EECE 5550 Mobile Robotics Lab #3

David M. Rosen

Due: Dec 1, 2021

Objective: In this lab, you will apply tools from the Robot Operating System (ROS) to build a map of an initially unknown environment using maximum-likelihood-based graphical SLAM and occupancy grid mapping, and then (globally) relocalize into that map using Monte Carlo localization (MCL). These operations (especially the first) will be a core building block of the systems you implement for the course final project.

Problem 1: Map acquisition using Cartographer

In order to perform localization, one must first have a map to localize against. Therefore, our first objective will be to construct a 2D occupancy grid map of the robot's operating environment.

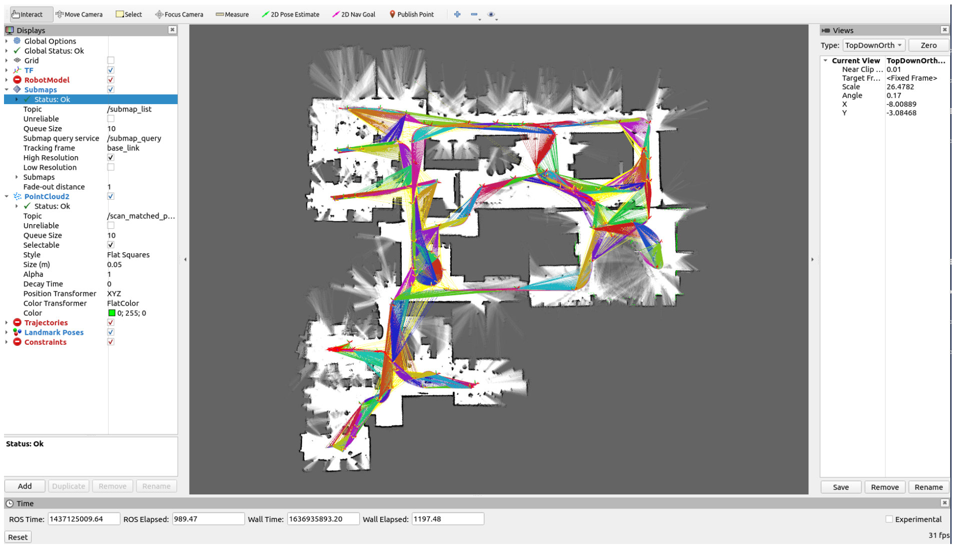
To do so, we will make use of the Cartographer ROS package. Cartographer is a complete SLAM system for performing 2D and 3D occupancy grid mapping using laser and IMU data.

In brief, the Cartographer system is based upon the idea of *submapping.* It uses laser scan matching (and IMU data, if available) to estimate the (local) motion of the laser scanner over *sequences* of scans (as we discussed in class). Since this odometric estimation can be done fairly accurately over short sequences (e.g. a few dozen scans), it is possible to build accurate *submaps* by fusing these short sequences of laser scans into an occupancy grid. Cartographer incrementally generates these (small) submaps as the robot explores, and co-registers them into a coherent *global* map by solving a graphical SLAM problem to compute the optimal registration $T_{WM_i} \in \mathrm{SE}(d)$ of the ith submap M_i in the global coordinate frame W, given (i) the estimated odometry between subsequent submaps and (ii) any loop closures between submaps that are discovered by comparing each newly-acquired laser scan against all previously-built submaps. (You may refer to the original paper for a more detailed description of the Cartographer system.)

Our goal in this exercise will be to install the Cartographer system, exercise some of its functionality, and then run it on the Turtlebot to produce a (global) occupancy grid map suitable for localization.

(a) **Install Cartographer:** Go to the SLAM page of the ROBOTIS e-manual for the Turtlebot3, and check the box labeled "Noetic" at the top of the page (to make sure that you're referring to the instructions for the Noetic version of ROS). Then scroll down to the bottom of Section 4.1 (Run SLAM node), and click to expand the green box labeled "Read more about other SLAM methods". Follow the instructions in the "Cartographer" section to download and install the Cartographer ROS package.

