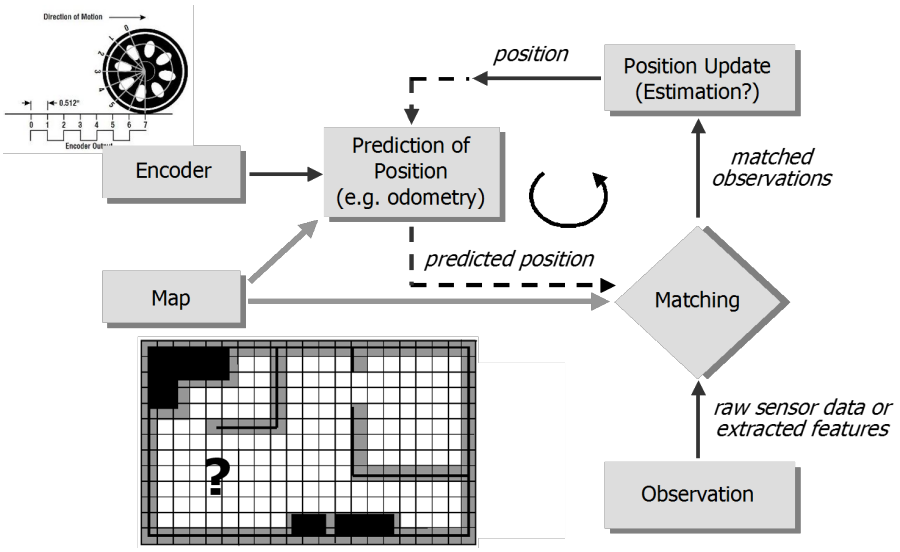


Figure 1: Robot Localization Scheme

### Reading Instructions

The objective of this lab is to implement a Monte Carlo localization with a particle filter for the Turtlebot robot.

Please carefully study the paper on Monte Carlo Localisation with a particle filter [2]. A more detailed description is provided in the book *Probabilistic Robotics* [1] or the journal article [3].

### Robot Localization

Localization is concerned with estimating the robots pose w.r.t. a map of the environment. Dead reckoning relies on odometry information to determine the relative motion of the robot. The new pose is calculated from the robot kinematics and the incremental motion of wheels recorded by encoders. Dead reckoning requires knowledge of the robot's initial pose. Its accuracy decreases over time as every measurement is subject to noise and errors accumulate due to the integration of incremental movements. Nevertheless, dead reckoning is a fundamental component of general localization methods as it provides accurate short term information.

The presence of uncertainty in both the robot's odometry and the range sensor complicates the localization task. Both pieces of information are integrated by a Bayes filter to estimate the pose as shown in figure 1.

One distinguishes between local and global localization. Local approaches compensate odometric errors accumulated during robot motion. They keep track of a robot pose under the assumption that the robot's initial pose is at least approximately known. Global techniques localize a robot without any prior assumption about its pose within the environment. Their initial belief about the robots pose is uniform across the map. These

methods are robust, in that they manage to recover from substantial position errors. They cope with the kidnapped robot problem which manually displaces the robot to some unknown location.

### Particle Filter

Monte Carlo Localization (MCL) localizes a robot by maintaining a particle filter. The algorithm relies on a map of the enviroment and estimates the robot pose within the map according to odometry and range sensor information. The algorithm represents its belief of the robot pose by a probability density function. The density function is sample based; each particle accounts for a possible hypothesis of the robot pose. Each particle contains information about the hypothesized robot pose and a weight that captures the probability of that pose. The prediction step updates the particles poses according to a robot motion model based on the odometry information obtained from the wheel encoders or commanded velocities as shown in figure 1. New robot poses are sampled from the motion model. The correction step updates the particle weight based on new sensor readings. The particle pose is evaluated based on the likelihood of the sensor readings at the current pose given the map. Particles are resampled according to their weights, generating more particles in regions of likely poses and eliminating unlikely hypotheses. Upon iterating prediction, correction, and resampling the cloud of particles eventually converges to a single cluster at the true pose of the robot.

Markov localization with particle filters hypothesizes at any iteration a set of possible robot poses. Each particle refers to a candidate pose as it consists of the hypothesized pose of the robot along with an importance weight. This importance weight reflects the likelihood that the position hypothesized by the particle is true. The localization algorithm is initialized by populating the state space with a set of randomly chosen particles (global localization) or initializing their pose at the known robot position (local localization). Initially, all particles possess equal weights. Algorithm 2 illustrates the pseudo code for Monte Carlo localization with a particle filter.

### Monte Carlo Localisation Algorithm Robotics System Toolbox

The Monte Carlo Localization (MCL) algorithm estimates the robot's pose given a map of the environment, range sensor readings, and odometry data. In Matlab's Robotics System Toolbox, the `robotics.MonteCarloLocalization` object and its methods provide an implementation of MCL.

The MCL algorithm estimates the robot's pose with a particle filter. The particles reflect a sample based probability density of likely robot poses. Each particle $\{x, y, \theta, w\}$ accounts for a possible robot pose $[x, y, \theta]$ w.r.t. the map frame and a weight $w$ that reflects the confidence in that pose. The particle clouds converges to a single cluster as the robot wanders around in the environment and senses novel regions and objects by its range sensor. Wheel encoders provide odometry information that captures the robot's relative motion.

The set of particles is maintained and updated through the following process:

**Algorithm 1** MCL with particle filter

**Input:** $\mathcal{X}_{t-1}$: set of particles $x$ at time $t-1$, $u_t$: actuation command, $z_t$: measurement, $N$: number of particles in a set $\mathcal{X}$

1: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$;
2: **for** $n = 1$ **to** $N$ **do**
3: $\quad x_t^{[n]} =$ **motion_model**$(u_t, x_{t-1}^{[n]})$; ▷ sample new pose from motion model
4: $\quad w_t^{[n]} =$ **sensor_model**$(z_t, x_t^{[n]})$; ▷ likelihood according to sensor model
5: $\quad \bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \left\langle x_t^{[n]}, w_t^{[n]} \right\rangle$; ▷ add particle to intermediate set
6: **for** $n = 1$ **to** $N$ **do**
7: $\quad$ draw $x_t^{[n]}$ with probability $\propto w_t^{[n]}$ from $\bar{\mathcal{X}}_t$ ; ▷ resampling
8: $\quad$ add $x_t^{[n]}$ to $\mathcal{X}_t$;
**return** $\mathcal{X}_t$;

