Weight.)
  - Select particles for the next round based on their Importance Weight

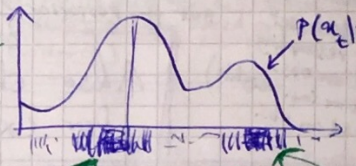
  repeat

- We want to estimate a non linear posterior dist, f but we can't directly sample it.
- So, we use a set of n (weighted) particles X_t to estimate it:
  The density of f is represented by where the particles are and their weights

Goal: That is: if my state estimate is at the end of X_t among the particles, then it is my $(x_t \mid z_{1:t})$

$$p(x_t \in X_t) \approx p(x_t \mid z_{1:t})$$

with equality for $n \rightarrow \infty$



Algorithm particle_filters (input: $S_{t-1} = \{\langle x^j_{t-1}, w^j_{t-1} \rangle\}_{j=1,\dots,n}$, u_t, z_t)

- S_{t-1}: set of particles and their weights from previous step
- u_t: control
- z_t: obs.

1: $S_t = \emptyset$, $\eta = 0$ ← normalizer for weights so that they sum to 1
2: For $i = 1..n$ # resample; generate n new particles
3: Sample index $j(i)$ from the discrete dist. S_{t-1} based on w_{t-1}
4: Sample x^i_t from $p(x_t \mid x_{t-1}, u_t)$ using $x^{j(i)}_{t-1}$ and u_t ← # motion model (Prediction Step)
5: $w^i_t = p(z_t \mid x^i_t)$ ← # measurement update (Observation model) #computing the importance weights for new sample based on the given observation
6: $\eta = \eta + w^i_t$ normalizer
7: $S_t = S_t \cup \{\langle x^i_t, w^i_t \rangle\}$ # add the new sample to posterior

8: For $i = 1 \dots n$:
$w_t^i = w_t^i / n$ # normalize the new weights.

- Particle Filter is actually a realization of Bayes Localization Filter:
  - Motion Model (Motion Update, Prediction Step):

$$\underbrace{P(x')}_{\text{new particle set after the robot motion}} = \sum_i \underbrace{P(x' \mid x^i)}_{\text{posterior estimate}} \underbrace{P(x^i)}_{\text{transition prob.}} \underbrace{}_{\text{prior}}$$

← we sampled by taking a random particle from prior and applying the motion model and the noise model to generate the random particle x'

particles: 1000s of particles together represent the prior X

  - Observation Model (Measurement Update):

$$P(x \mid z) \propto P(z \mid x) P(x)$$

posterior state given the measurement ← resampling to get the posterior based on weights ← importance weights

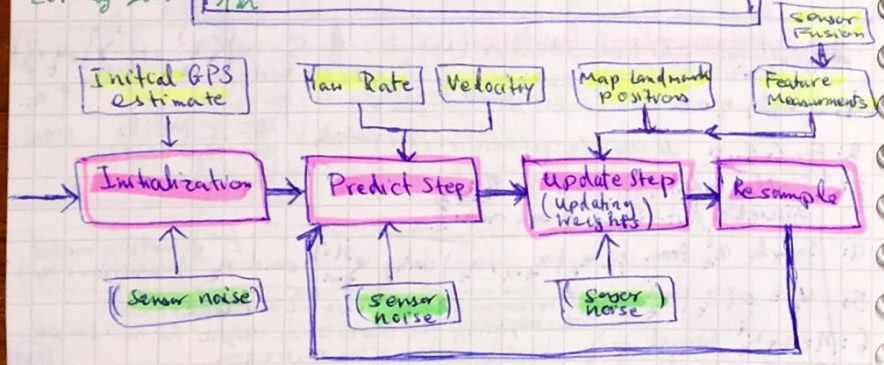
new particle set after resampling based on obs. (based on imp. weights) (based on obs.

Sampling very important in PFs. There some different algorithms for it, check the papers.

Also check the python code for example robot

25. May 20:30
26. May 20:00

Lesson 5: Impl. of Particle Filter
1h



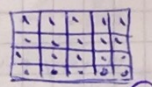
- Particle Filter Algorithm for Localizing an AV

Steps: Initilizing, Prediction, Particle Weight update, Resampling

Particle_Filter (X_{t-1}, u_t, z_t):

1. $\bar{X}_t = X_t = \emptyset$ ← initialization using position estimate from GPS
2. For $m = 1$ to M do:
3. [sample $x_t^{[m]} \sim P(x_t \mid u_t, x_{t-1}^{[m]})$ ← prediction step. We add control input (yaw rate, velocity) for all particles
4. $w_t^{[m]} = P(z_t \mid x_t^{[m]})$ ← update step: update weights for all particles using map landmark position and feature measurements
5. $\bar{X}_t = \bar{X}_t \cup \{\langle x_t^{[m]}, w_t^{[m]} \rangle\}$]
6. For $m = 1$ to M do: ← Resampling Step: drawing particle i proportional to its weight
7. draw i with probability $\propto w_t^{[i]}$
8. add $x_t^{[i]}$ to X_t
9. Return X_t ← Return the Bayes filter posterior (we now refined the estimate of the Vehicle's position based on the input evidence)

- Initialization
  - How many particles? Is usually determined empirically. But we know that the the PF will exactly represent the Baysian posterior dist. when $n \to \infty$
  - too few ⇒ we don't cover all high likelyhood positions. We might miss the correct initial position.
  - too many ⇒ slows down the filter, preventing it from localizing the car in real time.
  - two ways to initialize:
    - sample uniformly accross the state space
      - ↳ not very practical if state space is too large.



- Second way: Sample around some kind of initial estimate, e.g. GPS

- In the project:
  - We initialize the particles by sampling from a Gaussian dist. around the initial GPS estimate and initial heading estimate; Taking to account Gaussian sensor noise

- C++ Standard library: normal dist
- C++ Stl: random engine.

Prediction Step:

- Use the motion model equations to update the position (and other state variables) of each particle using the current position (and other state vars) and taking to account the uncertainty in the control input with Gaussian noise.

$\theta_f = \theta_0 + \dot{\theta} dt$

$x_f = x_0 + \frac{v}{\dot{\theta}}[\sin(\theta_0 + \dot{\theta} dt) - \sin\theta_0]$

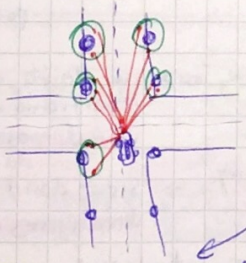
$y_f = y_0 + \frac{v}{\dot{\theta}}[\cos\theta_0 - \cos(\theta_0 + \dot{\theta} dt)]$

Update Step

- Data Association: Nearest Neighbor
  - We have to solve Data Association problems, before we can use measurements to landmarks objects around us for updating the state estimate.

