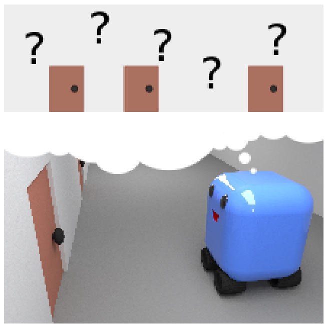
WIKIPEDIA

# Monte Carlo localization

**Monte Carlo localization (MCL)**, also known as **particle filter localization**,[1] is an algorithm for robots to localize using a particle filter.[2][3][4][5] Given a map of the environment, the algorithm estimates the position and orientation of a robot as it moves and senses the environment.[4] The algorithm uses a particle filter to represent the distribution of likely states, with each particle representing a possible state, i.e., a hypothesis of where the robot is.[4] The algorithm typically starts with a uniform random distribution of particles over the configuration space, meaning the robot has no information about where it is and assumes it is equally likely to be at any point in space.[4] Whenever the robot moves, it shifts the particles to predict its new state after the movement. Whenever the robot senses something, the particles are resampled based on recursive Bayesian estimation, i.e., how well the actual sensed data correlate with the predicted state. Ultimately, the particles should converge towards the actual position of the robot.[4]



A robot in a one-dimensional corridor containing doors. The goal of Monte Carlo localization is to let a robot determine its position based on its sensor observations.

## Contents

## Basic description

Consider a robot with an internal map of its environment. When the robot moves around, it needs to know where it is within this map. Determining its location and rotation (more generally, the pose) by using its sensor observations is known as robot localization.

Because the robot may not always behave in a perfectly predictable way, it generates many random guesses of where it is going to be next. These guesses are known as particles. Each particle contains a full description of a possible future state. When the robot observes the environment, it discards particles inconsistent with this observation, and generates more particles close to those that appear consistent. In the end, hopefully most particles converge to where the robot actually is.

## State representation

The state of the robot depends on the application and design. For example, the state of a typical 2D robot may consist of a tuple $(x, y, \theta)$ for position $x, y$ and orientation $\theta$. For a robotic arm with 10 joints, it may be a tuple containing the angle at each joint: $(\theta_1, \theta_2, \ldots, \theta_{10})$.

The *belief*, which is the robot's estimate of its current state, is a probability density function distributed over the state space.[1][4] In the MCL algorithm, the belief at a time $t$ is represented by a set of $M$ particles $X_t = \{x_t^{[1]}, x_t^{[2]}, \ldots, x_t^{[M]}\}$.[4] Each particle contains a state, and can thus be considered a hypothesis of the robot's state. Regions in the state space with many particles correspond to a greater probability that the robot will be there—and regions with few particles are unlikely to be where the robot is.

The algorithm assumes the Markov property that the current state's probability distribution depends only on the previous state (and not any ones before that), i.e., $X_t$ depends *only* on $X_{t-1}$.[4] This only works if the environment is static and does not change with time.[4] Typically, on start up, the robot has no information on its current pose so the particles are uniformly distributed over the configuration space.[4]

## Overview

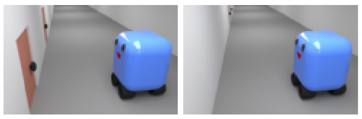
Given a map of the environment, the goal of the algorithm is for the robot to determine its pose within the environment.

At every time $t$ the algorithm takes as input the previous belief $X_{t-1} = \{x_{t-1}^{[1]}, x_{t-1}^{[2]}, \ldots, x_{t-1}^{[M]}\}$, an actuation command $u_t$, and data received from sensors $z_t$; and the algorithm outputs the new belief $X_t$.[4]

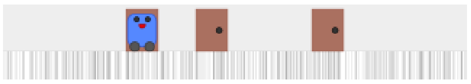
```
Algorithm MCL($X_{t-1}, u_t, z_t$):
    $\bar{X}_t = X_t = \emptyset$
    for $m = 1$ to $M$:
        $x_t^{[m]} =$ motion_update($u_t, x_{t-1}^{[m]}$)
        $w_t^{[m]} =$ sensor_update($z_t, x_t^{[m]}$)
        $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
    endfor
    for $m = 1$ to $M$:
        draw $x_t^{[m]}$ from $\bar{X}_t$ with probability $\propto w_t^{[m]}$
        $X_t = X_t + x_t^{[m]}$
    endfor
    return $X_t$
```

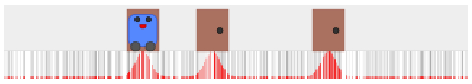
### Example for 1D robot

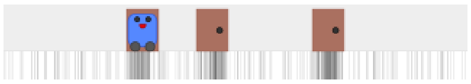
Consider a robot in a one-dimensional circular corridor with three identical doors, using a sensor that returns either true or false depending on whether there is a door.


