Localization with Monte-Carlo

Real-Time Embedded System - The F1tenth autonomous racing

UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

High Performance Real Time Lab

Course outline

- Intro course + basics of AD
- Hardware platform
- ROS2: Installation and profiling
  - Ex: ROS2 to HiL, open a bag
- Navigation: FTG, FTW, Pure pursuit
  - EX: navigation HiL
- Perception: scan matching, PF, LIO?
  - Ex: perception (PF with PThreads)
- Build the car

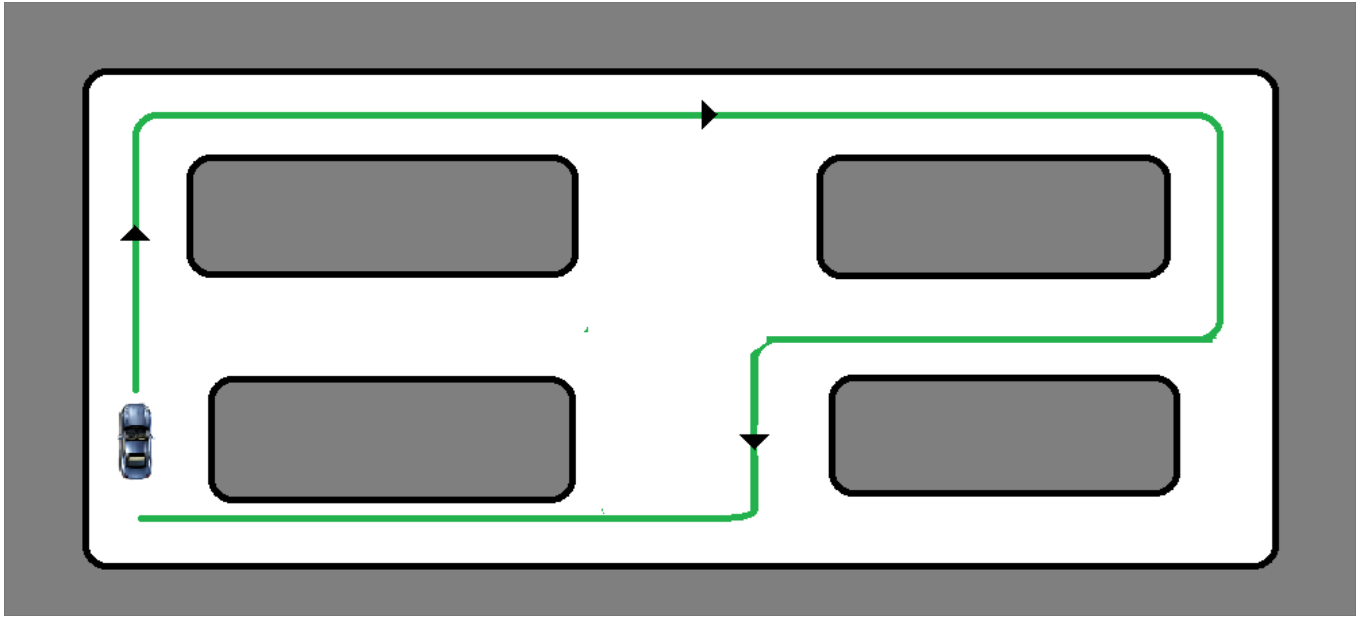
I do <u>not</u> cover all aspects of AD!!!

- Systems and control theory => Prof. Falcone
- Platforms and algorithms for autonomous systems => Prof. Sanudo & Prof. Falcone
- High-Performance Computing => Prof. Marongiu (FIM)
- Machine Learning => Cucchiara's

Why do we need maps?

Defining path

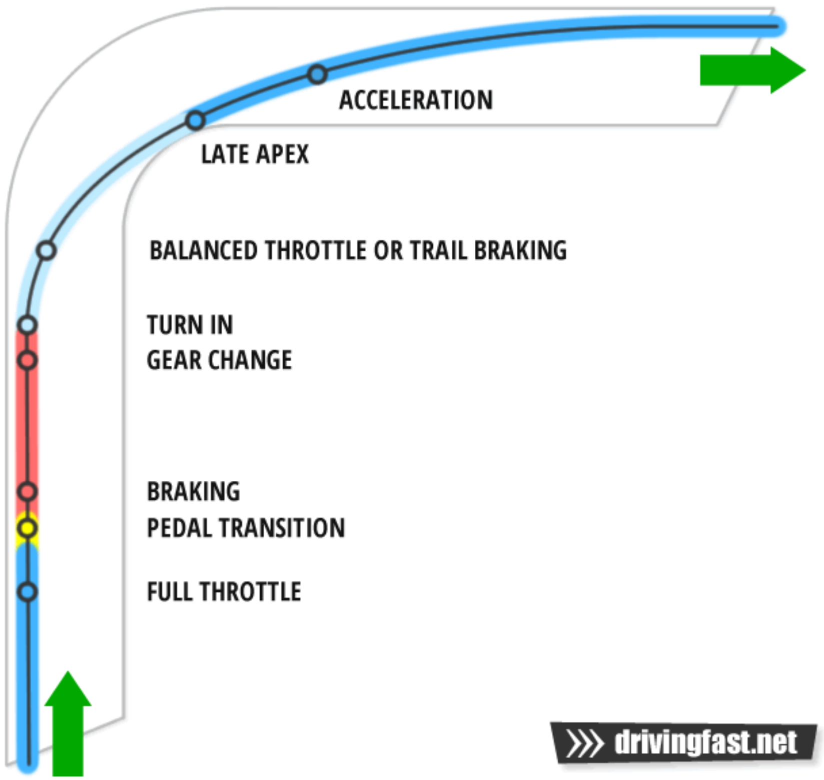
→ 2nd right, 2nd right, 1st right, 1st left, 1st right



Why do we need maps?