- Data Association: the problem of matching landmark measurements to objects in the real world.

We have multiple measurements around a landmark



Which Measurement to pick for each Landmark?

The nearest neighbor.

affect positively:
- + High signal to noise ratio for sensor
- + a very accurate motion model

- Nearest Neighbor Data Association
  - ⊕ Easy to understand and implement: just take the closest measurement.

- High density of measurments will make NN technique very prone to errors
  - → Because one map landmark could be really close to multiple measurements.
  - ⊖ quite inefficient: for each (landmark) LM you have to go through all the measurment to find the closest ones $O(mn)$
  - ⊖ Spurious (false, fake) features (one that does not corresponds to any real object) if sensor is noisy.
  - ⊖ not robust to errors in the vehicle position estimate
  - ⊖ does not take the uncertaintys of the sensor into account. For example, radar accurate about bearing, but inaccurate about range.

Combine the likelihood of all measmts by taking the product of

- updating the weights: multivariate Gaussian PDF (assuming the measurments are independent)

$$w = \prod_{i=1}^{m} \frac{\exp\left(-\frac{1}{2}(x_i - \mu)^T \Sigma^{-1} (x_i - \mu)\right)}{\sqrt{|2\pi\Sigma|}}$$

x_i = Transformed Measurements i position; μ_i = Measurment's associated Landmark position; Σ = cov. of Measurment

- tells us how likely a set of landmark measurments is, given our predicted state of the car and the assumption that the sensors have Gaussian noise.

Eval: Weighted Avg. Error

RMSE

$$error_{weighted} = \frac{\sum_{i=1}^{M} w_i \sqrt{|p_i - g|}}{\sum_{i=1}^{m} w_i}$$

- another possibility: only take the best (the highest weighted) particle.

$i^* = max(w_i)$ $\quad error_{best} = \sqrt{|p_i^* - g|}$ ← RMSE

Update Step:

- Transform car sensor LM observations from car cord. sys. to map coord. sys.
- Associate the transformed obs. with the nearest LM on the map
- Update particle weights: determine measurment prob. for each LM measurment
  - take product for computing
  - Π

Transform car observations to map coordinates for the particle:

a homogenous transform: { rotation, translation

$$\begin{bmatrix} x_m \\ y_m \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & x_p \\ \sin\theta & \cos\theta & y_p \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix}$$

measurment in map coord | homogenous transformation matrix | particle coordinates | ← measurement in car coordinate

// as if particle is making this observation ~~measurement~~

Calculating Particles Final Weight:

For each measurement:

$$P(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\left(\frac{(x-\mu_x)^2}{2\sigma_x^2} + \frac{(y-\mu_y)^2}{2\sigma_y^2}\right)}$$

$\begin{pmatrix}\mu_x \\ \mu_y\end{pmatrix}$: the position of the landmark associate with (transformed) measurement

$\begin{pmatrix} x \\ y \end{pmatrix}$: position of the landmark observed by the transformed measurement.

27. May
28. May ~~Mon~~
28 May

Project 6: Particle Filter Localization (kidnapped Vehicle)

- C++: std normal dist
- C++: std discrete dist
- C++: Funtors
- C++: for each loop and References! and copy assignment!

vector allocation (size) and vector<>.push_back()!

Lesson 7: Search

Path Planning
May 28. 16:00
28 May

- Motion Planing: input: perception and Localization
  - Predictive task: what are other vehicles are about to do
  - pick behavior: e.g. ~~Kreuz~~ steer left or right.
  - generate trajectory: also has a time dimension (i.e. speed of the vehicle considered)
- Path Planing (= Motion Planing)
  - generate safe drivable trajectories to goal positions
- Perception, Localization } input → Path Planning:
  - decide which manover to take next.
  - construct a trajectory for the controller to execute.
- Foundational Search Algorithms used in Discrete Path Planing
- Prediction: use the data from sensor fusion to predict what the objects around us are ~~about~~ likely to do
- Behavior Planing: decide at the high level what the car shall do in the next 10 sec. or so
- Trajectory Generation: create smooth, driveable, and collision free trajectories for the motion controller to follow.

- Avoid collisions | maintain safe distance to other vehicles | pass other vehicles |

- Discrete Path Planning vs. Continous Path Planning

Motion Planning = Path Planning

- Cost function
- Optimality : best vs. good enough
- Online vs. offline algorithm

- Robot Motion Planning : the process of finding a path from Start loc. to Goal loc.

- Planing Problem :

Given : Map, Starting Location, Goal Location, cost Function

Output : Find the minimum cost path.

- Path Planning as a Search Problem :
  - Start , Frontier
  - expand, g value : length of the path (Cost)
  - → take the node with lowest cost from frontier and add its not seen neighbors to the Frontier. (removing it from frontier)

[Check the Search Lecture From Intro to SDC]

- A* : Peter Hart, Nils Nilsson, Bertram Raphael.

- uses a heuristic function :
  an optimistic estimate of the cost from a point to the goal

for example for the grid world from lecture

$h(x,y) \leq$ real distance to goal From x,y

- it must underestimate the cost to the goal.
- ≈ admissable

→

- A* uses $f = g + h$ as the cost and expands the node with least f value.
- minimizing g ↓ ⟹ keeps the path short
  minimizing h ↓ ⟹ keeps the search directe towards goal

- A* in action
  - Amazing!
    - finding the path to goal through a maze (the maze is unknown up front and its map gets updated as the car moves inside it)
    - Parking
    - Road blocked; the car did a multipoint U-Turn!

An alternative approach for Planing : Dynamic Programming

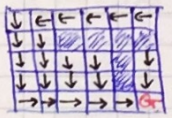
like A* it gives a shortest path

Given : • Map • Goal

Output : • Best path from any possible starting position. (Anywhere)

- Why do we need it ?
  - Because the environment is stochastic. The outcome of actions are not deterministic. Example, the car wants to take a left turn, but suddenly a big truck blocks the way; so the car has to take another path..