A robot travels along a one-dimensional corridor, armed with a sensor that can only tell if there is a door (left) or there is no door (right).

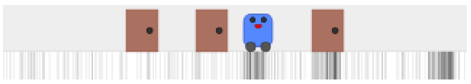
$t = 0$

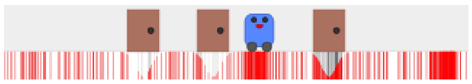

The algorithm initializes with a uniform distribution of particles. The robot considers itself equally likely to be at any point in space along the corridor, even though it is physically at the first door.

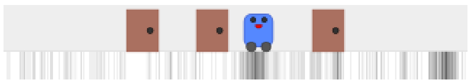

**Sensor update**: the robot detects **a door**. It assigns a weight to each of the particles. The particles which are likely to give this sensor reading receive a higher weight.


**Resampling**: the robot generates a set of new particles, with most of them generated around the previous particles with more weight. It now believes it is at one of the three doors.

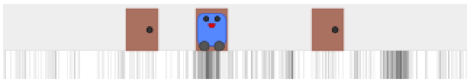
$t = 1$

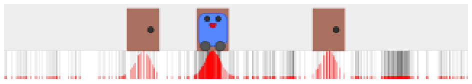

**Motion update**: the robot moves some distance to the right. All particles also move right, and some noise is applied. The robot is physically between the second and third doors.

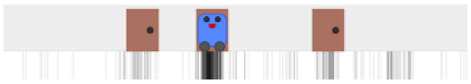

**Sensor update**: the robot detects **no door**. It assigns a weight to each of the particles. The particles likely to give this sensor reading receive a higher weight.


**Resampling**: the robot generates a set of new particles, with most of them generated around the previous particles with more weight. It now believes it is at one of two locations.

$t = 2$


**Motion update**: the robot moves some distance to the left. All particles also move left, and some noise is applied. The robot is physically at the second door.

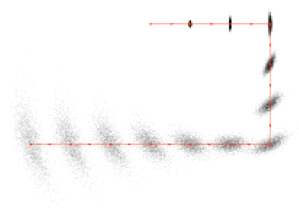

**Sensor update**: the robot detects **a door**. It assigns a weight to each of the particles. The particles likely to give this sensor reading receive a higher weight.


**Resampling**: the robot generates a set of new particles, with most of them generated around the previous particles with more weight. The robot has successfully localized itself.

At the end of the three iterations, most of the particles are converged on the actual position of the robot as desired.

## Motion update

During the motion update, the robot predicts its new location based on the actuation command given, by applying the simulated motion to each of the particles.[1] For example, if a robot moves forward, all particles move forward in their own directions no matter which way they point. If a robot rotates 90 degrees clockwise, all particles rotate 90 degrees clockwise, regardless of where they are. However, in the real world, no actuator is perfect: they may overshoot or undershoot the desired amount of motion. When a robot tries to drive in a straight line, it inevitably curves to one side or the other due to minute differences in wheel radius.[1] Hence, the motion model must compensate for noise. Inevitably, the particles diverge during the motion update as a consequence. This is expected since a robot becomes less sure of its position if it moves blindly without sensing the environment.


Belief after moving several steps for a 2D robot using a typical motion model without sensing.

## Sensor update

When the robot senses its environment, it updates its particles to more accurately reflect where it is. For each particle, the robot computes the probability that, had it been at the state of the particle, it would perceive what its sensors have actually sensed. It assigns a weight $w_t^{[i]}$ for each particle proportional to the said probability. Then, it randomly draws $M$ new particles from the previous belief, with probability proportional to $w_t^{[i]}$. Particles consistent with sensor readings are more likely to be chosen (possibly more than once) and particles inconsistent with sensor readings are rarely picked. As such, particles converge towards a better estimate of the robot's state. This is expected since a robot becomes increasingly sure of its position as it senses its environment.

## Properties

### Non-parametricity

The particle filter central to MCL can approximate multiple different kinds of probability distributions, since it is a non-parametric representation.[4] Some other Bayesian localization algorithms, such as the Kalman filter (and variants, the extended Kalman filter and the unscented Kalman filter), assume the belief of the robot is close to being a Gaussian distribution and do not perform well for situations where the belief is multimodal.[4] For example, a robot in a long corridor with many similar-looking doors may arrive at a belief that has a peak for each door, but the robot is unable to distinguish *which* door it is at. In such situations, the particle filter can give better performance than parametric filters.[4]

Another non-parametric approach to Markov localization is the grid-based localization, which uses a histogram to represent the belief distribution. Compared with the grid-based approach, the Monte Carlo localization is more accurate because the state represented in samples is not discretized.[2]

### Computational requirements

The particle filter's time complexity is linear with respect to the number of particles. Naturally, the more particles, the better the accuracy, so there is a compromise between speed and accuracy and it is desired to find an optimal value of $M$. One strategy to select $M$ is to continuously generate additional particles until the next pair of command $u_t$ and sensor reading $z_t$ has arrived.[4] This way, the greatest possible number of particles is obtained while not impeding the function of the rest of the robot. As such, the implementation is adaptive to available computational resources: the faster the processor, the more particles can be generated and therefore the more accurate the algorithm is.[4]

Compared to grid-based Markov localization, Monte Carlo localization has reduced memory usage since memory usage only depends on number of particles and does not scale with size of the map,[2] and can integrate measurements at a much higher frequency.[2]

The algorithm can be improved using KLD sampling, as described below, which adapts the number of particles to use based on how sure the robot is of its position.