Compute racing lines

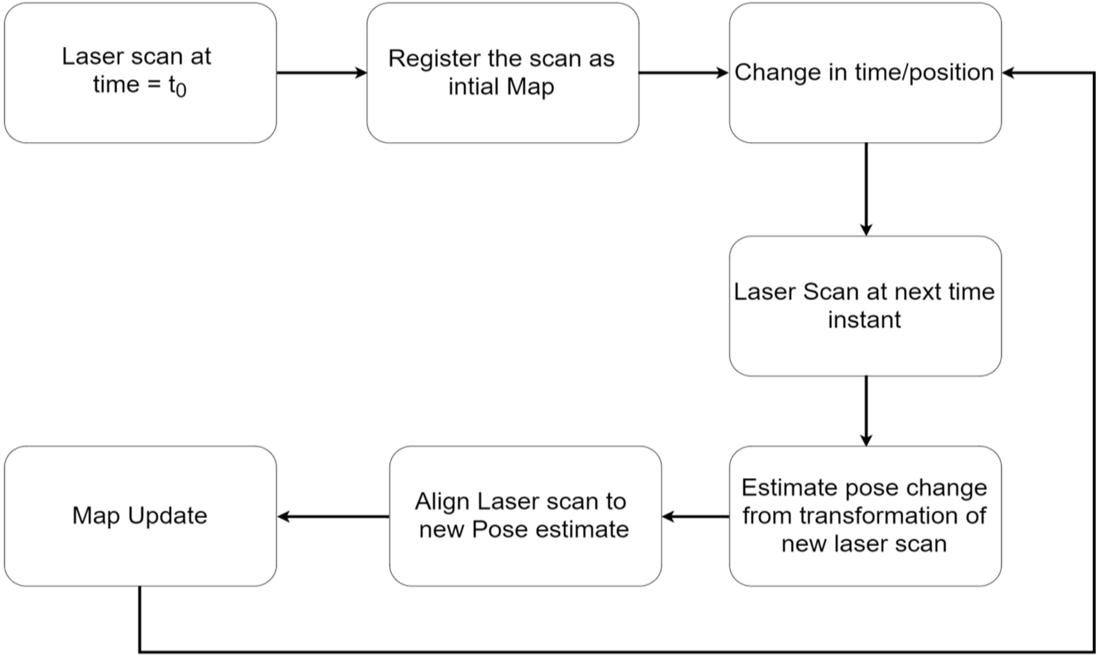
- Precise acceleration/braking/turn points
- Typically, a LUT



Simultaneous Location And Mapping

Task: Build the map and Localize

- Can be treated as two different problems…os as one (SLAM)!
- Is SLAM really needed?
- (In racing..no..)



Which sensors to use?

LiDAR is the most precise for distance

- All known limitations in cost, fragility…

Cameras are less precise

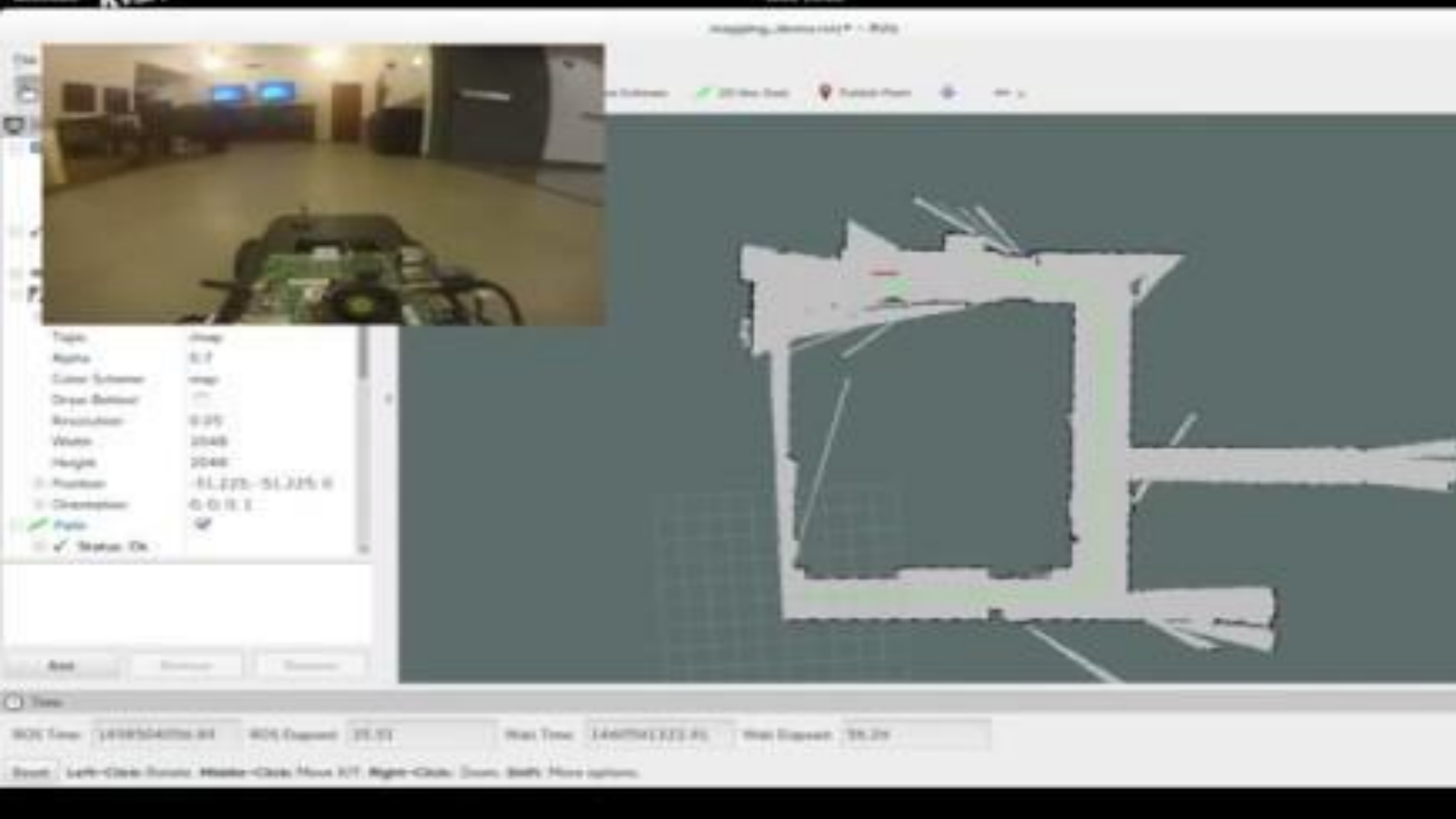
- ..but cheaper and robust
- Feature-based localization

Mix different sensors (**fuse** them)