- DP gives you a plan for every position!
- DP gives an optimal action for every single grid cell



- Each grid cell has a label (Policy)
- Policy is a function that maps each grid cell to an action

Policy(x,y) ⟶ action

- **Dynamic Programing** for Robot Path Planning
  - write a software that for each of the navigable cells outputs what the best thing is to do, if the robot is in that cell. // Given a grid world and a goal //
- More computationally than A*

5	4	3	2
6		2	1
7		1	0

The **Value** function f assigns to each grid cell, the length of the shortest path to the goal

$$f(x,y) = \min_{x',y' \in \text{Neighbors}(x,y)} f(x',y') + 1$$

- take the optimal neighbor x',y' and add 1 to its $f(x',y')$ value
- The optimal control action is obtained by minimizing the value (A **Hill-Climing** process)

Check the Jupyter Notebook

- The ~~function~~ code for evaluating the ~~cost~~ value function sweeps the state space propagating value 0 from Goal State to other cells.
- Each cell gets updated with one an action that propagate the minimum value from its neighbor to it
- That action is actually the optimal action for that cell.
- The optimal policy returns the sequence of optimal actions.

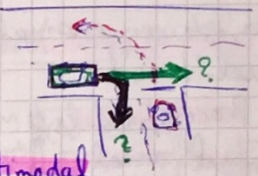
01. June
02. June 17:00

Lesson 8: Prediction

1h 30 min

- Mahmi Shayegan for : Mercedes-Benz Sr. Vehicle Intelligence Eng. (Prediction Team)

- Prediction: intuition
  - you want to turn left at intersection
  - another vehicle comes



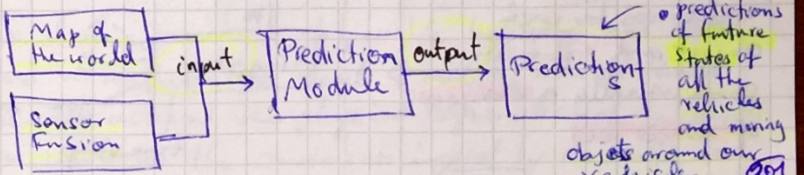
- Prediction is **inherently Multimodal**
  - Where is the green car likely to be in 5 sec?
- We handle multimodality (multiple peaks in prob. dist.) by maintaining a belief about how probable each potential mode is:

  Turn Right vs. Go Straight

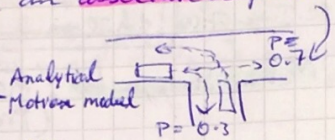
- initial beliefs can be initialized using some prior knowledge about this intersection: eg. 85 of the cars go straight at this intersection

Imagine the car turns right

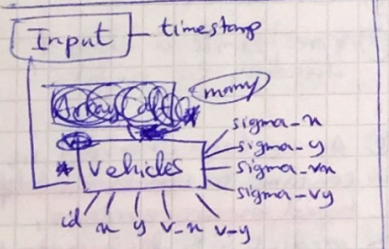
0. initial belief
1. car slows down
2. car starts a chang of heading to right
⋮
6. As we keep observing, we continue to update our belief based on new evidence, until we eventually we can predict with high certainty that the car is turning right at this intersection

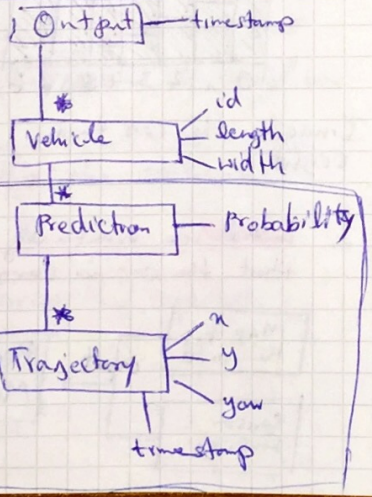


- Predictions are typically presented as the set of possible trajectories; and an associated prob. for each trajectory.



- Model-based prediction ← Analytical Motion model
  VS.
  Data-driven approaches ← ML
- Data-driven example: Trajectory Clustering
- Model based:
  - Process models: Mathematical tool for modeling various maneuvers like lane changes, vehicle following, etc.
  - Multimodal Estimators: An effective technique for handling the uncertainty associated with prediction: uncertainty about which maneuver an object will do in a particular situation.
- Hybrid-Approaches: use data and process models to predict motion through the cycle of intent classification
  - Intent Classification (what an object wants to do?) → trajectory generation (how is it going to do it?)
- Naive Bayes to predict the motion of a car at a T-shaped intersection.

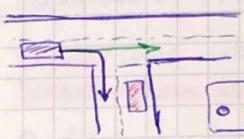




- The array of predicted trajectories for each object and probability spans usually a horizon of 10-20 secs.

→

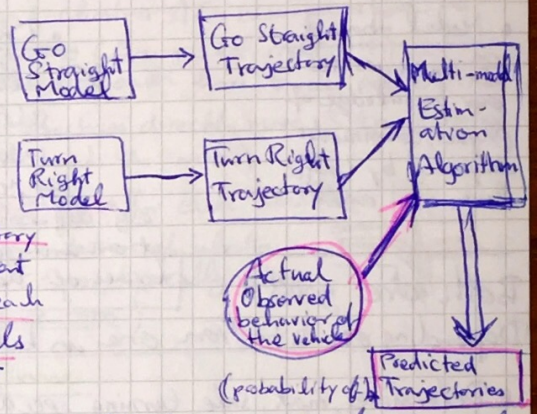
- The resolution is fine ≤ 0.1 sec?
- Predictions for all moving objects (cars, bikes, peds.)
- For each object, the sum of probabilities of all possible trajectories sums to 1



- Red car need to predict what the blue car is going to do.

Model-based Approach

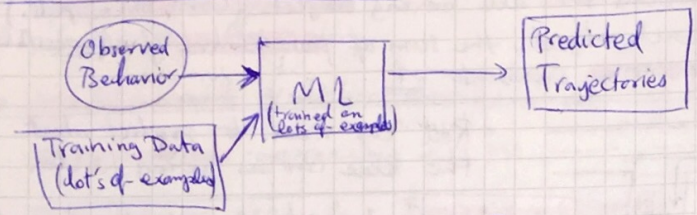
1. Come up with two process models: one for going left, and one for going right
2. Use a single trajectory generator to figure out the trajectory for each of the process models: straight or right
3. Pay attention to the actual behavior of the target vehicle
4. Use a Multimodal Estimation Algorithm to compare the observed trajectory to the ones we would expect for each of our models
5. Based on that, we would assign a probability to each of the possible trajectories.



Important Take-away for purely Model-Based prediction:

We have some bank of possible behaviors and each has a mathematical model of motion which takes into account the physical capabilities of the objects as well as the constraints imposed by the traffic laws.

• Data Driven Approach



• Which approach is the Best?