(b) **Run Cartographer demo:** You can verify that the Cartographer package has been built and installed correctly by running the "Revo LDS" demo described here; this demo will run Cartographer on a pre-recorded dataset collected using the laser scanner on a low-cost robotic vacuum cleaner. This will take about 15 minutes to run (in real-time), and should produce a final SLAM reconstruction that looks similar to the one shown in Fig. 1.



```
Global Options
Global Status: Ok
Grid                      ☐
TF                        ✓
RobotModel                ✓
Submaps                   ✓
  Status: Ok
  Topic                   /submap_list
  Unreliable              ☐
  Queue Size              10
  Submap query service    /submap_query
  Tracking frame          base_link
  High Resolution         ✓
  Low Resolution          ☐
  Submaps
  Fade-out distance       1
PointCloud2               ✓
  Status: Ok
  Topic                   /scan_matched_p...
  Unreliable              ☐
  Queue Size              10
  Selectable              ✓
  Style                   Flat Squares
  Size (m)                0.05
  Alpha                   1
  Decay Time              0
  Position Transformer    XYZ
  Color Transformer       FlatColor
  Color                   🟩 0; 255; 0
Trajectories              ✓
Landmark Poses            ✓
Constraints               ✓

Status: Ok
```

```
Type: TopDownOrth
Current View    TopDownOrth...
  Near Clip ...   0.01
  Target Fr...    <Fixed Frame>
  Scale           26.4782
  Angle           0.17
  X               -8.00889
  Y               -3.08468
```

```
ROS Time: 1437125009.64   ROS Elapsed: 989.47   Wall Time: 1636935893.20   Wall Elapsed: 1197.48   ☐ Experimental
31 fps
```

Figure 1: Cartographer SLAM reconstruction for Revo LDS demo

(c) **Saving and loading Cartographer SLAM reconstructions:** You can save the Cartographer SLAM reconstruction computed in step (b) to disk for future use by issuing the following commands in the terminal:

- Finalize the current trajectory/SLAM estimate:
  `$ rosservice call /finish_trajectory 0`
- Save Cartographer's internal state to a Protobuf stream (.pbstream) file:
  `$ rosservice call /write_state "filename: '$HOME/catkin_ws/revo_lds.pbstream' include_unfinished_submaps: false"`

Similarly, you can reload this reconstruction and visualize it in RViz by issuing the following command in the terminal:

```
$ roslaunch cartographer_ros visualize_pbstream.launch
pbstream_filename:=$HOME/catkin_ws/revo_lds.pbstream
```

(d) **Global map fusion:** As we saw earlier, Cartographer's internal map representation is a *collection of submaps* M_i, each of which maintains its own (local) coordinate system and the estimated transformation T_{WM_i} relating that submap to the global coordinate frame. This is a convenient representation for a *SLAM system* (since it explicitly models the [uncertain] registration T_{WM_i} of each submap in the global frame, which will need to be adjusted as the SLAM estimation runs), but not so much for a *localization system* (which assumes access to a [single] *global* map). Therefore, as a final processing step, we would like to use Cartographer's final estimates T_{WM_i} for each submap's registration in the global frame to *combine* them all into a *single* 2D occupancy grid that we can use for localization.

For *online* operation, the Cartographer system includes an `occupancy_grid_node` that automatically performs this fusion operation, and publishes the resulting (global) occupancy grid map as an `OccupancyGrid` message on the `/map` topic; this makes it easy for Cartographer to interoperate with other ROS nodes that require occupancy maps as input (for example, path planners in the ROS navigation stack).

Alternatively, given a .pbstream file containing a finalized Cartographer SLAM reconstruction (including estimates for all of the sensor poses) and a rosbag containing the original sensor data, Cartographer provides a specialized `assets_writer` node that can create high-resolution *global* occupancy grid maps by fusing the *complete* set of raw sensor scans contained in the original .bag file into an occupancy grid according to the (final) estimated sensor poses in the .pbstream file.

For example, we can extract a global occupancy grid map from the saved reconstruction in part (c) by issuing the following command in the terminal:

```
$ roslaunch cartographer_ros assets_writer_ros_map.launch
bag_filenames:=$HOME/Downloads/cartographer_paper_revo_lds.bag
pose_graph_filename:=$HOME/catkin_ws/revo_lds.pbstream
```

This will output a high-resolution occupancy grid map in the form of a .pgm file containing the raw occupancy probabilities assigned to each cell, and a .yaml file containing a small amount of metadata for the map (e.g. the resolution of each grid cell, in meters); for this example, the resulting .pgm file should look like the one shown in Fig. 2. Both of these files will share the same prefix as the .bag file from which they were created (so in this example, they will appear in the Downloads directory).