SLAM : A Chicken-Egg problem

MAP → LOCALIZATION

LOCALIZATION → MAP

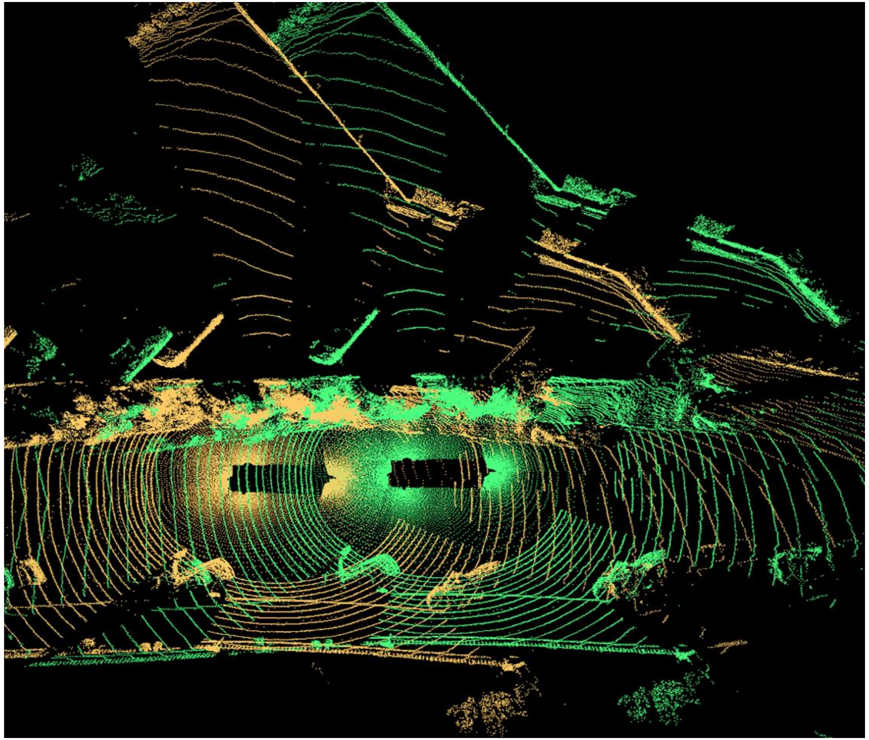


Localization: Scan Matching

Challenge:
Where is the robot with respect to the previous frame

Learning Outcome:
Iterative closest point algorithm, implementing a real research paper

Assignment:
Scan matching using iterative closest point in the simulator

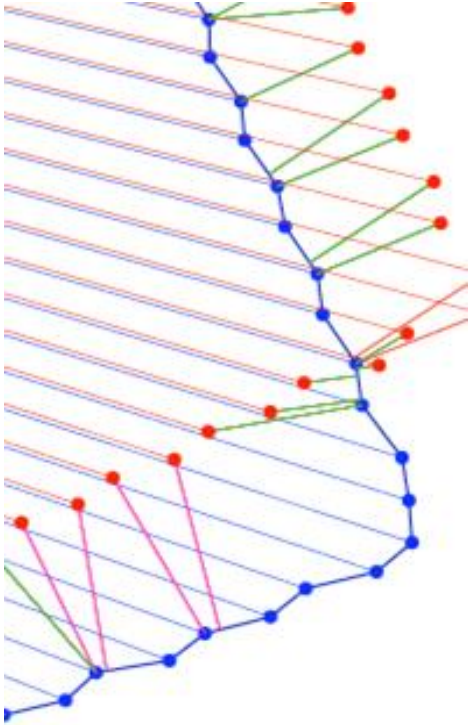


Localization: Scan Matching

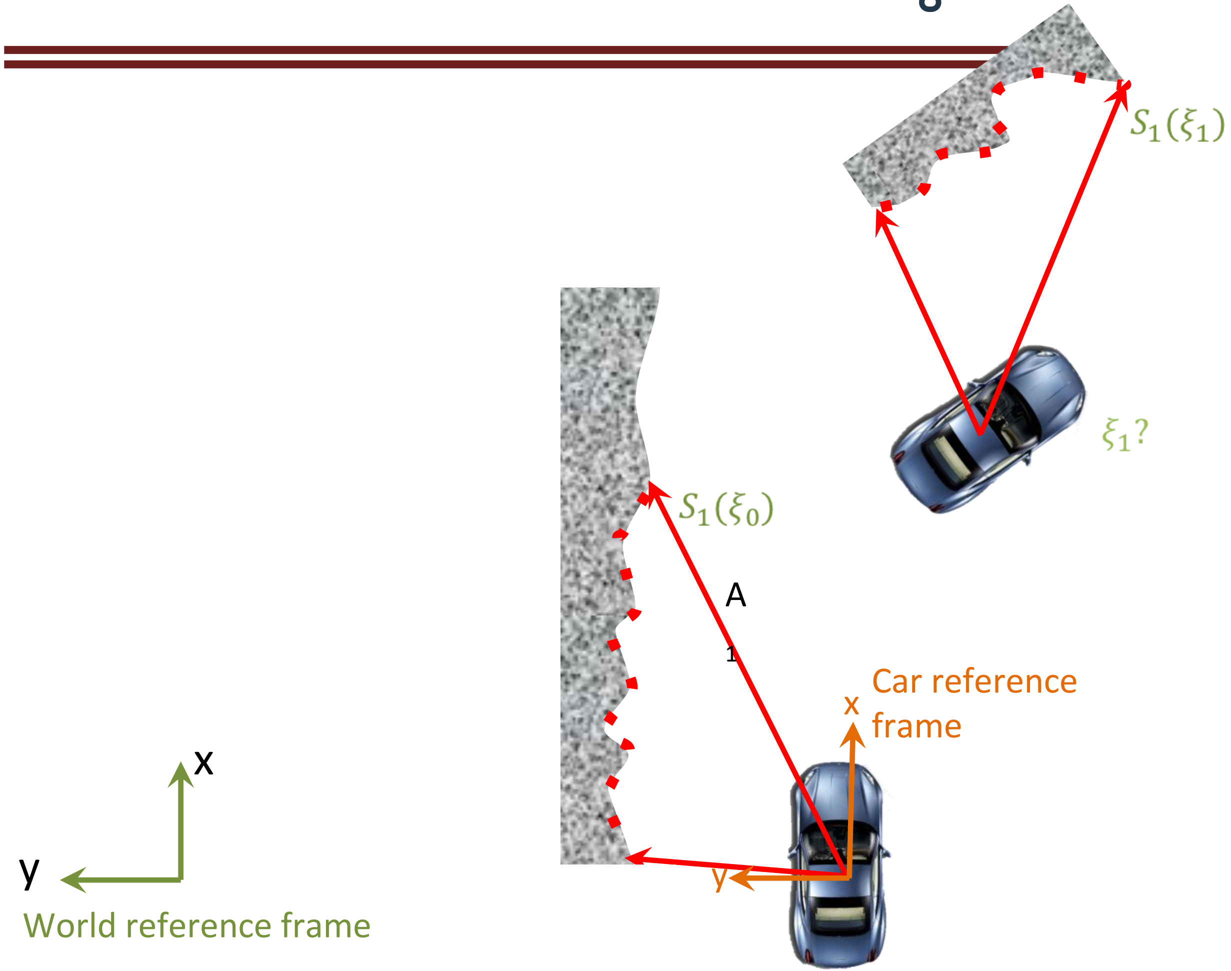
Scan matching is a fundamental localization algorithm, and is used in most of the modern SLAM algorithms.