• Model-based

⊕ incorporates our knowledge of physics, constraints imposed by traffic laws, etc. and

• Data-driven

⊕ Allows using data to extract subtle patterns, that otherwise would be missed by ~~the~~ Model-driven approach, e.g. differences of vehicle behavior at an intersection during different times of day.

Both are good.
Depending on situation one is better:

- Determine max safe turning speed on a wet road → (Model-based: using our knowledge of physics and vehicle dynamics, an exact prediction is possible)
- Predicting behavior of an unidentified object sitting on the road → Data Driven: using model-based is nearly impossible, if we don't even know what the object is.
- Predicting the behavior of a vehicle in a two lane highway in light traffic → Both work, a hybrid approach: Few behaviors to model; easy to collect a lot of data

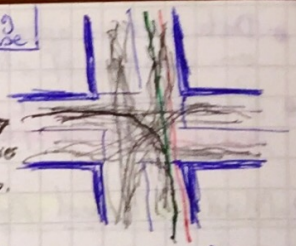
• Data-driven example: Trajectory Clustering

→ • offline training phase
→ online prediction phase

→

Offline Training Phase:

1. Get a lot of data, e.g. by placing a static camera at the intersection

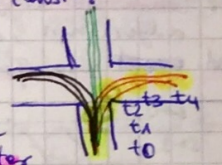


2. Clean the data, e.g. remove occluded cars, ... → resulting trajectories after data clean up.
3. Define some math. measure of trajectory similarity (1 and 1 more similar, than 1 and 7)
4. Perform unsupervised clustering (e.g. agglomerative clustering or spectral clustering). In example above we expect to see 12 clusters (forward, left, right x 4). If we had traffic lights we would have twice possibilities for each stop (3 trajectories directly move, 3 trajectories first stop for a while and the move)
5. Define prototype trajectories for each cluster (provide a compact representation of what a trajectory, e.g. a left turn, typically look like at this intersection)

• Online Prediction Phase

- we have now Prototype trajectories and cluster

1. Observe the vehicles partial trajectory



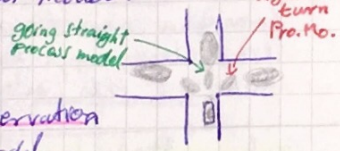
2. Compare it to the corresponding segment of the prototype trajectories of each cluster (using the similarity measures used for clustering)
3. The Belief for each cluster is updated based on how similar the partial trajectory is to the prototype trajectories.
4. For each cluster compute a predicted trajectory with probability by taking the most similar prototype trajectory.

- At the begining the partial traj. overlappes with all three clusters.
- As we move prob. for red grows

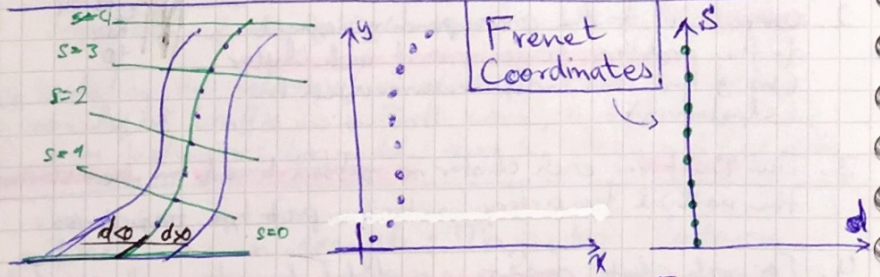
- Data-driven approaches are naive, because they solely depend on history of observations.
- Ideally we want to integrate our useful insights about driver behavior, physics, or vehicle dynamics.

Model-Based Approach:

1. Identify common driving behaviors (e.g. change lane, turn left, ... for cars, or cross the street for pedestrians.)
   [we need to be able to describe it mathematically]
2. Define a process model for a behavior: A mathematical description of object ~~model~~ for a behavior. A function
   - that can be used to compute the motion state of the object at time $(t+1)$ from the state of the object at time t.
   - Process model must incorporate some uncertainty, which reflects how much we trust our model.
   - if we keep running the process model, our uncertainty increases



3. Update beliefs by comparing the observation with the output of the process model
4. Generate trajectories by using a multimodal estimation algorithm: and iterate on the process models until the pred. ~~event~~ horizon is reached.



Cartesian coord. Curve: (x, y)

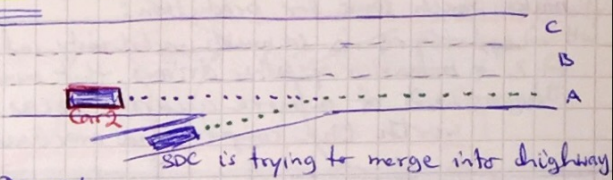
Frenet coord. Straight line: (s, d)

s: longitudinal displacement

d: lateral displacement

Process Models

Possible Car 2 Behaviors:



- Ignore us —Proc. Mod.→ Follow lane A with constant Velocity
- Speed up, so that we enter the highway after it —Proc. Mo.→ Follow lane A with positive acceleration
- Slow down, to let us get ahead of it —P.M.→ Follow lane A with negative accel.
- or, it might change lanes.

→ Lane following on lane B with constant velocity.
↳ in Frenet terms

- Four Models for Lane Following: trade-off between simplicity and complexity
- Linear point Model (constant velocity)
  - treat the car as a point particle with holonomic properties
    ↳ we assume the point can move at any direction at any time
  - too simplistic

$$\begin{bmatrix} \dot{s} \\ \dot{d} \end{bmatrix} = \begin{bmatrix} \dot{s}_0 \\ 0 \end{bmatrix} + \mathbf{w}$$

w: uncertainty matrix/vector (normally a multivariate Gaussian with zero mean)

  - the car moves forward at each time step and is assumed to keep a constant distance to the lane center.
- Non-linear Point Model (constant accel. with curvature)
  → cos, sin in motion equations
- Next step in complexity: Kinematic bicycle Model with Controller (PID controller on distance and angle)
  ↳ Car is non-holonomic system
- Next: Dynamic bicycle model

- In practice, using these more complex models does not make much sense for prediction:
  → there is so much uncertainty inherent in the behavior of other drivers, that minor accuracy improvement of a more accurate process model, is just not worth the computational overhead that they come with

Multimodal Estimation

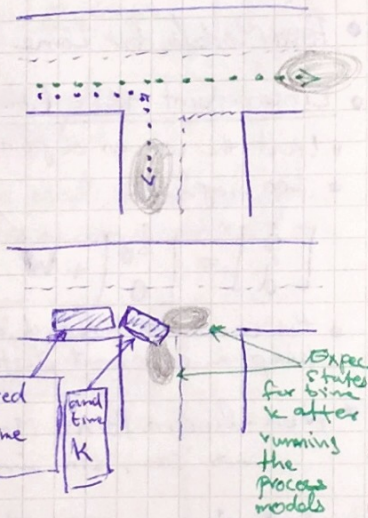
- maintain a belief about which behavior the driver intents to perform.