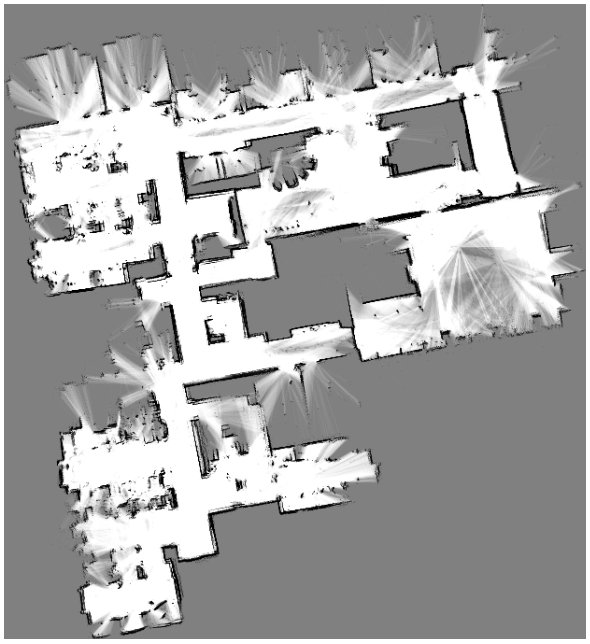


Figure 2: Fused occupancy grid constructed for Revo LDS demo

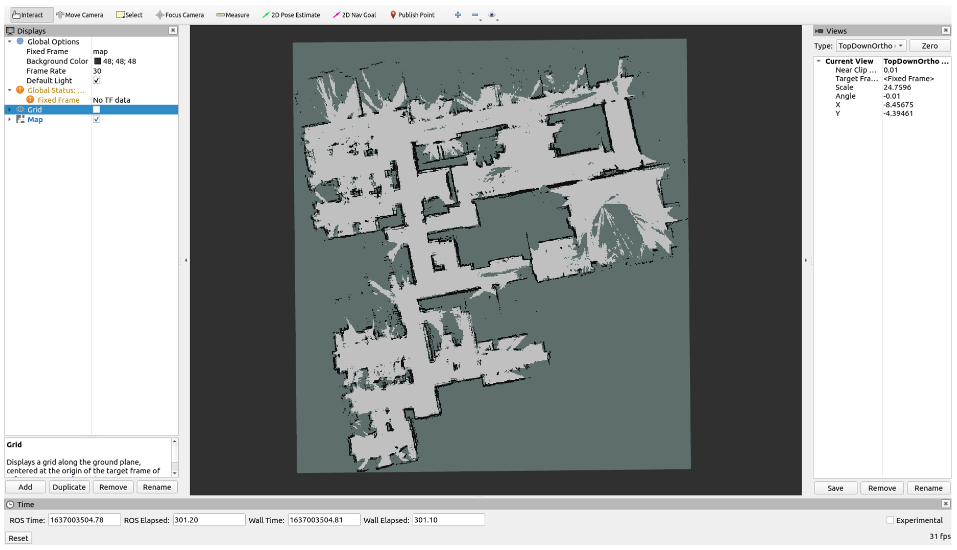
(e) **Loading saved occupancy grids:** The ROS `map_server` package provides functionality for reloading and publishing `OccupancyGrid` maps that have previously been saved to disk. For example, to reload the occupancy grid map that you constructed and saved in part (d), you may issue the following commands (in three separate terminals):

- Terminal 1: Start main ROS process: `$ roscore`
- Terminal 2: Start RViz visualization: `$ rosrun rviz rviz`

– Terminal 3: Run `map_server` node to reload and publish previously-constructed occupancy grid map:

```
$ rosrun map_server map_server $HOME/Downloads/cartographer_paper_revo_lds.bag_map.yaml
```

To display the reloaded occupancy grid map in the RViz viewer, click on the "Add" button in the bottom-left corner of the main display, select the "By topic" tab in the popup window, highlight the "`/map`" topic, and then click "OK". The result should be that the RViz viewer now displays the reloaded occupancy grid map, as in Fig. 3.