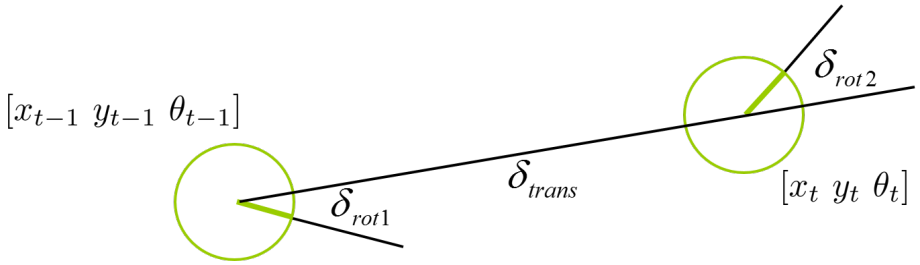


Figure 2: Elemental motions of odometry based motion model

- Particle poses $[x, y, \theta]$ are propagated based on incremental motion provided by odometry and the detailed motion model `MotionModel`.
- The particle weights $w$ are assigned based on the likelihood of receiving the range sensor reading for each particle. The `SensorModel` captures the probability of a sensor reading given the location of obstacles according to the particles pose.
- The robot pose is estimated by weighting the pose hypothesis with the particle weights.
- Finally, the particles are resampled at a rate defined by the `ResamplingInterval`. The purpose of resampling is to better approximate the underlying probability distribution with a sample set by removing particles of low height and increasing the particle density in regions of high probability. Resampling replicates high weighted particles and adjusts the number of particles according to the complexity of the distribution.

### Odometry-Based Motion Model

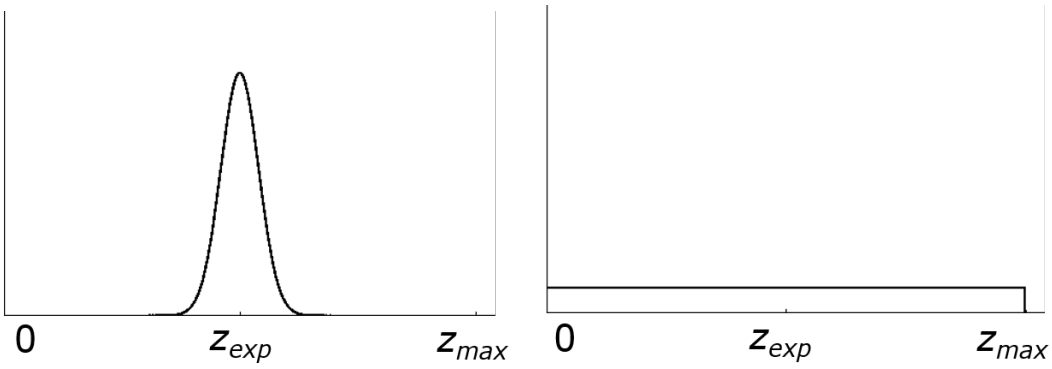


Figure 3: Probability distribution for expected obstacle and random measurement.

The odometry based motion model composes the transition from previous pose $\vec{x}_{t-1} = [x_{t-1}\ y_{t-1}\ \theta_{t-1}]$ to the current pose $\vec{x}_t = [x_t\ y_t\ \theta_t]$ into three elementary motions, a rotation by $\delta_{rot1}$ to align the initial $\theta_{t-1}$ with the heading towards the new pose, a translation by $\delta_{trans}$ followed by a final rotation $\delta_{rot2}$ to achieve the final orientation $\theta_t$ as shown in 2. Odometry provides the information on the incremental motion between consecutive poses. The relationship between consecutive poses and the three elementary motions is given by:

$$\begin{aligned} \delta_{trans} &= \sqrt{(x_t - x_{t-1})^2 + (y_t - y_{t-1})^2} \\ \delta_{rot1} &= atan2(y_t - y_{t-1}, x_t - x_{t-1}) - \theta_{t-1} \\ \delta_{rot2} &= \theta_t - \theta_{t-1} - \delta_{rot1} \end{aligned} \tag{1}$$

The motion model samples transitions by considering the hypothesized particle transition as a noisy version of the nominal transformation obtained from odometry according to equation (1). For that purpose zero mean Gaussian noise is added to $[\delta_{trans}\ \delta_{rot1}\ \delta_{rot2}]$ which variance is proportional to the magnitude of the motion itself.

$$\begin{aligned} \hat{\delta}_{trans} &= \delta_{trans} + \mathcal{N}(0, \alpha_3\delta_{trans} + \alpha_4(\delta_{rot1} + \delta_{rot2})) \\ \hat{\delta}_{rot1} &= \delta_{rot1} + \mathcal{N}(0, \alpha_2\delta_{trans} + \alpha_1(\delta_{rot1})) \\ \hat{\delta}_{rot2} &= \delta_{rot2} + \mathcal{N}(0, \alpha_2\delta_{trans} + \alpha_1(\delta_{rot2})) \end{aligned} \tag{2}$$

in which $\mathcal{N}(\mu, \sigma)$ denotes a random number sampled from a Gaussian with mean $\mu$ and variance $\sigma$. The parameters $\alpha_i$ denote the relative error due to odometry, in particular $\alpha_1$ denotes the rotational error due to rotational motion, $\alpha_2$ the rotational error due to translational motion, $\alpha_3$ the translational error due to translation motion and $\alpha_4$ the translational error due to rotational motion.