Autonomous Multiple-model Estimation (AMM)

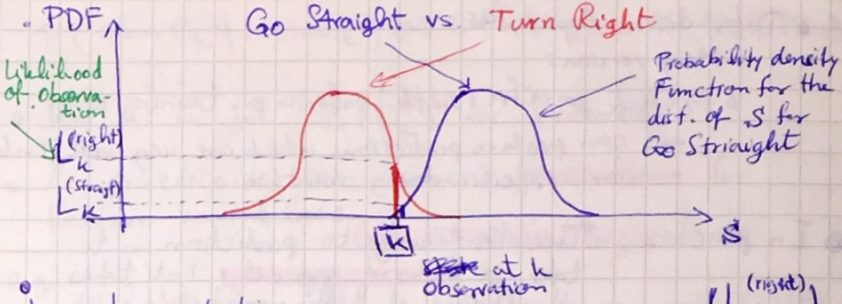
- Consider a set of M process models (=behaviors)
- probabilities for process models $\mu_1, \mu_2, \dots, \mu_M$

Example:

- Let's say we have two process models:
  - one to go straight --->
  - one to go right ...>
  and they both have Gaussian uncertainty.



- Let's say we do an observation at time k-1 and the next one at time k
- In order to update the new probabilities for the behaviors based on the new observation we will run our two process models for one step starting at time k-1
- we then get two expected states and for time k
- now let's look at the distribution of vehicle's s coordinate →



- Observation at time k is substantially more consistent with turn right ($L_k^{(right)}$) that with Go Straight ($L_k^{(straight)}$)
- This is measured by the likelihood of the observation s for each model.
- Now, the probability of each behavior is a function of these likelihood and of the probabilities computed in the previous steps.

$$\mu_k^{(i)} = \frac{L_k^{(i)} \; \mu_{k-1}^{(i)}}{\sum_{j=1}^{M} \mu_{k-1}^{(j)} \; L_k^{(j)}}$$

$\mu_k^{(i)}$: probability of model i at time k (posterior)
$\mu_{k-1}^{(i)}$: probability of model i at time step k-1 (prior)
$L_k^{(i)}$: Likelihood of the observation at time k for model i
$\sum_{j=1}^{M} \mu_{k-1}^{(j)} \; L_k^{(j)}$: normalizer

$$\text{posterior} = \frac{\text{Likelihood of obs.} \cdot \text{prior}}{\text{normalizer}}$$

Bayes :)

- in our example of T-shaped intersection, M=2 (we have two process models: go straight and turn right) (only considering two maneuvers)

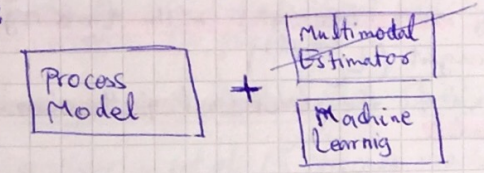
- Model-based Approaches
  - Process model: incorporate our knowledge of object motion dynamics
  - Multimodal Estimators: handle uncertainty on maneuvers.
  - Many approaches (ways) to implement.

→

- Data-driven Approaches
  - Many versions
  - Extract specific (subtle) patterns on training data
    ⇒ can produce predictions which are very well-tailored for specific driving situations.

- In practice: The best way to prediction is to take a hybrid approach, that takes advantages of both approaches.

e.g:



e.g. replace the Multimodal Estimator of a model-based approach with a Machine Learning component (a classifier)

- One strategy that is often used in hybrid Approaches for behavior classification is to combine a classifier with a filter.

- Naive Bayes Classifier:
  - Naive, because it assumes independence between features. $f_1, \dots, f_n$

$$P(\text{class } C_k \mid \text{instance}) = \frac{P(f_1 \mid f_2, f_3, \dots, f_n, C_k) \cdot P(f_2 \mid f_3, f_4, \dots, f_n, C_k) \cdots P(C_k)}{P(f_1, \dots, f_n)}$$

$$\overset{\text{(if dep. assumption)}}{=} \frac{P(f_1 \mid C_k) \cdot \dots \cdot P(f_n \mid C_k) \cdot P(C_k)}{P(f_1, \dots, f_n)}$$

Example:

Classes: Male (M), Female (F)
features: height (h), weight (w)

$$P(M \mid h, w) = \frac{P(h \mid M) \cdot P(w \mid M) \cdot P(M)}{P(h) \cdot P(w)}$$

not important, just the normalizer

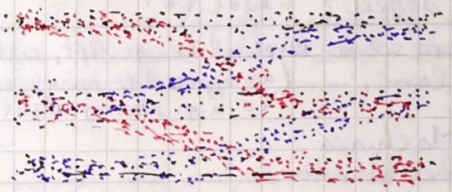
$$P(\text{class } C_k \mid \text{instance}) \propto P(f_1 \mid C_k) \cdot \dots \cdot P(f_n \mid C_k) \cdot P(C_k)$$

- How do we get $P(f_1 \mid C_k)$ for features?
- In practice we can assume a Gaussian dist for feature variables: Gaussian Naive Bayes

e.g. $P(h \mid \text{Male}) \sim \mathcal{N}(\text{mu\_male\_height}, \text{var\_male\_height})$

- In practice, implementing a Gaussian Naive Bayes Classifier:
  1. Select ~~correct~~ relevant features
  2. Identify the mean and variances for $P(f \mid C)$
     → either guess
     → or make lots of observations and learn

- C++ Imple of a GNB Classifier for highway
  - 3 lane autobahn; 3 classes (behaviors): { left, keep, right }



1. train:
   - calculate priors for each class $P(C_k)$
   - calculate the μ and σ^2 for each attribute for each class ("feature/label pairs")

2. Predict
   - calculate the conditional probability for each feature/label pair

$$P(x = v \mid C) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{(v-\mu)^2}{2\sigma^2}}$$

x: feature, v: value from instance

← 2. Use the conditional probabilities in Naive Bayes

$$\hat{y} = \underset{k \in (1..K)}{\arg\max}\ p(C_k) \prod_{i=1}^{n} p(x_i = v_i \mid C_k)$$

- $\hat{y}$: predicted class
- K: num. of Classes
- $p(C_k)$: class prior for C_k
- n: num. of features
- C_k: label (class)
- $p(x_i = v_i \mid C_k)$: conditional prob. for feature/label pair: $\mathcal{N}(v_i; \mu_i, \sigma_i)$

Considered

- In this lesson we ~~took~~ only one moving object at a time
- If we have multiple ~~objects~~ objects, then we have to take also interactions between objects into account. which becomes very complex, very quickly.
- In practice only considering 1 object at time can still deliver useful result for situations like highway.