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Default Light ✓
Global Status: ...
Fixed Frame No TF data
Grid ☐
Map ✓

Grid

Displays a grid along the ground plane, centered at the origin of the target frame of

Type: TopDownOrtho
Current View TopDownOrtho ...
Near Clip ... 0.01
Target Fra... <Fixed Frame>
Scale 24.7596
Angle -0.01
X -8.45675
Y -4.39461

ROS Time: 1637003504.78 ROS Elapsed: 301.20 Wall Time: 1637003504.81 Wall Elapsed: 301.10

Figure 3: Reloaded occupancy grid map

Note that when the `map_server` reloads an occupancy grid, its default behavior is to output a ***trinary-valued*** grid map whose cells are labeled as *free*, *occupied*, or *unknown*, by *thresholding* the occupancy probabilities contained in the original (saved) occupancy grid. See the (short) `map_server` documentation for more information, including how to modify this default behavior.

Now that we've installed the Cartographer SLAM system and exercised its basic functionality, we turn to our main objective for the first half of the lab: building a map of the Turtlebot3's operating environment.

(f) **Live data capture and SLAM reconstruction on the Turtlebot3:** In this portion of the lab, you will manually drive a Turtlebot3 around its environment in order to construct a map, using the Cartographer SLAM system.

– Follow the Turtlebot3 bringup procedure to start the core ROS system on the robot.

– Start the `turtlebot3_teleop` node: this will enable you to manually drive the Turtlebot3 around (using your computer keyboard) in order to scan the environment for mapping.

– Start recording to a ROS .bag file (to capture the raw sensor data for later use).

– Start the Cartographer SLAM system, following the instructions on the SLAM page of the ROBOTIS e-manual for the Turtlebot3.

At this point, you should be able to see a real-time visualization (in RViz) of Cartographer's SLAM reconstruction. Using the computer keyboard, manually drive the robot to generate a complete occupancy grid map of its environment (e.g. a room or potion of a house).

(g) **Global occupancy grid map construction:** After you have finished scanning the Turtlebot3's environment, repeat steps (c) and (d) to save the Cartographer SLAM reconstruction and fuse the complete set of collected sensor data into a *global* occupancy grid that can be used for localization.

Submission: After completing this problem, submit the following:

- your .bag file containing the raw sensor data.[1]
- the .pbstream file containing the finalized Cartographer reconstruction
- the .pgm and .yaml files for the final fused (global) occupancy grid map for the environment.
- A video containing a side-by-side display of (i) a screen recording of the RViz display as the Cartographer mapping is being done, and (ii) the robot itself as it is capturing the mapping data (this can be a cell-phone video). You may record each of these separately, and then produce the final (submitted) video by compositing them together using a video editor (e.g. OBS Studio or similar).

Problem 2: Global relocalization using AMCL

When a robot comes back online after having been switched off, it may have high uncertainty regarding its current location (for example, if it was possibly moved while shut down – this is the so-called "wake-up robot problem"). It is thus necessary to perform *global* relocalization in order to reacquire a high-precision estimate of its pose. As we saw in class, one of the major advantages of particle-filter-based localization algorithms is that they are capable of automatically capturing the complex, highly-diffuse, and possibly multi-modal probability distributions that arise in this global relocalization task.

In this exercise, your task will be to demonstrate global relocalization into the map that you built in Problem 1, using the amcl package in ROS's navigation stack. As its name suggests, amcl implements the adaptive Monte Carlo localization algorithm that we discussed in class.

(a) **Dataset acquisition:** Collect a **separate** dataset (in a ROS .bag file) to use for relocalization by driving your Turtlebot through the same environment that you mapped in Problem 1.

(b) **Global relocalization demo:** In this exercise, you will configure and launch a network of ROS nodes to implement global relocalization using amcl. Specifically, your task will be to write a ROS launch file that performs the following actions:

   – Sets the ROS parameter `use_sim_time` to `true`: this tells the ROS system to use timestamps provided by the "/clock" topic, rather than the real (wall-) time. This is useful for working with prerecorded data, especially when playing back at speeds other than real-time.

   – Starts a `map_server` node that reloads the occupancy grid map you constructed in Problem 1.

[1] As .bag files can be quite large, you may upload this to a cloud storage service (e.g. Dropbox or Google Drive), and then simply submit a link

- Starts an amcl node with a high initial pose covariance (high enough to represent that the pose uncertainty is on the order of the scale of your map).
- Starts an RViz node that is configured to display the following data:
  * The occupancy grid map
  * The particle cloud provided by amcl
  * The raw laser scan data

  (You can save an RViz configuration as described here, and reload the same configuration by passing the configuration file as an argument when starting RViz.)
- Starts playback of your localization dataset. Note that you will need to pass the "–clock" argument to this player in order to publish timestamps for pre-recorded data to the /clock topic. You may also want to pass a rate argument to slow down the playback speed (depending upon the processing power of the computer you are using).
- Passes any additional arguments needed to configure the above nodes (i.e. names of topics on which to subscribe/publish, etc.) to work correctly, or passes any other required nodes/services.