Iterative closest point algorithm

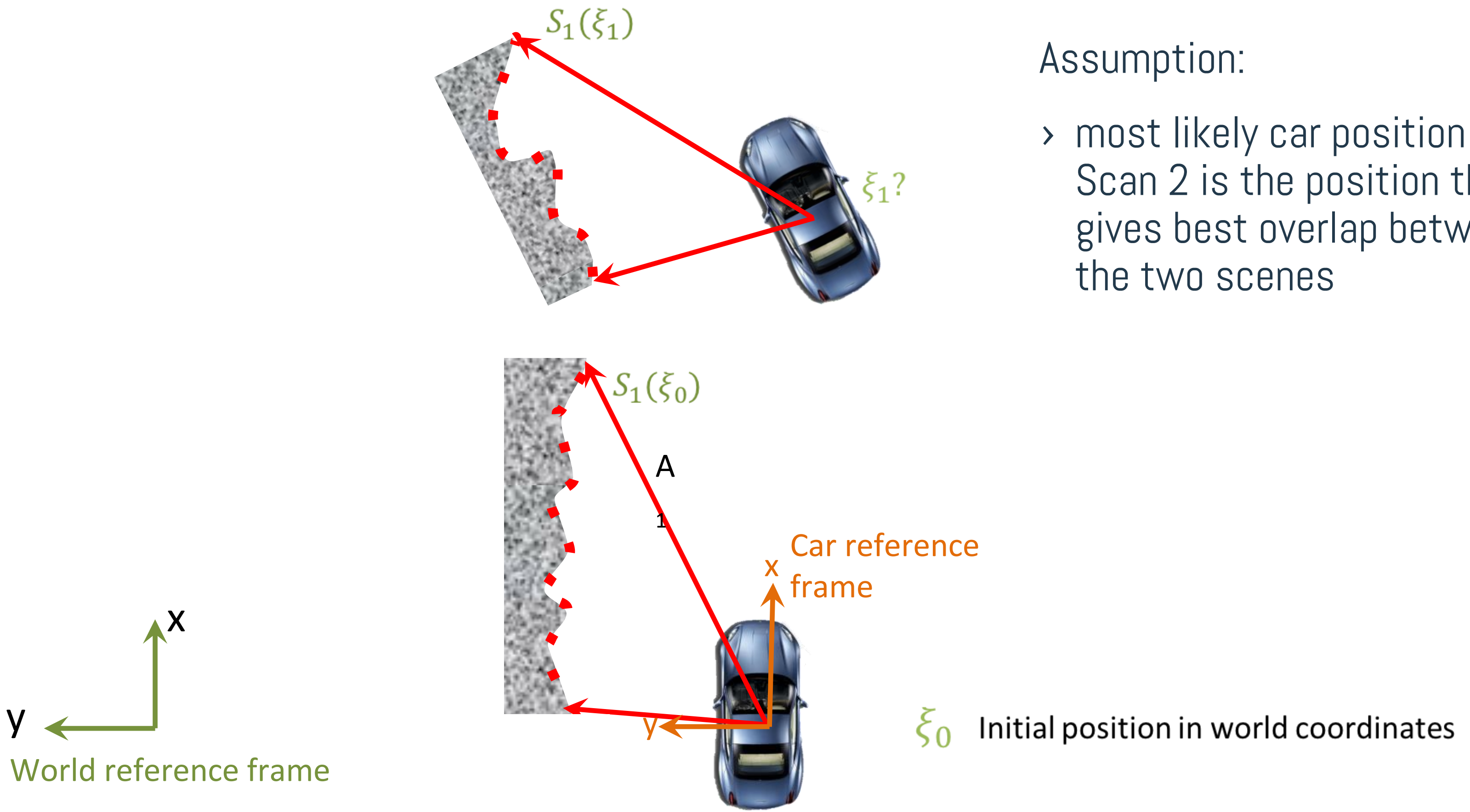
Highly sensitive to noise



Scan matching: intuition



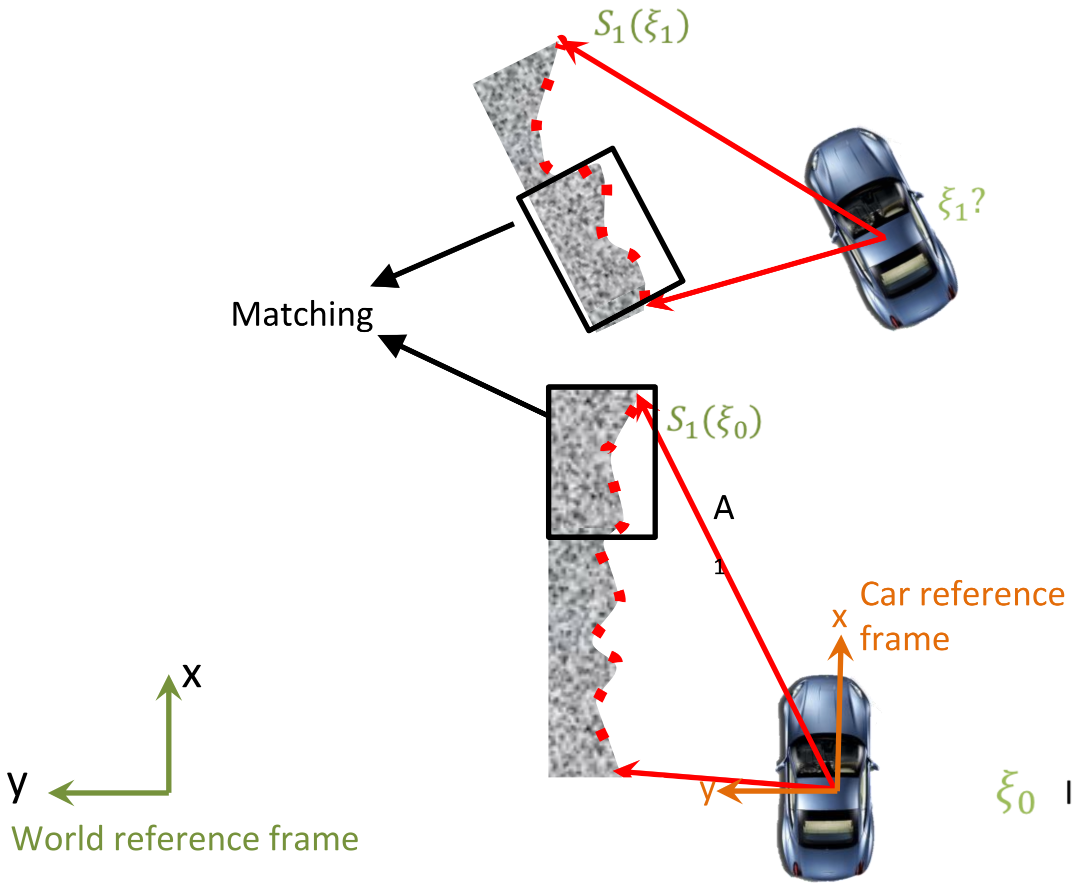
Scan matching: intuition



Assumption:

- most likely car position at Scan 2 is the position that gives best overlap between the two scenes