### Range Finder Sensor Model

The range finder sensor model considers the causes of range readings:

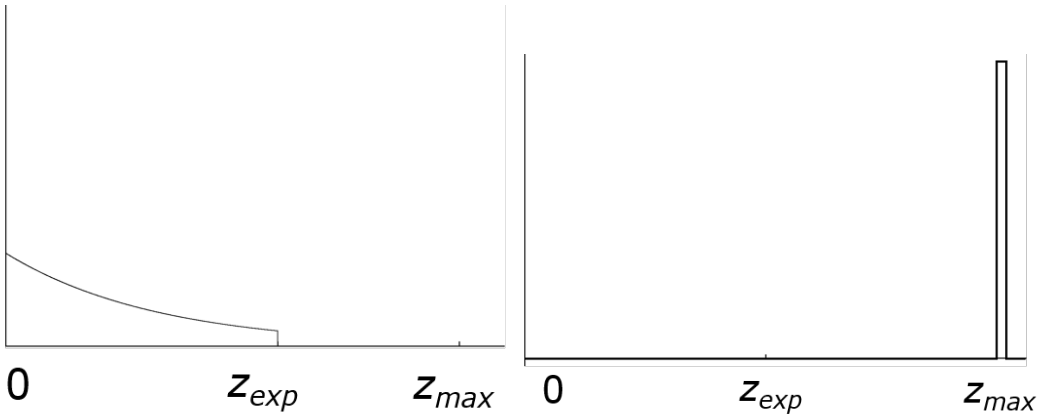


Figure 4: Probability distribution for unexpected obstacle and out of range readings.

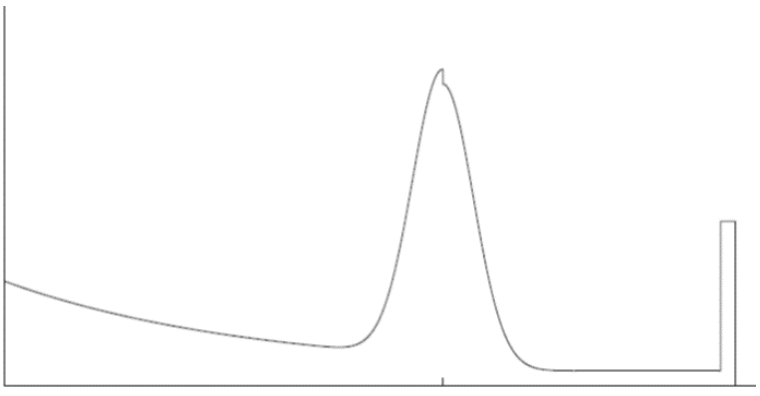

Figure 5: Mixture distribution for range readings.

- expected measurements: These are correct readings caused by obstacles in the environment that are part of the map. The reading is subject to additive noise from a Gaussian distribution. The expected reading $z_{exp}$ is obtained from ray tracing along the sensor axis for the first intersection with an obstacle. The Gaussian distribution with mean $z_{exp}$ and variance $\sigma$ is shown in figure 3 left.
- random measurements: Range sensors occasionally produce inexplicable measurements such as crosstalk in sonar sensors. These phantom readings are modeled by a uniform distribution (figure 3 right).
- unknown obstacles: Objects in the environment that are not part of the map might cause short range readings w.r.t. to the expected reading according to the map. The probability of short unexpected readings is captured by an exponential distribution cut-off at the expected obstacle distance $z_{exp}$ as shown in figure 4. The exponential distribution is characterized by a single parameter $\lambda$ which denotes the mean of the exponential.
- sensor failures: Obstacles are missed altogether, for example absorbing or opaque surfaces for the Kinect infrared sensor. These failures usually generate a maximum range measurement modeled by a point mass distribution located at $z_{max}$ as shown in figure 4.

The mixture distribution

$$\begin{aligned} p(z|z_{exp},m) &= \alpha_{hit}p_{hit}(z|z_{exp},m) + \alpha_{rnd}p_{rnd}(z|z_{exp},m) + \\ &\quad \alpha_{unexp}p_{unexp}(z|z_{exp},m) + \alpha_{max}p_{max}(z|z_{exp},m) \end{aligned} \tag{3}$$

reflects the superposition of the four possible causes for a range reading $z$. as shown in figure 5. The mixture weights $\alpha_{hit}, \alpha_{unexp}, \alpha_{max}, \alpha_{rnd}$ denote the relative contribution of the individual causes such that

$$\alpha_{hit} + \alpha_{unexp} + \alpha_{max} + \alpha_{rnd} = 1 \tag{4}$$

Notice, that the Matlab Robotics Toolbox implementation of AMCL only considers $p_{hit}$ and $p_{rnd}$ and does not provide for $p_{unexp}$ and $p_{max}$ ($\alpha_{unexp} = \alpha_{max} = 0$). In equation (3), $z$ denotes the observed range reading, $z_{exp}$ the expected reading according to robot pose and map $m$ obtained from ray tracing along the sensor axis.

In the following, you are supposed to replace the amcl localization in the navigation stack with a Matlab code. This requires to launch the Gazebo simulator and Rviz with amcl localization and robot navigation turned off. For this, move the files `house.yaml` and `house.pgm` in the `maps` folder of the `turtlebot3_navigation` package. To locate this package, use the command

```
rospack find turtlebot3_navigation
```

**Algorithm 2** MCL with particle filter in Matlab

**Input:** $u_t$ odometry, $z_t$ measurements, $m$ map of the environment

1: Configure odometry motion model with noise
2: Generate binary occupancy grid from mapfile
3: Configure sensor model with noise parameters and mixture coefficients
4: Configure sensor transform between camera and base link frame
5: Instantiate subscribers to `odom` and `scan` topic
6: Setup and initialize AMCL model
7: **for** NumberOfIterations **do**
8: Convert `scanMsg` to LidarScan object `scan`
9: Convert odometry message(`odom`) to `pose`
10: Iterate the amcl particle filter
11: Determine and publish motion command
12: Visualize the particle point cloud and map

in a terminal which will return you the corresponding path. If admin rights are required to copy files into the `maps` folder you can use the command

```
sudo cp path/to/house.yaml path/to/house.pgm path/to/turtlebot3_navigation/maps
```

Then, to achieve the whole configuration of deactivated Monte Carlo localization and navigation stack, save the launch file `turtlebot3_noamcl.launch` to the `launch` folder of the `turtlebot3_navigation` package. Again if admin rights are required, you can use

```
sudo cp path/to/turtlebot3_noamcl.launch path/to/turtlebot3_navigation/launch/
  turtlebot3_noamcl.launch
```

In two separate terminals start Gazebo and Rviz with commands:

```
roslaunch turtlebot3_gazebo turtlebot3_house.launch
roslaunch turtlebot3_navigation turtlebot3_noamcl.launch
```

The pseudo code for your script is shown in algorithm 2.

1) Compared to the real robots, the odometry data in the Gazebo simulator contain a lower noise level. To make the simulation more realistic, the helper function `GazeboNoisyOdom` is provided in the template which adds additional noise to the received odometry data. Review the `GazeboNoisyOdom` helper function.