When implemented correctly, it should be possible to launch your entire relocalization demo by issuing the command `$ roslaunch <your launch file>` at the command line (to start your processing pipeline, visualization, and playback), followed by the command `$ rosservice call global_localization` (to trigger global relocalization by reinitializing amcl's particle set uniformly over the map). A successful demonstration is one in which the particle set eventually coalesces to a tight cluster around the robot's true position.[2]

When you are finished, submit the following:

- Your relocalization dataset (.bag file)
- Your .launch and RViz configuration files
- A second video showing a side-by-side display of your RViz display, and your robot as it is capturing the relocalization dataset.

Challenge Problem: Implementing Monte-Carlo Localization (extra credit)

If you are feeling ambitious, you may want to try implementing your own Monte Carlo localization algorithm :-). This exercise will walk through the pieces that such an implementation requires.

Recall the basic form of the Monte Carlo Localization (MCL) algorithm that we saw in class (Algorithm 1). We observe that this algorithm consists of three basic steps:

- Sampling from the forward motion model $p(x_{t+1}|x_t, u_t)$.
- Computing the likelihood $p(z_t|x_t, m)$ of a measurement z_t given a robot pose $x_t \in \mathrm{SE}(d)$ and a reference map m.
- Resampling the propagated particles $x_t^{[k]}$ with probability proportional to the weights $w_t^{[k]}$.

[2]Note that in order for this to occur, you robot must actually have enough sensor data to disambiguate where it is! So in your relocalization dataset, you should take care to capture a sufficiently long and interesting trajectory that your robot can make this determination.

Algorithm 1 Monte Carlo localization

Input: Prior particle set $\mathcal{X}_{t-1}$, last motion command u_{t-1}, current sensor scan z_t, map m.
Output: Posterior particle set $\mathcal{X}_t$.

```
 1: function MCL($\mathcal{X}_{t-1}, u_{t-1}, z_t, m$)
 2:     Initialize empty particle set $\mathcal{X}_t = \varnothing$.
 3:     for $k = 1, \ldots, N$ do
 4:         $x_t^{[k]} = \text{SAMPLE\_MOTION\_MODEL}(x_{t-1}^{[k]}, u_{t-1})$        ▷ Prediction step
 5:         $w_t^{[k]} = \text{MEASUREMENT\_MODEL}(z_t, x_t^{[k]}, m)$       ▷ Compute particle weights
 6:     end for
 7:     for $k = 1, \ldots, N$ do                                           ▷ Resampling step
 8:         Randomly draw label $l$ from $\{1, \ldots, N\}$ with probability proportional to $w_t^{[l]}$
 9:         Add particle $x_t^{[l]}$ to $\mathcal{X}_t$
10:     end for
11:     return $\mathcal{X}_t$
12: end function
```

In this exercise, you will write a set of small functions to implement each of the above basic operations, and then implement a ROS MCL localization node that ties them all together.

(a) **Motion model:** Consulting the documentation, we discover that the `turtlebot3_teleop` package works by issuing `cmd_vel` messages: that is, it works by controlling the *target velocity* of the Turtlebot3. This suggests that we should use a *velocity motion model*, as we discussed in class.

Now we saw earlier [in Problem 2(b) of Lab 2] that these velocity commands are expressed as instantaneous velocities in the robot's *body-centric* coordinate frame – that is, as elements of the Lie group SE(2)! This gives us a very nice way of formalizing a velocity motion model, as follows:

Let $x_0 \in \mathrm{SE}(2)$ be the initial pose of the robot, $v \in \mathrm{Lie}(\mathrm{SE}(2))$ the *commanded* robot velocity, and $\Delta v \in \mathrm{Lie}(\mathrm{SE}(2))$ the *process noise*: the (additive) *error* in the commanded velocity. Given this initial data, derive an expression for the pose $x(t) \in \mathrm{SE}(2)$ of the Turtlebot3 as a function of time t.