Scan matching: intuition

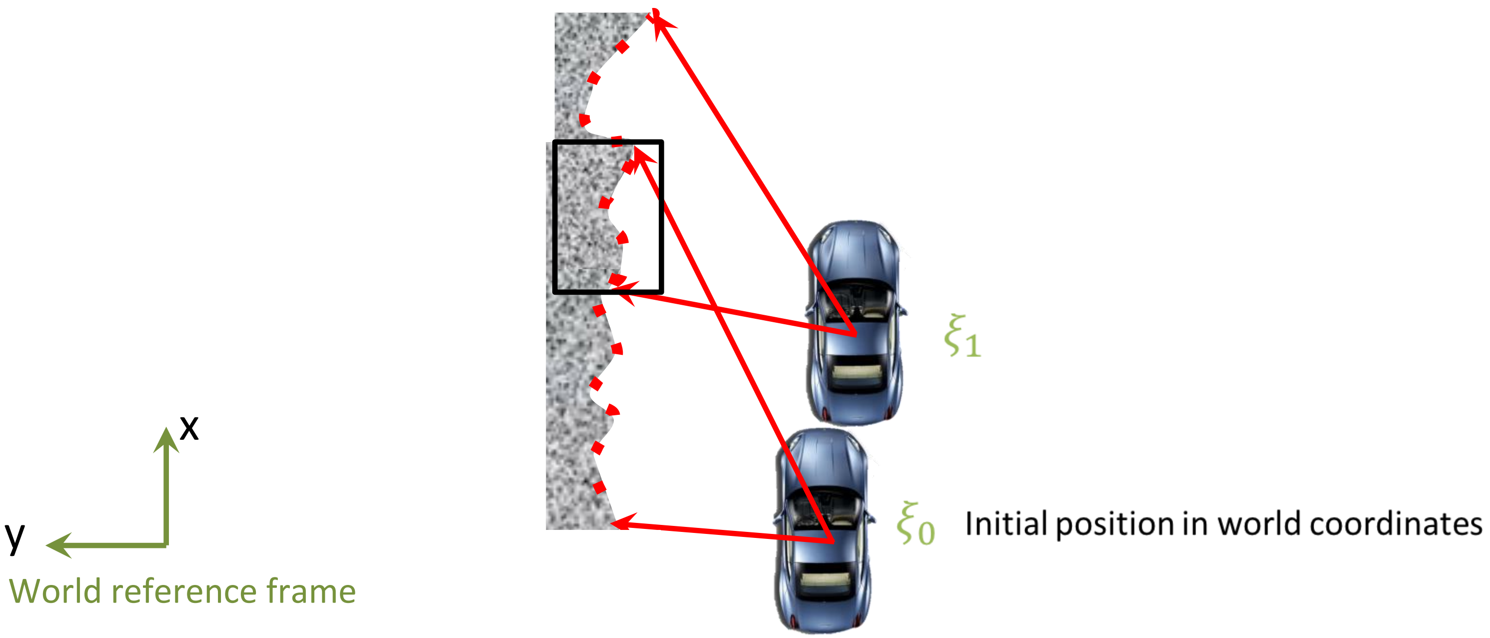


ξ_0 Initial position in world coordinates

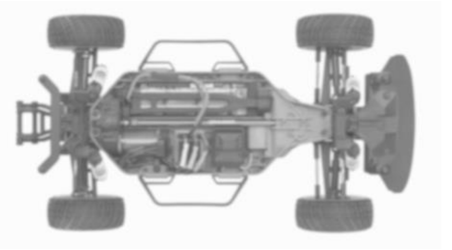
Assumption:

- most likely car position at Scan 2 is the position that gives best overlap between the two scenes