02. June 18:00
03. June 20:00

Lesson 9: Behavior Planning

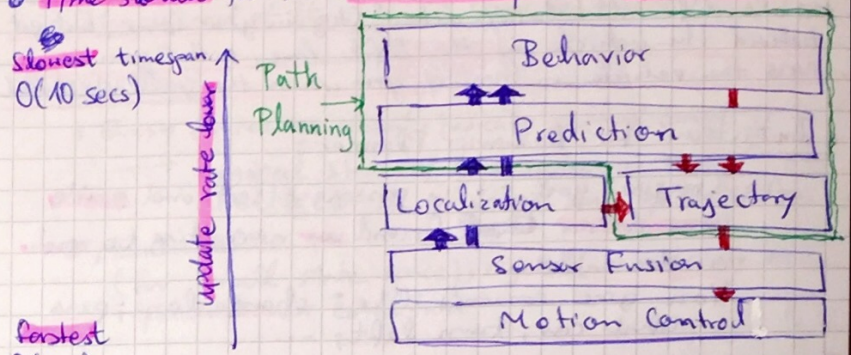
2h

- One of the most challenging task in SDC
  - → what to do next? where to?
  - → at Macroscopic level like go left, right, change lane, ... (as opposed to minute control input to e.g. steering wheel)
- Finite State Machines for controlling the behavior
- Benjamin Ulmer; Tobias Roth (Mercedes Benz)

- Behavior Planning team is responsible for providing guidance to the Trajectory Planner about what sort of manoeuvres they should plan trajectories for.

→

- Time scales for the overall flow of data on a SDC



- Input to Behavior Planner comes from Prediction and Localization
- Input to Prediction and also Localization comes from Sensor Fusion
- Output of Behavior Planning goes to Trajectory Planning
- Trajectory Planning also gets input from Prediction and Localization.
- Output of Trajectory Planner goes to Motion Controller.

- Behavior Planner has to incorporate a lot of data to make decisions about fairly long time horizons in order of 10 sec. or more.

– Intro
  → inputs and outputs to the Behavior Planner
  → what problem Behv Planner solves

– Finite State Machines
  → One technique for implementing a behavior planner

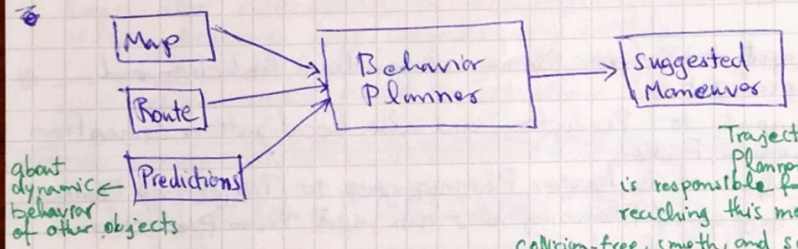
– Cost Functions
  → are used to make behavior level decisions.

213

- Understanding the output: simply by specifing different values for quantities like "target_lane", "target_vehicle (to follow)", →

← "target_speed", and time to reach these targets, it is possible different behaviors like "stay in your lane, but get behind the vehicle in the left lane, so that you can pass the vehicle in front of you when the speed is reached"

- **Intuition for Behavior Planner:**
  - Imagine you sit in passenger seat and guide your driver friend in navigating to reach a destination.
    You give comands like: change lane; pass that car; turn left; ...



Behavior Planner: responsible for

Suggest "States"/maneuvers which are:
- Feasable
- Safe
- Legal
- Efficent

Not Responsible for:
- Execution Details
- Collision Avoidance

- What we learn here was used in DARPA Urban Challenge and Bertha Benz Drive
  - → capable of handling complex traffic situations
    - → navigating intersections
    - → dense urban traffic
  - → not the most common anymore
  - → But very widely known in the field and it give a very high-level understanding of a behavior planner

- Finite State Machine (one, self transition, accepting state, transition function, ...)
  I know most of it

FSM

Strengths
- Easy to reason about; basically self-documenting
  → Logical state of the system is mapped directly to the physical state of the system.
- Maintainable (for small state space); e.g add a new state to represent new condition.

Weaknesses
- Easily abused (e.g. adding too many new states)
  ↳ if not designed well at the beginning, or if the problem changes
- Hard to maintain (if state space gets bigger); messy code

Example States for an SDC driving in Highway

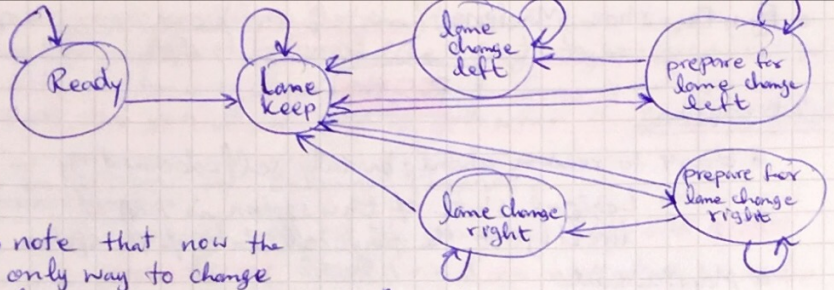
Accelerate
Slow down
keep lane
Change lane
Follow vehicle
Stop
Pass vehicle
Keep target speed
change lane left / right
prepare change lane left / right
⋮

- There no single correct set of states for a given scenario; not easy to find the perfect state space
- **Trade-off between**
  - have enough logical states to represent all physical states we care about
  - vs. Keep state space as small as possible

- **States for this Lesson:**
  - keep lane; change lane right; " " left; prepare change lane right; " " " left

- See the video in lecture about why the two "prepare" states are needed:
  - adapting to the gap position
  - adapting to the speed of target lane
  - triggering the turn signal

→ turn signal is the responsibility of behavior team



- note that now the only way to change lane is over "prepare.." states

States:

LANE KEEP:
- d: stay near center line for lane
- s: drive at target speed for lane when feasible
  - otherwise ... drive at whatever speed that is safe
    - d: Stay near center line for lane