(b) **Motion model sampling function:** Using the result of part (a), write a function that implements the motion model sampler required in line 4 of the MCL algorithm. This function should accept the following arguments:

- A particle set $\mathcal{X}_{t-1} \subset \mathrm{SE}(2)$ of initial robot poses
- A commanded velocity $v_{t-1} \in \mathrm{Lie}(\mathrm{SE}(2))$
- A symmetric positive-definite matrix $\Sigma_{t-1} \succ 0$ parameterizing a mean-zero Gaussian distribution $\mathcal{N}(0, \Sigma_{t-1})$ over velocity noise
- An elapsed time $\Delta t \in \mathbb{R}$.

Given these arguments, your function should do the following:

- For each particle $x_{t-1}^{[k]} \in \mathcal{X}_{t-1}$, sample a random realization $\Delta v_{t-1}^{[k]} \sim \mathcal{N}(0, \Sigma_{t-1})$ of the process noise.
- Apply your solution in part (a) to compute the predicted next pose $x_t^{[k]}$ of the robot obtained from $x_{t-1}^{[k]}$, v_{t-1}, $\Delta v_{t-1}^{[k]}$, and Δt.

- Return the particle set $\{x_t^{[k]}\}$ of predicted particle locations.

(c) **Sensor likelihood function:** We saw in Lecture 12 that we can model the likelihood of sampling the laser range scan z_t given the robot pose $x_t \in \mathrm{SE}(2)$ and map m as:

$$p(z_t|x_t, m) = \prod_{k=1}^{N} p(z_t^k|x_t, m) \tag{1}$$

where z_t^k is the range reported for the kth laser beam, and $p(z_t^k|x_t, m)$ is the mixture model:

$$\begin{aligned} p(z_t^k|x_t, m) = {} & w_{\text{hit}} \cdot p_{\text{hit}}(z_t^k|x_t, m) + w_{\text{short}} \cdot p_{\text{short}}(z_t^k|x_t, m) \\ & + w_{\max} \cdot p_{\max}(z_t^k|x_t, m) + w_{\text{rand}} \cdot p_{\text{rand}}(z_t^k|x_t, m) \end{aligned} \tag{2}$$

(see e.g. Sec. 6.3.1 of *Probabilistic Robotics*). In (2), the scalars w_{hit}, w_{short}, $w_{\max}$ and w_{rand} are *mixture weights* (nonnegative scalars summing to 1), and the mixture components are as follows:

- $p_{\text{hit}}(\cdot|x_t, m)$ is a Gaussian distribution with mean z_t^{k*} (the *true* distance from x_t to the nearest obstacle in the map m along the direction of the beam z_t^k) and standard deviation σ_{hit}. This component models the case in which we get a "clean" laser return, so σ_{hit} should be relatively small (on the order of a few centimeters).
- $p_{\text{short}}(\cdot|x_t, m)$ is an exponential distribution with inverse scale parameter $\lambda_{\text{short}} > 0$. This component is intended to model the effect of transient "clutter" in the scene (not captured in the map), such as people walking in front of the robot, which can cause an unexpectedly short laser return.
- $p_{\max}(\cdot|x_t, m)$ is a very narrow uniform distribution centered on the maximum range $z_{\max}$ of the laser scanner. This component accounts for the probability of obtaining a maximum-range reading from the sensor due to sensor failure (e.g. the true range to the nearest obstacle is greater than the maximum sensing range, or the beam was absorbed or scattered by an obstacle).
- $p_{\text{rand}}(\cdot|x_t, m)$ is the uniform distribution over the range $[0, z_{\max}]$. This component is a catch-all component that models "general weirdness".

Given the sensor model (1)–(2), write a sensor likelihood function that accepts as input the following arguments:

- The laser scan z_t
- The current pose x_t
- The map m
- All parameters of the beam likelihood mixture model $p(z_t^k|x_t, m)$ defined in (2)

and returns:

$$\ell(z_t|x_t, m) \triangleq \sum_{k=1}^{N} \log p(z_t^k|x_t, m) \tag{3}$$

the ***log***-likelihood of the laser scan z_t given the pose x_t and map m.

Remark 1 (Numerical precision)**.** While in exact arithmetic (1) and (3) are completely equivalent, when implementing the particle filtering algorithm on *finite-precision* computers it is necessary to use the latter. The reason is that multiplying many small likelihoods together (as is required in (1)) can easily cause numerical underflow.