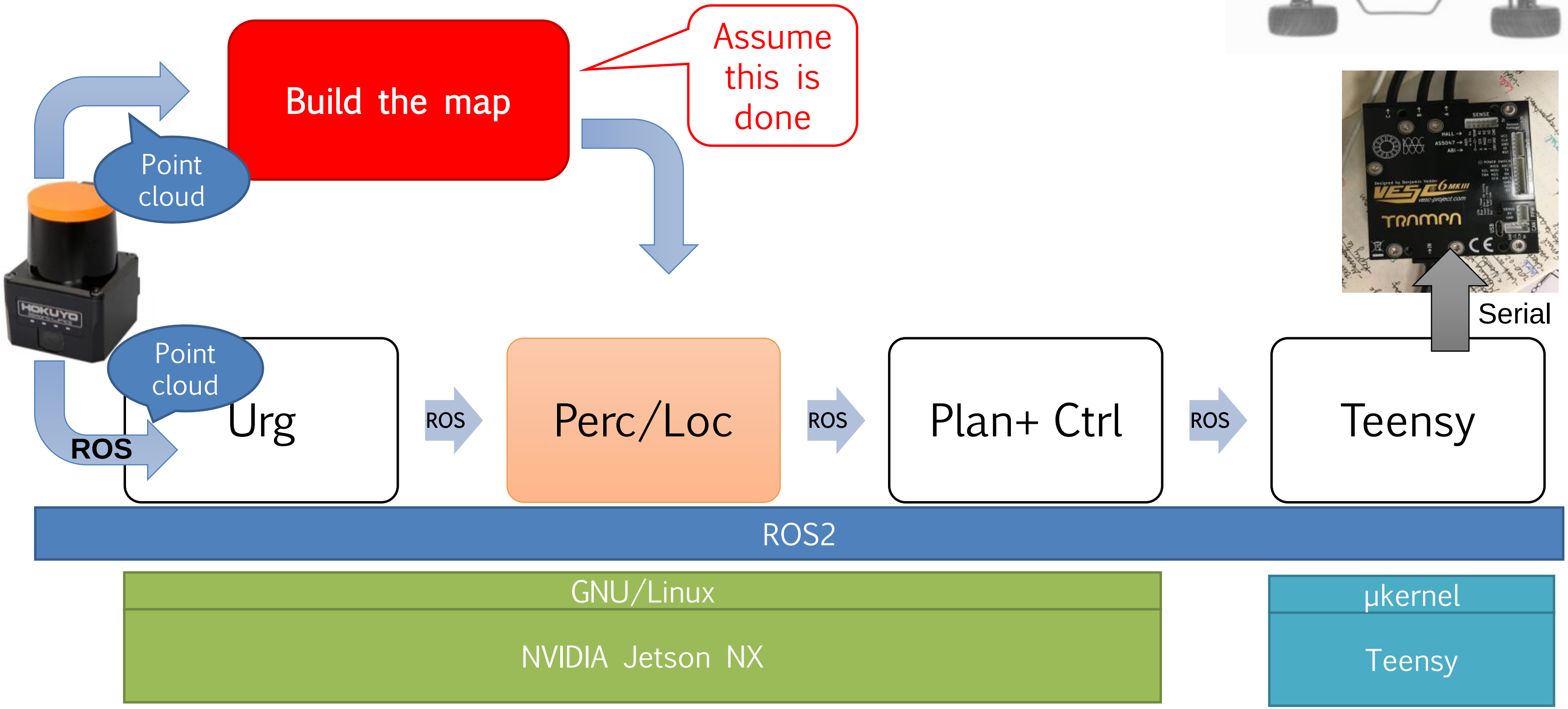
Scan matching: intuition



Localization flow



We don't employ SLAM, and we use a fast MC method, more robust to noise



Monte-Carlo methods

- Random-based experiments

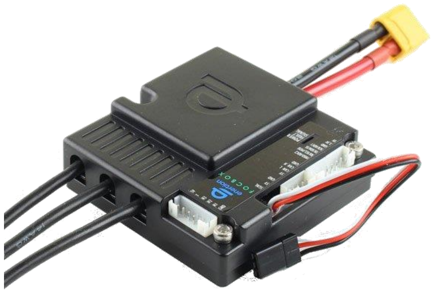
Used in

- Solving deterministic problems (e.g., π computation)
- Studying random systems

Sensors

LiDAR

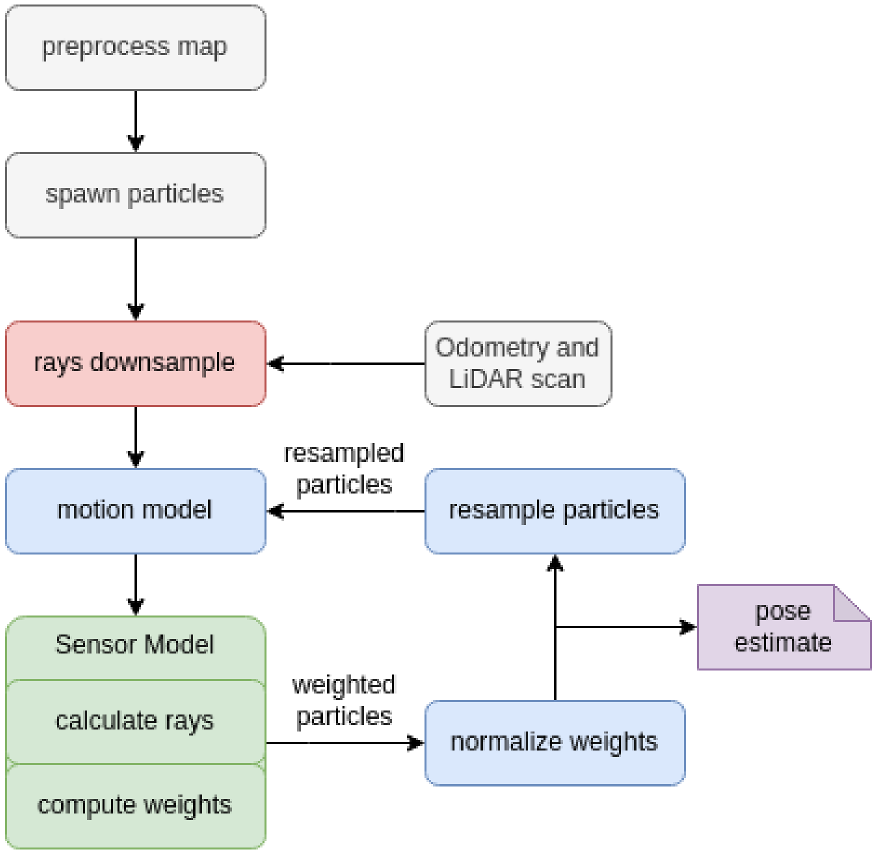
- 270° FoV
- 10m range
- High accuracy



Odometry

- Engine speed
- Steering angle
- Low accuracy

Particle Filter



1) Initialization

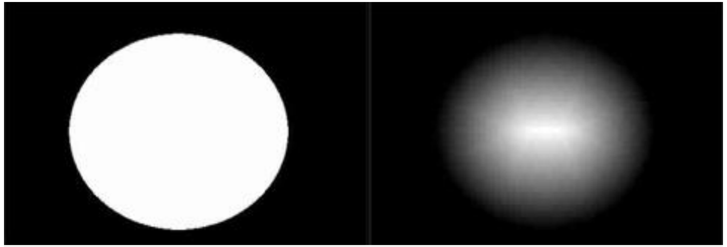
Map preprocessing

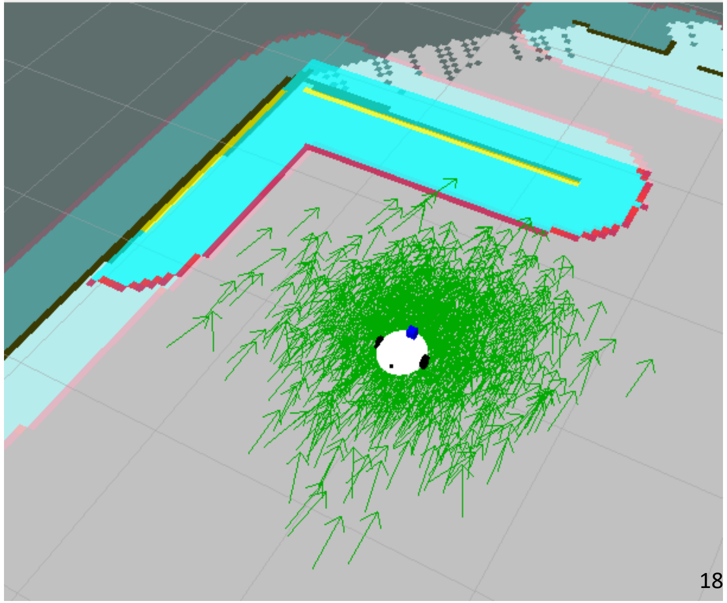
- Build a distance map
- By querying we can get the nearest wall distance given a set of coordinates (x, y)
- Used to compute the **weight** of every ray => particle

Precompute sensor model

Spawn (random) particles

- We spawn the particles at a given initial pose
- We spread them around with a gaussian distribution





2) Scan downsampling

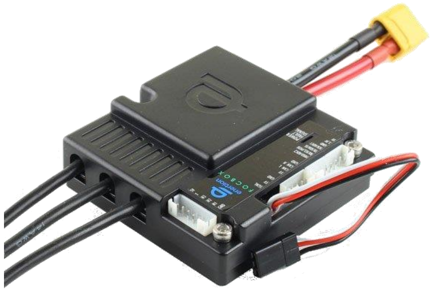
Each LiDAR scan has a lot of redundant information