2) The helper class `AMCLWandererRST.m` drives the robot around randomly while avoiding obstacles with VFH+. Copy it to your Matlab working directory. Use the template for the following tasks for the implementation of the AMCL algorithm.

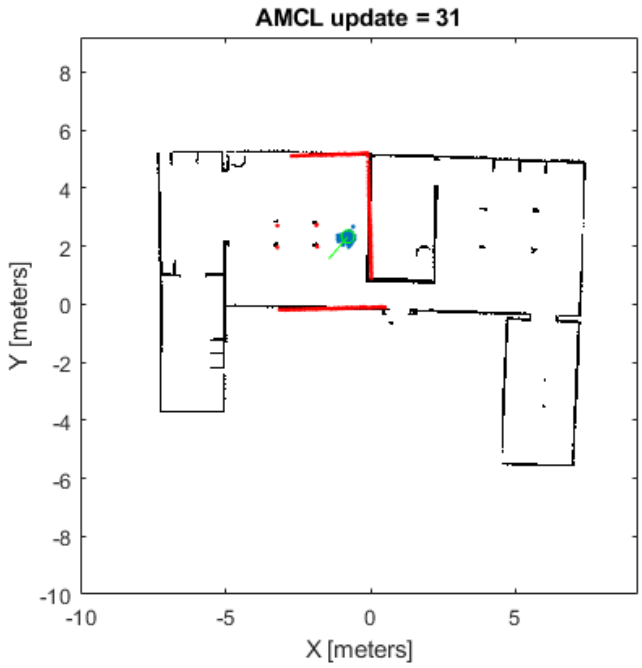


Figure 6: House map with estimated robot pose (green) , particles (blue) and scan (red).

3) Set up the turtlebot motion model (`robotics.OdometryMotionModel`): The TurtleBot motion is estimated from odometry data. The `Noise` attribute reflects the uncertainty in the robot's angular and linear movement according to equation (2). Initialize the motion model where all noise parameters $\alpha_i$ should be initialized with 0.05.

4) Set up the turtlebot sensor model (`robotics.LikelihoodFieldSensorModel`): The likelihood field method computes the probability of perceiving a set of measurements by comparing the endpoints of the range finder measurements to the occupancy map. If the end points match the occupied points in occupancy map, the probability of perceiving such a measurement is high. Initialize the sensor model with for the house map, configure the following parameters:

   - `SensorLimits` - Minimum and maximum range of sensor (defined in `scanMsg`)
   - `NumBeams` - Number of beams used for likelihood computation (defined in `scanMsg`)
   - `Map` - The `BinaryOccupancyGrid` of the map
   - `MeasurementNoise` - Standard deviation of measurement noise (a reasonable value is 0.1)
   - `RandomMeasurementWeight` - Weight $\alpha_{rnd}$ for the probability of random measurement $p_{rnd}$ (a reasonable value is 0.05)

- `ExpectedMeasurementWeight` - Weight $\alpha_{hit}$ for the probability of expected measurement $p_{hit}$ (a reasonable value is 0.95).

5) The attribute `SensorPose` of the `robotics.LikelihoodFieldSensorModel` refers to the pose of the range sensor relative to the robot. Set `SensorPose` to the coordinate transform of the sensor with respect to the robot base (`/base_link`). The transform can be obtained from the ROS transformation tree using `getTransform` by identifying the frame of `scanMsg`. Note, that the transform needs to be specified as a three-element vector, i.e. a 2D pose.

6) Set up an AMCL object `amcl` with `robotics.MonteCarloLocalization` and assign the `MotionModel` and `SensorModel` properties in the `amcl` object.

7) Configure the `amcl` object for localization with initial pose estimate and provide the following properties

```
amcl.UpdateThresholds = [0.2, 0.2, 0.1];
amcl.ResamplingInterval = 1;
amcl.UseLidarScan = true;
amcl.ParticleLimits = [200 2000];
amcl.GlobalLocalization = false;
amcl.InitialPose = AMCLGazeboTruePose; % Get true initial pose
amcl.InitialCovariance = 0.2*eye(3);
```

The helper function `AMCLGazeboTruePose` queries the true pose from the Gazebo simulator.

8) A helper class `AMCLVisualization` is provided for visualization and driving the TurtleBot. Set up a helper object which can be obtained from the `AMCLVisualization` class to plot the map and update the robot's estimated pose (green), particles (blue), and laser scan (red) on the map as shown in figure 6. The class constructor takes as an input the `BinaryOccupancyGrid` of the map.

9) Within a `while` loop

- messages arriving on the `scan` topic are received and converted to a `lidarScan` object
- the latest odometry information from the `odom` topic is received and converted to a `pose` vector regarding $x_t, y_t, \theta_t$, e.g. with the helper function `OdometryMsg2Pose`.
- update the estimated robot's pose and covariance based on the current odometry and laser `scan` object

```
[isUpdated, estimatedPose, estimatedCovariance] = amcl(odomPose, scan);
```

- Each time the ACML updates the set of particles, visualize the robot's estimated pose, particles and laser scans on the map.

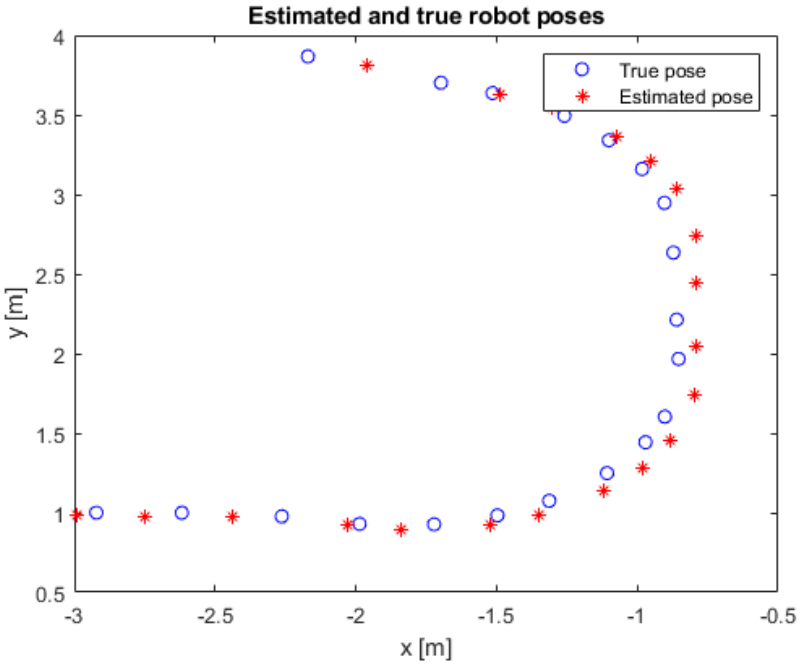


Figure 7: True robot pose (blue) vs. pose estimated by AMCL (red).

```
if isUpdated
  % Plots
  plotStep(visualizationHelper, amcl, estimatedPose, scan, iUpdate);

  iUpdate = iUpdate + 1;
end
```