Remark 2 (Computing the range z_t^{k*})**.** The trickiest part of implementing the particle filter is computing the true range z_t^{k*} to the nearest object along the kth beamline given the pose x_t and map m: doing this exactly requires *raycasting* each of the (several hundred) beams z_t^k in the scan z_t through the map m, which is an expensive operation. You may want to take advantage of specialized (highly-optimized) libraries for performing this operation to speed up your code.

Alternatively, a popular *approximate* approach is to estimate the error $e_t^k \triangleq z_t^k - z_t^{k*}$ in the kth range measurement z_t^k implied by the pose x_t and map m using the Euclidean distance transform E of the occupancy grid map: in this approach, one simply calculates the cell in which the *measured* range z_t^k falls, and then takes the resulting distance assigned to that cell of E as the measurement error e_t^k (see e.g. Sec. 6.4.1 of *Probabilistic Robotics*). The advantage of this approach is that since E only depends upon the map m itself, it can be *precomputed* once (when the particle filter is started), and then cached throughout the rest of the algorithm; calculating ranging errors given z_t, x_t, and m then amounts to performing table lookups in E, which is fairly quick. If you opt to take this approach, you may want to make use of OpenCV's `distanceTransform` function to calculate E.

(d) **Resampling function:** In lines 7–10, we must resample the particle set, drawing each particle $x_t^{[l]}$ with probability proportional to its weight $w_t^{[l]}$. If we had in hand the particle weights $w_t^{[l]}$, this would be fairly straightforward: we could compute the sampling probability $p(x_t^{[l]})$ for particle $x_t^{[l]}$ as:

$$p(x_t^{[l]}) = \frac{w_t^{[l]}}{\sum_{k=1}^N w_t^{[k]}}. \tag{4}$$

However, because we computed the measurement *log*-likelihoods in (3), we cannot apply (4) directly. Fortunately, it *is* still possible to compute the sampling probabilities $p(x_t^{[l]})$ using only the *logarithms* $\ell_t^{[l]} \triangleq \log w_t^{[l]}$ of the particle weights. To see this, let LSE denote the log-sum-exp function:

$$\mathrm{LSE}(x_1, \dots, x_N) \triangleq \log\left(\sum_{k=1}^N \exp(x_k)\right), \tag{5}$$

define:

$$\bar{w}_t \triangleq \max_{1 \le k \le N} w_t^{[k]}, \qquad \bar{\ell}_t \triangleq \log \bar{w}_t, \tag{6}$$

and observe that we can equivalently rewrite (4) as:

$$\begin{aligned} p(x_t^{[l]}) &= \frac{w_t^{[l]}/\bar{w}_t}{\sum_{k=1}^N w_t^{[k]}/\bar{w}_t} \\ &= \exp\log\left(\frac{w_t^{[l]}/\bar{w}_t}{\sum_{k=1}^N w_t^{[k]}/\bar{w}_t}\right) \\ &= \exp\left(\ell_t^{[l]} - \bar{\ell}_t - \log\left(\sum_{k=1}^N w_t^{[k]}/\bar{w}_t\right)\right) \\ &= \exp\left(\ell_t^{[l]} - \bar{\ell}_t - \mathrm{LSE}(\ell_t^{[1]} - \bar{\ell}_t, \dots, \ell_t^{[N]} - \bar{\ell}_t)\right). \end{aligned} \tag{7}$$

As in the case of part (c), the advantage of (7) versus (4) is that by working (carefully) with logarithms, the former avoids the possibility of numerical underflow.

Using (7), write a function that accepts as input:

- The propagated particle set $\{x_t^{[k]}\}$
- The measurement log-likelihoods $\{\ell_t^{[k]}\}$

and returns the resampled particle set $\mathcal{X}_t$ obtained by drawing N samples from $\{x_t^{[k]}\}$ with replacement, where $x_t^{[l]}$ is drawn with probability $p(x_t^{[l]})$.

(e) **Putting it all together:** Finally, using the functions you wrote in parts (b)–(d), write a ROS node that implements the MCL algorithm (Algorithm 1). On startup, your node should do the following:

- Load in the occupancy grid map that you created in Problem 1 (possibly by receiving it from a `map_server`)
- Subscribe to laser scan and commanded velocity messages
- Initialize a particle set that is widely distributed over the map (to indicate high initial uncertainty in the robot's location)

and then run the MCL algorithm.

As in the case of Problem 2, a successful demonstration entails showing that the particle set for your MCL implementation eventually converges to a single cluster around the true robot pose.