- With our LiDAR we have 1080 rays
- Processing each ray is expensive and practically useless
- We initially reduce the lidar scan size by culling the out-of-range rays
- We then linearly downsample them into a fixed size

3) Motion model

Moves the particles to the estimate position

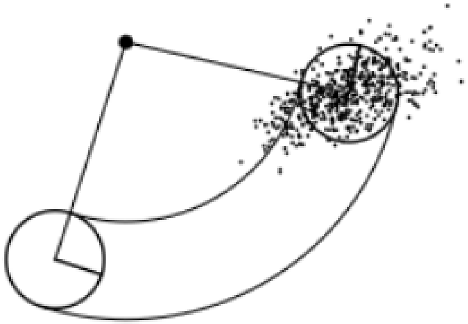
- Ackermann model
- Using odometry data (in our case, by VESC)
- Adds noise (to represent noise)



Culls the particles outside the map

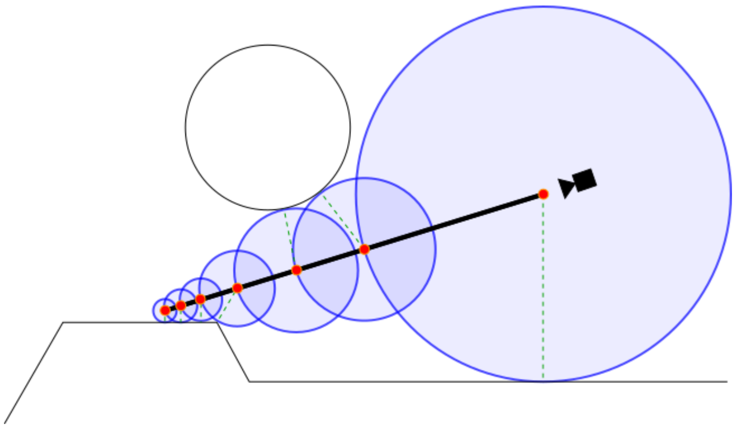
This tells us the general area where the vehicle might be located

- initialized manually at race start



Issue in simulator: odometry is too precise!!

4) Sensor model (particles weighting)



Calculate Rays

- Calculates a simulated scan for each particle
- We use ray marching for this task
- Highly parallel

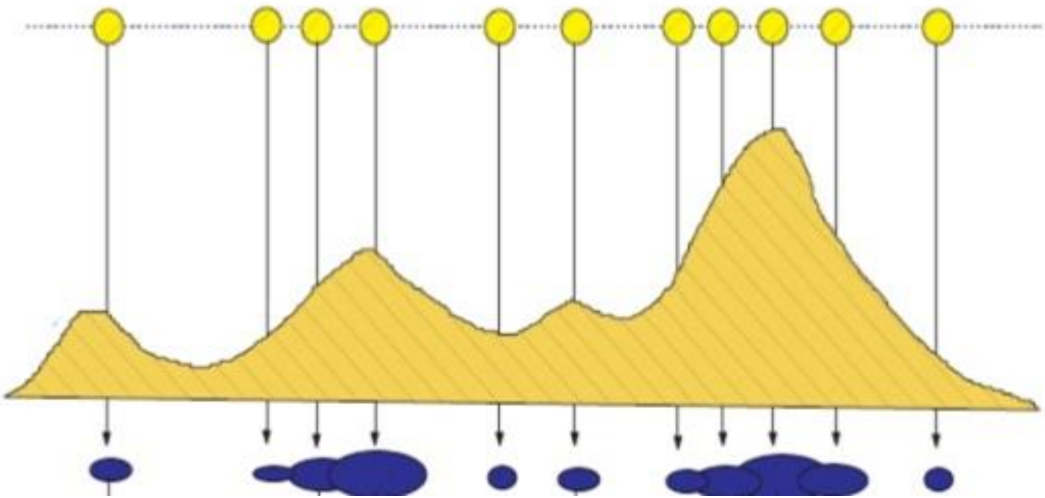
Compute Weights

- Scores each particle based on the simulated scan similarity
- Look-up table

Normalize

- Normalize particle weights
- Weight sum is now 1
- We can easily estimate the pose from here