If Matlab is not responding anymore or you experience the Matlab error `Connection to process with Exchange: ... was lost.`, restart Matlab and Gazebo. Try to avoid scrolling within the Matlab Live-Script when the localization is running as it may lead to a crash of Matlab as well.

10) The true pose can be obtained from the `AMCLGazeboTruePose` function at each update of the AMCL algorithm. Plot the evolution of the true and estimated pose in a separate figure as shown in figure 7.

11) Investigate the evolution of particles according to the motion model only. For that purpose set

```
rangeFinderModel.RandomMeasurementWeight = 1.0
rangeFinderModel.ExpectedMeasurementWeight = 0.0
```

which ignores the range information as all readings become equally likely. Initialize the pose at the true robots pose with zero covariance and observe the spread of

the point cloud. To accentuate the particle spread, increase the noise in the motion model.

```
odometryModel.Noise = [1.0 1.0 1.0 1.0];
```

Reset the parameters to their original values afterwards.

12) Analyze the performance of the `amcl` localization when using the noisy odometry poses from `GazeboNoisyOdom()` instead of the original odometry poses which were obtained directly from the `odom` topic.

13) Reconfigure your AMCL algorithm for global localization. For that purpose, set the `amcl` property `GlobalLocalization` to `true` in which case the particles are uniformly distributed across the entire map. Hence, global localization requires a large number of particles. Increase the lower and upper bounds on the number of particles `ParticleLimits` in the `acml` object. Test the ability of the algorithm to estimate the true pose by global localization.

```
amcl.ParticleLimits = [2000 10000];
amcl.GlobalLocalization = true;
```

14) (Optional) Test the ability of the AMCL algorithm to recover from the kidnapped robot problem. While the Gazebo simulator is running, displace the robot in the simulator and observe whether the ACML algorithms recovers from the kidnapping. For this, make the robot stop after half of the update iterations within the while loop using

```
    if iUpdate == floor(0.5 * numUpdates)
        laserSub.NewMessageFcn = [];
        t_start_reverse = rateObj.TotalElapsedTime;
        disp('Displace the TurtleBot in the Gazebo simulation...')
        while (rateObj.TotalElapsedTime - t_start_reverse < 30.0)
          stop(wanderHelper);
          waitfor(rateObj);
        end
        laserSub.NewMessageFcn = @wanderHelper.scan_callback;
    end
```

When the robot stops, you have 30 s time to displace the robot in Gazebo. For this, select the tab *World -> Models -> turtlebot3* (the last). This should select the robot visualized by a white bounding box around the robot in Gazebo. Now, press TTön the keyboard which enables the translation mode and drag-and-drop the robot to a different place. Press Ësctto leave the translation mode afterwards. Now, wait until the localization procedure continues.

### Dual Monte Carlo Localization

Dual Monte Carlo localization inverts the sampling process, by exchanging the roles of the sampling step and the weighting of samples in Monte Carlo Localization. The dual MCL

**Algorithm 3** Dual MCL with particle filter

**Input:** $\mathcal{X}_{t-1}$: set of particles at time $t-1$, $u_t$ control or odometry, $z_t$ measurement, $m$ map of the environment

1: $\mathcal{X}_t = \emptyset$;
2: generate kd-tree $b$ from $\mathcal{X}_{t-1}$
3: **for** $m = 1$ **to** $M$ **do**
4: generate random $x' \sim p(x'|z)/\pi(z)$
5: generate random $x \sim p(x'|\bar{u}, x)/\pi(x'|u)$
6: $w' = b(x)$
7: add $< x', w' >$ to $\mathcal{X}_t$
8: normalize the importance factors $w'$ in $\mathcal{X}_t$
9: **return** $\mathcal{X}_t$;

algorithm in described in algorithm 3 generates samples of the pose $x'$ according to the proposal distribution $p(x'|z)/\pi(z)$ (with $\pi(z) = \int p(z|x)dx$) and weights those samples according to $p(x|x', u)$. It thus samples according to the (inverse) sensor model and updates weights according to the motion model. The initial belief $Bel(x)$ is transformed into a kernel density-tree in order to map a sample based representation to a continuous probability density function $b(x)$. For each sample $x'$ one draws a sample from $p(x'|\bar{u}, x)/\pi(x'|u)$ with $(\pi(x'|u) = \int p(x'|x, u)dx)$. For the sake of simplicity one assumes $\pi(x'|u)$ to be constant. The sample originates from the projection of the successor pose $x'$ back to its hypothesized predecessor pose $x$, thus reverse the roles of $x'$ and $x$ w.r.t. ordinary sampling step of the motion model. The weights $w'$ are calculated according the belief $Bel(x)$ which is computed from a kd-tree that represents the belief density $b$. Unfortunately, dual MCL alone is insufficient for localization.

### Mixture Monte Carlo Localization

In most applications the standard MCL is sufficient for accurate robot localization. Nevertheless neither the standard MCL nor the dual MCL demonstrate good performance across all scenarios and robot sensors. The standard MCL algorithm does not work well with highly accurate range sensors as $p(z|x)$ is sparse. The dual MCL is not robust w.r.t. to sensor failures as it only considers the most recent perception $z$. Mixture Monte Carlo localization combines the strengths and weakness of standard and dual MCL [3]. Mixture MCL generates a fraction $1 - \phi$ of samples by standard MCL, and the remaining samples by dual MCL. Both sample sets are merged prior to normalization. The weights $w' = b(x)$ of the dual samples has to be adjusted by the factor $\pi(z)\pi(x'|u)$ to match it with the weights $w' = p(z|x)$ of the standard MCL samples.