LANE CHANGE LEFT/RIGHT:
- d: move to target d of left/right lane
- s: same rule as keep lane (for initial lane) safe speed

PREPARE LANE CHANGE LEFT/RIGHT:
- d: stay near center line for current lane
- s: attempt to match position and speed of "gap" in target lane
- signal: activate turning signal

- The input to the transition function:
  - Predictions
  - speed limit
  - current state
  - Map
  - Localization data

- Transition Function

One possible way to impl. a trans. Function is →

- generate rough trajectories for each accessible "next" state
- Find the best among them using a cost function

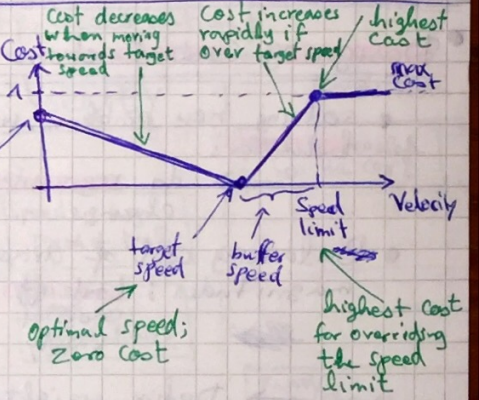
```
def transition_function(predictions, current_fsm_state, current_pose,
                        cost_functions, weights):

    possible_successor_states = get_successor_states(current_fsm_state)
    for state in possible_succ_states:
        trajectory_for_state = generate_trajectory(state, current_pose,
                                                   predictions)
        cost_for_state = 0
        for cost_function in cost_functions:
            cost_for_state += weight[i] * cost_function(trajectory_for_state
                                                        , predictions)

    find and return state with min cost.
```

- Cost functions:

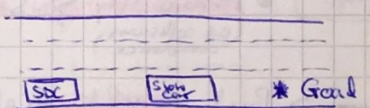
e.g.
- Speed penalty



high cost; but less than overriding the speed limit

- we define all these rules ourselves;

- e.g. Lane Change Penalty
  - should SDC change lane to (LC) pass the slow car? Or should it keep lane? (KL)



$\Delta s = s_G - s$: distance s to goal

$\Delta d = d_G - d$: lateral distance to goal

$\Delta d_{KL} = d_G - d_{KL} = 0$ & $\Delta d_{LC} = (d_G - d_{LC}) > 0$ →

- So, our cost function must be proportional to Δd and inversely proportional Δs
- We want to penalize large $|\Delta d|$ and want that penalty to bigger when Δs is small.
- We also want the cost function to be between 0 and 1 (just for convenience)

$$\Rightarrow \quad \text{Cost} = 1 - e^{-\frac{|\Delta d|}{\Delta s}}$$

- Functions with bounded ranges (e.g. e^{-x} for $x > 0$; or the sigmoid function $S(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$) are useful when defining cost functions.

Designing Cost Functions is difficult

- solving new problems, without breaking working behaviors
  → do regression testing after each change
- Balancing cost of drastically different magnitudes; trade off or no trade off e.g. in case of comfort or efficiency vs. safety

→ Define weights:

- Feasability ≫ Safety ≫ Legality ≫ Comfort ≫ Efficiency
  (e.g. collision avoidance)

→ But weights might differ depending on situation too!
e.g. A red traffic light gives a very much higher weight to Legality compared to keeping the speed limit in a high way.

- Reasoning about individual CFs
  - the specific responsibility of each CF
    → e.g. defining several CFs each addressing one concern (e.g. Legality; Efficiency, ...) w.r.t. speed
  - Binary vs. descrete vs. continous CFs
    e.g. (speed > speed-limit)? 1 : 0
  - Standardize all CFs to be between -1 and 1
  - Parametrize whenever possible
    → we can then use param. optimization (e.g. gradient descent) to tweak CF
  - Thinking in term of vehicle state is helpful (position, velocity, acceleration)

Class	Position	Velocity	Acceleration
Feasability	Avoid collisions?		Accel. is feasible for the car?
Safety	Buffer Distance	Speed ≈ avg. speed of traffic	
Legality	Stay on Road?	Speed < speed-limit?	
Comfort	Near center of current lane		Rapid/Low change in accel. (jerk)
Efficiency	Desired Lane	Speed ≈ Speed-limit	

- Depending on situation (e.g. merge into highway when traffic is dense; or reaching a green traffic light that just gets yellow; ...) these CF need to have many different weights.

- there are also many many other CFs not listed in the table below, some of them (eg. Obey Traffic Rules) can't ~~be~~ be expressed in terms of position, velocity and acceleration.

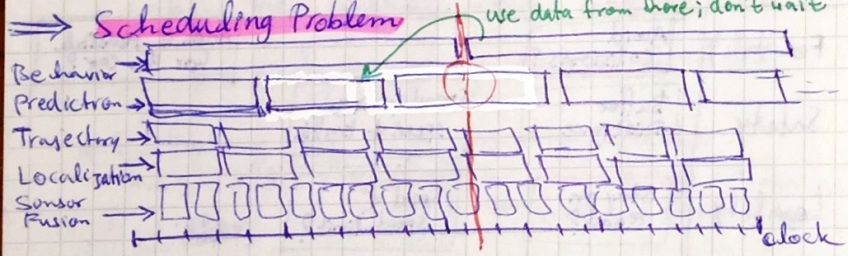
- It becomes very rapidly very complex. This complexity is to parts inherent in the problem, and to parts as a result of using FSMs for modelling behavior.

Scheduling the Compute Time:

- The Behavior module updates in a lower frequency than for example Trajectory Module
  → Because the high level decisions made in Behavior Planner span a longer ~~the~~ time horizon and just don't change very frequently

- But the Trajectory Module still counts on the Decisions of the Behavior Module. So, how to avoid that BM does not clock up (slow down) the TM and whole system?