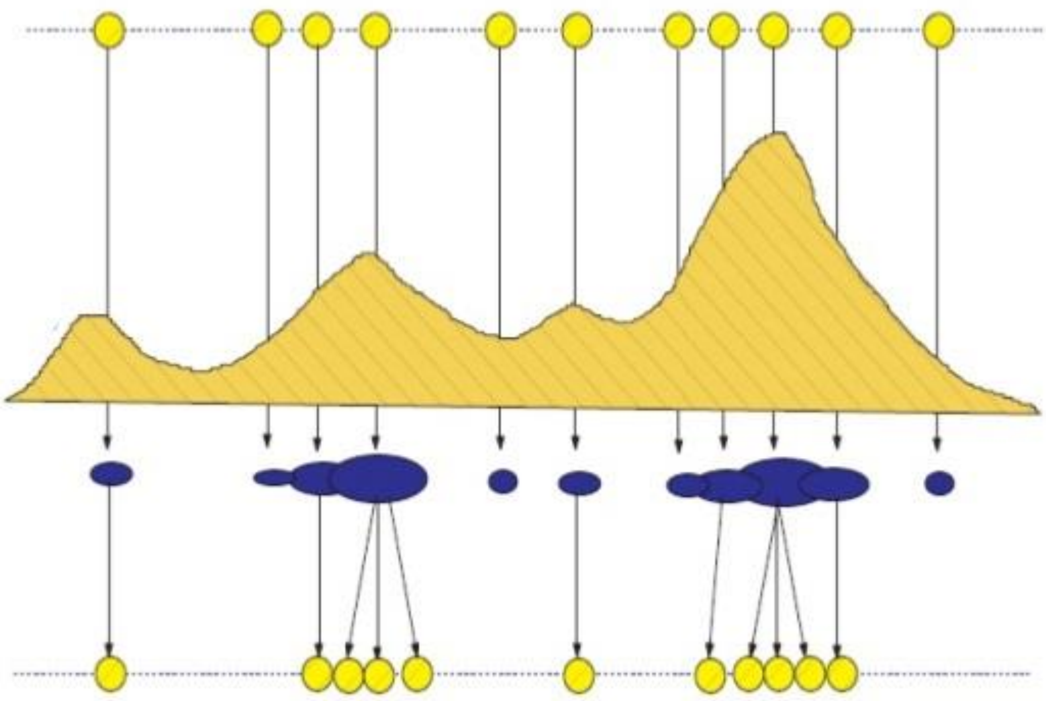
Issue with motion model



Original Particles

After N iterations

Issue with motion model



Original Particles

After N iterations

Resampling

5) Resampling

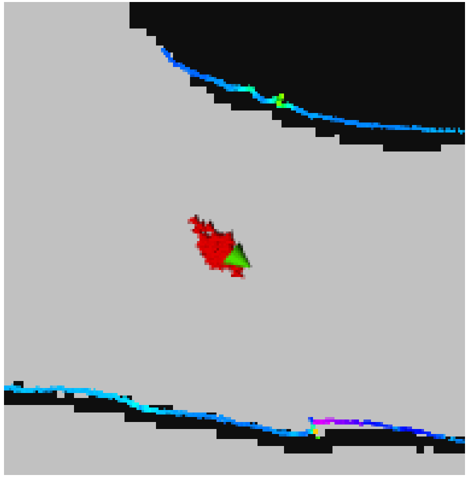
We respawn the particles around the estimated pose

› Using a weighted uniform distribution

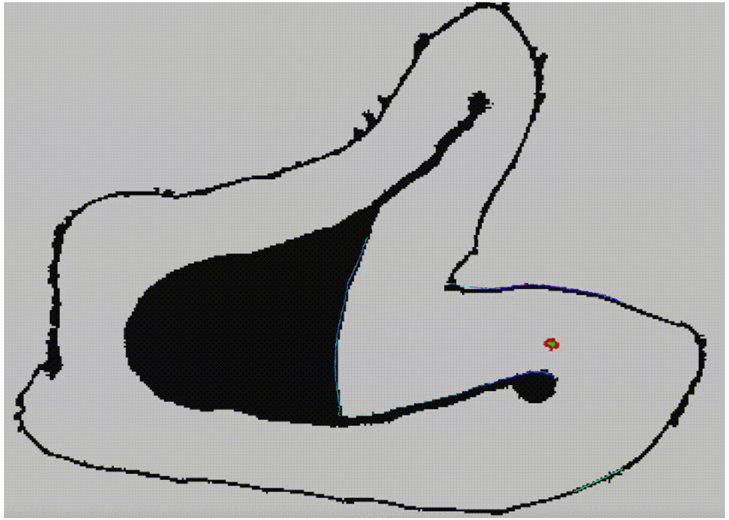
This step is useful to

› Eliminate outliers generated during the processing

› Avoid divergence by moving the particles closer to the estimated pose



Parameters



Accuracy/performance parameters

- Number of particles
- Number of downsampled rays
- Basically, reduce the computational load

Accuracy/divergence parameters

- Motion model noise distribution
  - If you spread the particles too much you reduce the overall accuracy
  - If you don't spread them enough, the particles may not model the odometry correctly

Parallelization

Let's code!

Ray marching is embarassingly parallel!

- Particle computation is data-parallel
- Rays computation is data-parallel

Can parallelize it with PThreads

- Find our code in `Code/particle_filter` folder from the course website
- Branch `refactor`!!!
- Also, acceleration with CUDA (GPUs) and FPGA (HW accelerators)

Multi-threading optimizations (possible project)

More particles/rays

- More precision
- Require more threads

More threads

- Faster (scale out)
- Can compute more particles (scale up)
- Less resources for other applications

Solution: dynamic thread adjustement!

References

Course website

- http://personale.unimore.it/rubrica/contenutiad/markober/2023/71846/N0/N0/10005
- https://github.com/HiPeRT/F1tenth-RTES
  - Online resources/preview

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/

Resources

- https://f1tenth.org
- A "small blog"
  - http://www.google.com