### K-D Tree

A k-d tree is a data structure for partitioning and organizing points in a k-dimensional space. k-d trees support efficient range and nearest neighbor search of multidimensional

**Algorithm 4** Mixture MCL with particle filter

**Input:** $\mathcal{X}_{t-1}$: set of particles at time $t-1$, $u_t$ control or odometry, $z_t$ measurement, $m$ map of the environment

```
1: X_t = ∅;
2: generate kd-tree b from X_{t-1}
3: for m = 1 to M do
4:     if with probability 1 − φ then
5:         generate random x from X_{t-1} according to w_1, ..., w_M
6:         generate random x' ~ p(x'|x, u)
7:         w' = p(z|x')
8:         add < x', w' > to X_t
9:     else
10:        generate random x' ~ p(x'|z)/π(z)
11:        generate random x ~ p(x'|ū, x)/π(x'|u)
12:        w' = π(z)π(x'|u)b(x)
13:        add < x', w' > to X_t
14: normalize the importance factors w' in X_t
15: return X_t;
```

data. The k-d tree construction proceeds in the following manner: Grow the tree by iterating through the axes (dimensions) to select the plane for the next split. For the 2d-tree in figure 8 the root has an x-aligned plane, the root's children both have y-aligned planes, the root's grandchildren (in this case leaf nodes) have x-aligned planes and so forth. Points are inserted by selecting the median of the points that belong to a subtree partition. The partition stops at a leaf node once the region contains a single point. In case of a 2d tree each subtree corresponds to a rectangular region. This procedure generates a balanced k-d tree, in which each leaf node is located about the same distance from the root node. Figure 8 illustrates the k-d tree spatial decomposition and the corresponding k-d tree structure. The first partition at the root $(7, 2)$ splits the data into the left set $(5, 4)(2, 3)(4, 7)$ with $x_i < 7$ and the right set $(9, 6)(8, 2)$ with $x_i > 7$. The set $(5, 4)(2, 3)(4, 7)$ is split along the y-axis w.r.t. $(5, 4)$ into the lower set $(2, 3)$ with $y_i < 4$ and the upper set $(4, 7)$ with $y_i > 4$. The partition stops as both $(2, 3)$ and $(4, 7)$ are single points.

### Kernel Density Estimation

Kernel density estimation (KDE) is a non-parametric way to estimate the probability density function of a random variable. Kernel density estimation is a fundamental data smoothing problem where inferences about the population are made, based on a finite data sample. The data are usually thought of as a random sample from that population.

[2]By KiwiSunset at the English language Wikipedia, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=16242249`, By MYguel - Own work, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=4535483`

[3]CC BY 2.5, `https://commons.wikimedia.org/w/index.php?curid=2377345`

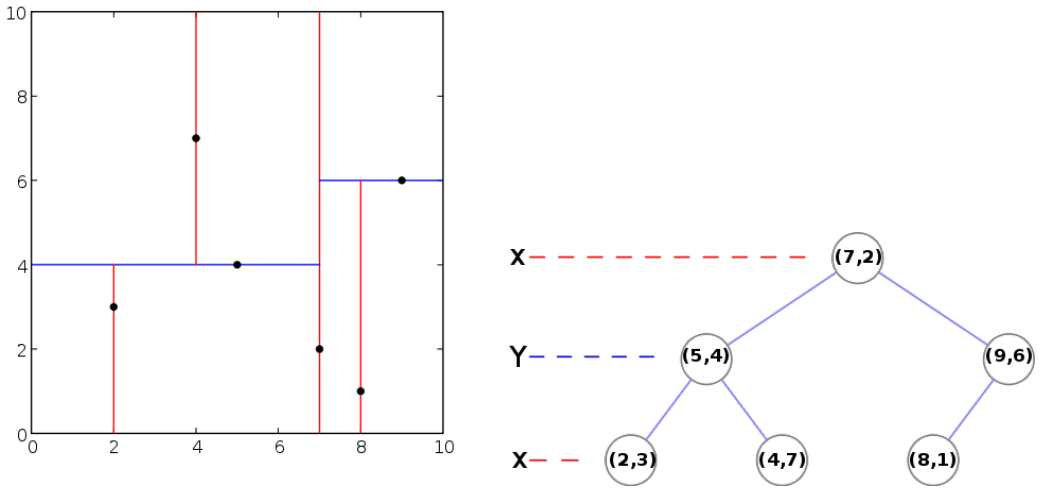


Figure 8: k-d tree decomposition for the point set (2,3), (5,4), (9,6), (4,7), (8,1), (7,2) (left), resulting k-d tree (right) [2]

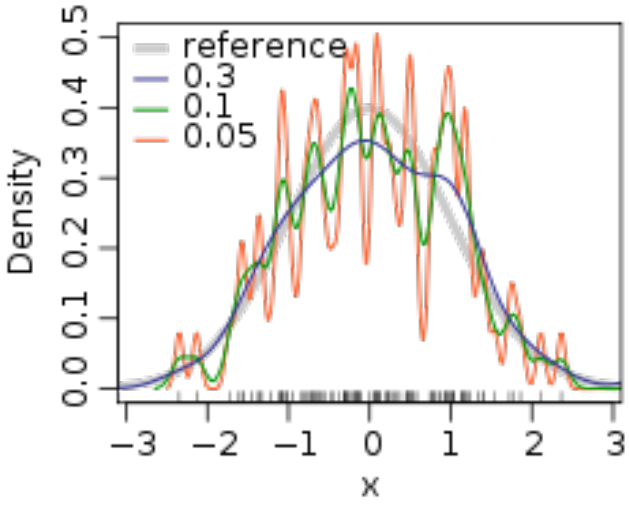


Figure 9: kernel density estimation with various bandwidth $h$ for a sample set $x_i$ [3]