⇒ **Scheduling Problem**



- Imagine Behavior just finished its first cycle.
  → it needs data from prediction and Localization to begin its second cycle
  → For Localization it is ok. Because at this stage it ~~must~~ has some fresh data
  → But what about prediction? it is still at the middle of an update cycle.
  Should Behavior just wait until prediction is done?
  No! If we start waiting, then we blockup the pipeline for down stream components
  ⇒ Use data from previous prediction cycle and accept that it is a little stale

---

03. June 20:00
08. June

2 h

Lesson 10: Trajectory Generation

- Trajectory Generation: A time sequence of positions, velocity and acceleration, ... for the car for a given planned behavior
  → Avoid Collisions
  → Avoid jerks (≠ uneven accelerations) (smoth accelerations)

- Emmanuel Boidot (Mercedes Benz)

- Continous trajectory Planning
  → How to generate Drivable Trajectories

- Motion Planning Problem
  → concepts and properties of MP Problem
  → Classification of Planning Approaches

- Review of A*
  → Hybrid A*

- Continuity, Optimization & Constraints
- Discreet vs. Continous

- Sampling-Based Method
  → Polynomial Trajectory Generation
    → very useful for Highway Driving

Overview

The Motion Planning Problem

we already got familiar with some Algorithms: Dynamic Programming, Optimal policy, A*

- Configuration Space:
  - defines all the possible configurations of a robot in a given world.
  - In a 2D world the Conf. space can be 2D [x,y] or 3D [x, y, θ (heading)] or even more depending on the MP algorithm

- MP Problem:
  Given: physics, map, traffic
  - A start conf q_{start} (current config of the car given to us by Localization & sensors (car location, speed, accel, ...))
  - A goal conf q_{goal} (Defined by Behavior module also gives some constraints regarding where to go, and speed limits, ...)
  - constraints (describing vehicle physics & dynamics and the description of the Environment)

← the prediction module gives us information about how the obstacle region will evolve in time.
↳ So that, the sequence of actions we generate takes into account other vehicles and pedestrain actions.
And we had a more sophisticated Prediction Module, how our actions might influence them.

Problem:

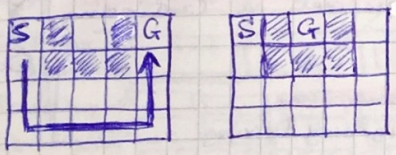
Feasible motion planning: Find a sequence of movements in configuration space that moves the robot from q_{start} to q_{goal} without hitting any obstacles.

Properties of a Motion Planning Algorithm:

Completness:

Always Return a Solution:

- If a solution exists the planner will always return a solution
- If no solution exists, the planner will terminate and report failure



X: terminate with failure

Optimality:

- Given a cost function (for evaluating the sequence of actions) the planner always returns a feasible sequence of actions that minimizes the cost (minimal cost)

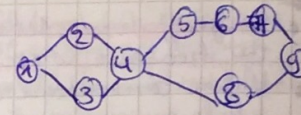
Classes of Motion Planning Algorithms:

There are many classes of MP Algorithms: (some listed)

Combinatorial Methods

divide the free space into small pieces and first solve the MP problem by connecting these atomic elements

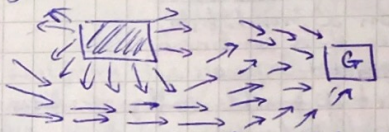
→



- very intuitive way for finding an initial aproximate solution, but usually do not scale well for large environments.

Potential Fields (a.k.a. Reactive) Methods:

- Each obstacle is going to create a sort of anti-gravity field which makes it harder for the vehicle to come close to it.
- For example this can be used around pedestrians or bikes to encourage the planning algorithm to find trajectories that stay away from them.
- The main problem with the most potential field methods is that they sometimes push us into a local minima.



Optimal Control

- solve the motion planning problem and the control input generation in one problem.
- Using a dynamic model of the vehicle, a start config, and an endcon. we want to generate a sequence of inputs, for example steering angle and throttle inputs, that would lead us from start to end config. while optimizing a cost function relative to the control inputs, such as minimizing the gas consumption, and ~~something~~ relative to the configuration of the car, such as staying at a distance from other vehicles.
- A lot of nice ways to do that. Most of them are based on numerical optimization methods.
- However, it is hard to incorporate all the constraints related to other vehicles in a good enough way in order for these algorithms to work fast.

Sampling-based Methods

- very popular, because they require a somewhat easier way to compute the definition of the Free Space

→

- Sampling-based Methods use a collision detection module that probes the free space to see if a config. has a collision or not.
- Unlike combinatorial or optical control methods which analyze the whole environment in sampling-based methods, not all parts of the free space need to be explored to find a solution.
- Explored parts are stored in a graph structure, that can be searched with a graph search Algorithm like Dijekstra or A*

- Two main classes of Sampling-based methods:
  - discrete methods: rely on a finite set of configurations and/or inputs, like a grid superimposed on top of config. space.
    - Deterministic graph search algorithms: A*, D*, D*-lite, Dijekstra, ARA*, etc...
  - probabilistic methods: rely on a probabilistic sample of a continuous configuration space.

The set of pasible configs or states that will be explored is potentially infinite → some of these methods have the nice property of being probabilistically complete, and sometimes probabilistically optimum: they will always find a solution if we allow them enough computation time.

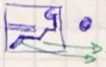
- Random explorations of the continuous configuration and the input space
- probabilistic graph search algorithms
  RRT, RRT*, PRM, etc...

A* reminder:

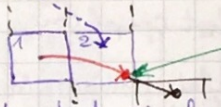
- A discrete method; needs a discrete set of states to represent the world
- needs an optimistic heuristic function to guide grid cell expansion: it should underestimate the cost to go from a cell to the goal
- Always finds a solution: is complete
- Solutions found are always optimal w.r.t. discretization of the environment
  - the more granular the discretization, the better the solution:
    Resolution Optimal

- Hybrid A*

The robot should have dificulty executing paths output by A*, e.g. because of very sharp, disruptive turns.



- A* is discrete, whereas the world for motion planning is continuous
  - => we need a version of A* that can deal with the continuous nature and give us provably executable paths
    - This is one of the big hard questions in robot motion planning!
- the key to Hybrid A* is the state transition function
  - imagine we use a sequence of very small simulations using continuous math



$$\begin{cases} x' = x + v\,\Delta t \cos\theta \\ y' = y + v\,\Delta t \sin\theta \\ \theta' = \theta + w\,\Delta t \end{cases}$$

instead of simply assigning the new cell to expand, the Hybrid A* memorizes the exact x', y', and θ' and associate it to the grid cell 2, the first time it is expanded.
Expanding the cell 2, the algorithm then uses that x', y', θ' as the starting point.
It is now possible that from expanding other cells, we end up in another x', y', θ' point (the dashed blue arrow), but since in A* we take the cells along the shortest path for expanding, before we look for longer paths we would only consider one of the x', y', θ'.

→ This leads to lack of completeness → there might be solutions for navigation problem that we don't capture

→ But it does give us correctness → as long as our motion equations are correct, the resulting path can be executed.

→ Our paths that come out are nice smoth and curved paths

→ Every time we expand a grid cell we memorize explicitly the continuous value of x', y', and θ' associated with that cell.