### Particle deprivation

A drawback of the naive implementation of Monte Carlo localization occurs in a scenario where a robot sits at one spot and repeatedly senses the environment without moving.[4] Suppose that the particles all converge towards an erroneous state, or if an occult hand picks up the robot and moves it to a new location after particles have already converged. As particles far away from the converged state are rarely selected for the next iteration, they become scarcer on each iteration until they disappear altogether. At this point, the algorithm is unable to recover.[4] This problem is more likely to occur for small number of particles, e.g., $M \leq 50$, and when the particles are spread over a large state space.[4] In fact, any particle filter algorithm may accidentally discard all particles near the correct state during the resampling step.[4]

One way to mitigate this issue is to randomly add extra particles on every iteration.[4] This is equivalent to assuming that, at any point in time, the robot has some small probability of being kidnapped to a random position in the map, thus causing a fraction of random states in the motion model.[4] By guaranteeing that no area in the map is totally deprived of particles, the algorithm is now robust against particle deprivation.

## Variants

The original Monte Carlo localization algorithm is fairly simple. Several variants of the algorithm have been proposed, which address its shortcomings or adapt it to be more effective in certain situations.

### KLD sampling

Monte Carlo localization may be improved by sampling the particles in an adaptive manner based on an error estimate using the Kullback–Leibler divergence (KLD). Initially, it is necessary to use a large $M$ due to the need to cover the entire map with a uniformly random distribution of particles. However, when the particles have converged around the same location, maintaining such a large sample size is computationally wasteful. [6]

KLD–sampling is a variant of Monte Carlo Localization where at each iteration, a sample size $M_x$ is calculated. The sample size $M_x$ is calculated such that, with probability $1 - \delta$, the error between the true posterior and the sample-based approximation is less than $\epsilon$. The variables $\delta$ and $\epsilon$ are fixed parameters.[4]

The main idea is to create a grid (a histogram) overlaid on the state space. Each bin in the histogram is initially empty. At each iteration, a new particle is drawn from the previous (weighted) particle set with probability proportional to its weight. Instead of the resampling done in classic MCL, the KLD–sampling algorithm draws particles from the previous, weighted, particle set and applies the motion and sensor updates before placing the particle into its bin. The algorithm keeps track of the number of non-empty bins, $k$. If a particle is inserted in a previously empty bin, the value of $M_x$ is recalculated, which increases mostly linear in $k$. This is repeated until the sample size $M$ is the same as $M_x$. [4]

It is easy to see KLD–sampling culls redundant particles from the particle set, by only increasing $M_x$ when a new location (bin) has been filled. In practice, KLD–sampling consistently outperforms and converges faster than classic MCL.[4]

## References

1. Ioannis M. Rekleitis. "A Particle Filter Tutorial for Mobile Robot Localization." *Centre for Intelligent Machines, McGill University, Tech. Rep. TR-CIM-04-02* (2004).
2. Frank Dellaert, Dieter Fox, Wolfram Burgard, Sebastian Thrun. "Monte Carlo Localization for Mobile Robots (http://www.ri.cmu.edu/pubs/pub_533.html) Archived (https://web.archive.org/web/20070917170430/http://www.ri.cmu.edu/pubs/pub_533.html) 2007-09-17 at the Wayback Machine." *Proc. of the IEEE International Conference on Robotics and Automation* Vol. 2. IEEE, 1999.
3. Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun, "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots (http://www.cs.washington.edu/ai/Mobile_Robotics/abstracts/sampling-aaai-99.abstract.html)." *Proc. of the Sixteenth National Conference on Artificial Intelligence* John Wiley & Sons Ltd, 1999.
4. Sebastian Thrun, Wolfram Burgard, Dieter Fox. *Probabilistic Robotics* (http://www.probabilistic-robotics.org/) MIT Press, 2005. Ch. 8.3 ISBN 9780262201629.
5. Sebastian Thrun, Dieter Fox, Wolfram Burgard, Frank Dellaert. "Robust monte carlo localization for mobile robots (http://robots.stanford.edu/papers/thrun.robust-mcl.html)." *Artificial Intelligence* 128.1 (2001): 99–141.

6. Dieter Fox. "KLD–Sampling: Adaptive Particle Filters." *Department of Computer Science and Engineering, University of Washington.* NIPS, 2001.