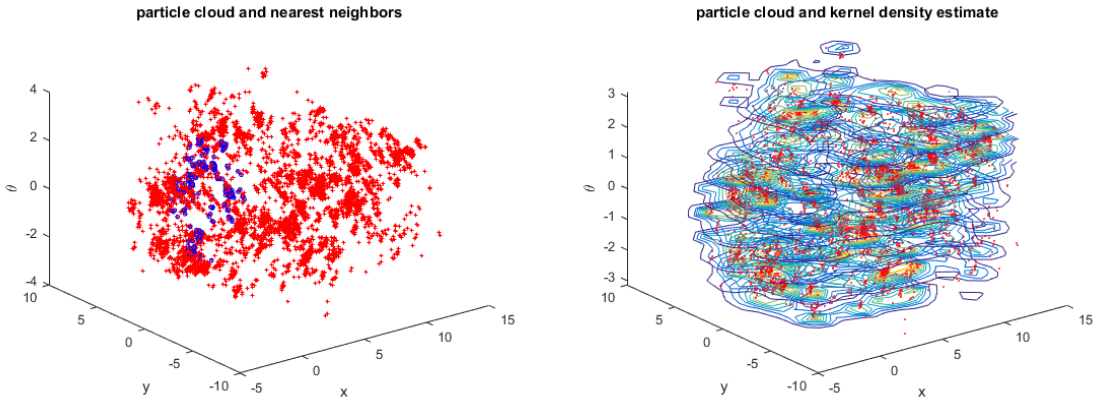


Figure 10: Particle cloud with $N = 500$ nearest neighbors to pose [2 2 0], particle cloud with estimated kernel density

Let $x_1, x_2, \ldots, x_n$ be an independent and identically distributed sample drawn from some distribution with an unknown density $f$. We are interested in estimating the shape of this function $f$. Its kernel density estimator is

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) = \frac{1}{nh}\sum_{i=1}^{n} K_h(\frac{x - x_i}{h}) \tag{5}$$

where $K()$ is the kernel - a non-negative function that integrates to one and has mean zero, and $h > 0$ is a smoothing parameter called the bandwidth. A common kernel is a Gaussian kernel

$$K(d) \sim e^{(\frac{-d^2}{h})} \tag{6}$$

For large sample sets, it suffices to restrict the kernel density estimation to the $N$ nearest neighbours of the query point $x$.

$$\hat{f}_h(x) = \frac{1}{k}\sum_{i \in \text{k-nearest neighbors}} K_h(x - x_i) = \frac{1}{kh}\sum_{i \in \text{n-near. neighb.}}^{n} K_h(\frac{x - x_i}{h}) \tag{7}$$

As the kernel has a limited support the contribution to $\hat{f}_h(x)$ of summands with data $x_i$ far from $x$ is negligible.

15) Retrieve the particles generated at each iteration of your AMCL algorithm for global localization with `[poses, weights]=amcl.getParticles()` which returns the current particles of the MonteCarloLocalization object. Notice that the number of rows can change with each iteration of the MCL algorithm.

16) Plot the particle poses over the course of AMCL iterations in a separate 3D plot (`plot3`).

17) Build a k-d tree for the particle set at each iteration step of the particle filter with `KDTreeSearcher`.

18) Search the k-d tree for the $K = 500$ nearest neighbors of the pose $x = [2\ 2\ 0]$ with `knnsearch(kdtree, x, 'K', K)`. Overlay the original particle plot with the plot of the nearest neighbours in a different colour as shown in figure 10.

19) (Optional) `knnsearch` not only provides the index to the $K$ nearest neighbours but also the mutual distance to the query point. Calculate the continuous probability density of the particle sets at each iteration for a grid that spans the volume $x \in [-5,\ 15]$, $x \in [-10,\ 10]$ and $\theta \in [-\pi,\ \pi]$. For this, use the kernel density estimation according to equation (7) from the subset of nearest nearest neighbours. Use a Gaussian kernel as defined in equation 6 with $h = 1$. Visualize the probability density together with the particles in a contour slice plot (`contourslice`) as shown in the right part of figure 10. Initially, set up the grid using

```
x = -5:1:15;
y = -10:1:10;
theta = -pi:pi/4:pi;
[X,Y,Theta] = meshgrid(x,y,theta);
V=zeros(size(X));

```

For plotting the estimated kernel density use

```
for i=1:numel(X)
  [idx_knn, dists_knn] = knnsearch(kdtree, [X(i), Y(i), Theta(i)], 'K', 500);
  V(i)= % your code here
end
contourslice(X, Y, Theta, V, [], [], theta);
```

### Maximum Likelihood Estimate of Sensor Model Parameters

It is difficult to obtain the parameter vector $\beta = [\alpha_{hit}, \alpha_{rnd}, \alpha_{unexp}, \alpha_{max}, \sigma, \lambda]$ of the sensor model (see figures 3, 4 and 5) from an analysis of the interactions between sensors and environment. Rather, the optimal parameters are obtained by a maximum likelihood estimate based on observed range readings of the robot at known poses and thus known $z_{exp}$. Assume a set of data of observed readings $z_i$ and expected readings $z_{exp,i}$. The optimal parameters of the model are those that best explain the observations, exhibiting the highest likelihood of the data. Rather than to maximize the likelihood, one maximizes the log likelihood under the reasonable assumption of independence of observations.

$$\beta^* = \underset{\beta}{argmax} \log P(Z|\beta, Z_{exp}) = \underset{\beta}{argmax} \sum_i \log p(z_i|\beta, z_{exp,i}) \tag{8}$$

A similar maximum likelihood approach is used to determine the optimal parameters of the motion model $\alpha = [\alpha_1, \alpha_2, \alpha_3, \alpha_4]$ from true robot motions and odometry information.

20) The file `sensordata.mat` contains sensor range readings `sensordata(:,1)` and the corresponding expected reading from ray tracing (clipped at maximum range reading) `sensordata(:,2)`. Plot the histogram of true range readings $z_i$ with `histogram`. Use the command `yyaxis left` before plotting to plot the histogram w.r.t. the left y-axis. The overall readings originate from a sensor range $z_i \in [0.0, 2.0]$.

21) Implement a function

```
function p = pmeasurement_simplified(z,zexp,beta)

```

that computes the likelihood of a range reading `z`, for an expected reading `zexp` and sensor model parameters $\beta = [\alpha_{hit}, \alpha_{random}, \sigma_{hit}]$ according to the simplified version of equation (3) ignoring $p_{unexp}, p_{max}$.

The probabilities $p_{hit}$ and $p_{rand}$ for each sensor reading are calculated from the actual sensor readings `z(k)` and the expected sensor readings `zexp(k)` with the Matlab function `pdf`:

```
y = pdf('name',x,A,B)

```

returns the two-parameter distribution family specified by the string `name`, evaluated at the values in `x`. `A` and `B` denote the parameters for the distribution, typically `A` denotes the mean and `B` the variance. Use the following code as a template for implementing the function:

```
function p = pmeasurement_simplified(z,zexp,beta)

  sigmahit = % your code here

  n=length(z);
  p=zeros(n, 1);
  out_of_range=2.0;
  for k = 1:n
    if (z(k) < out_of_range)
      phit=pdf('Normal',z(k),zexp(k),sigmahit);
    else
      phit=0;
    end
    prand = pdf('Uniform',z(k),0,out_of_range);

    p(k) = % your code here
  end
end
```

The input and output arguments `p, z, zexp` are supposed to be vectors (arrays) of data not merely scalars, `beta` contains the parameter vector.

22) Implement a function

```
function loglikelihood = logpdata(z,zexp,beta)

```

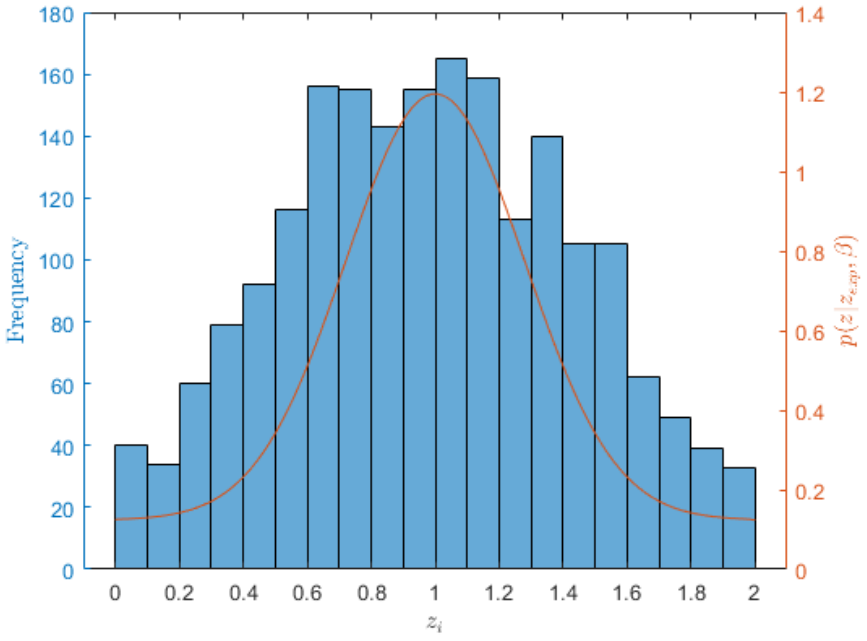


Figure 11: Histogram of range readings $z_i$ and probability of sensor model for $z_{exp} = 1.0$ with the identified parameter vector $\beta^*$.

that calculates the log likelihood of the entire data vector `z, zexp` given the parameter vector `beta` according to equation (8). Compute the log-likelihood of the entire probability vector `p` for a parameter vector $\beta = [0.6\ 0.4\ 0.5]$ by summation of the logarithms of the probability vector `p`.

23) Maximize the log likelihood of the data w.r.t. the parameter $\beta = [\alpha_{hit}, \alpha_{random}, \sigma_{hit}]$ subject to the constraint that $\alpha_{hit} + \alpha_{random} = 1.0$. Use `fmincon` to find the parameter vector $\beta^*$ that maximizes the argument in equation (8). The first argument to `fmincon` is a function handle to your objective function, which is the negative of `logpdata` as you want to maximize the likelihood of the data (see Scientific Programming Optimization Toolbox).

24) Plot the probability of the sensor model for $z_{exp} = 1.0$ with the identified parameter vector $\beta^*$ into the same figure as shown in Fig. 11. Use the command `yyaxis right` before plotting since the probability has a different scale than the histogram.

25) (Optional) Maximize the log likelihood of the data w.r.t. the full parameter vector $\beta = [\alpha_{hit}, \alpha_{random}, \alpha_{unexp}, \alpha_{max}, \sigma_{hit}, \lambda]$ subject to the constraint that $\alpha_{hit} + \alpha_{random} + \alpha_{unexp} + \alpha_{max} = 1.0$. Use an exponential function for modelling the probability of unknown sensor readings (see figure 4). For this, extend the likelihood function according to:

```
function p = pmeasurement(z,zexp,beta)
```

```
2
  sigmahit = % your code here
4  lambda = % your code here

6  n=length(z);
  p=zeros(n, 1);
8  out_of_range=2.0;
  for k = 1:n
10   if(z(k) < out_of_range)
       phit = pdf('Normal',z(k),zexp(k), sigmahit);
12     if (z(k) < zexp(k))
         eta = 1 - exp(-zexp(k) * lambda);
14       punexp = pdf('Exponential', z(k), 1/ lambda) / eta;
       else
16       punexp = 0;
       end
18     pmax = 0;
     else
20     phit = 0;
       punexp = 0;
22     pmax = 1;
     end
24   prand = pdf('Uniform',z(k),0,out_of_range);
     p(k) = % your code here
26 end
end
```

Use `fmincon` to find the new parameter vector $\beta^*$ that maximizes the argument in equation (8). Therefore, extend `lb, ub, Aeq` and `betainit` according to

```
    lb = [0 0 0 0 0 0];
2   ub = [1 1 1 1 5 5];
    Aeq = [1 1 1 1 0 0];
4   betainit = [0.6 0.4 0 0 0.5];
```

Plot the histogram of the sensor readings together with the probability of the new sensor model for $z_{exp} = 1.0$ with the new identified parameter vector $\beta^*$.

# References

[1] Sebastian Thrun, Wolfram Burgard, Dieter Fox, *Probabilistic Robotics*, MIT Press, (2005)

[2] F. Dellaert, D. Fox, W. Burgard, S. Thrun, Monte Carlo Localization for Mobile Robots, *Proceedings 1999 IEEE International Conference on Robotics and Automation*

[3] Sebastian Thrun, Dieter Fox, Wolfram Burgard, Frank Dellaert, Robust Monte Carlo localization for mobile robots, *Artificial Intelligence*, vol. 128, (2001), pp. 99-141