A grid-based approach to localization for robotics applications

Larry Carlton Reaves
A Grid-Based Approach to Localization for Robotics Applications

Larry Reaves

Problem Report Submitted
to Statler College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Powsiri Klinkhachorn, Ph.D., Chair
Elaine Eschen, Ph.D.
Alan Barnes, Ph.D.
Frances L. VanScoy, Ph.D.

Department of Computer Science and Electrical Engineering

Morgantown, WV
2017

Keywords: LIDAR, Localization, SLAM, Particle Filters, Robotics
Copyright 2017 Larry Reaves
ABSTRACT

A Grid-Based Approach to Localization for Robotics Applications

Larry Reaves

From robotic probes on Mars to self driving cars here on Earth, the process of moving around the planet begins with localization. Before an autonomous vehicle can decide where to go, it must first determine where it is and what obstacles may be nearby. By using various sensors, an autonomous platform can gather the data it needs to identify its position and orientation within its surroundings. This position and orientation is known as the pose of the robot.

In this report, we will review a particle filter based algorithm for localizing an autonomous mining robot in a known 2D map. This algorithm uses input from a LIDAR sensor that provides range information for every 0.25° in a 180° arc. We will also discuss how this algorithm’s pose outputs were used to guide the autonomous operations of the Mountaineer Mining Vehicle, WVU’s 2014 winning entry to NASA’s Robotic Mining Competition. Sufficiently fast runtimes were achieved with small deviations from the correct pose. These runtimes were further improved by culling bad particles after a partial evaluation of the sensor data. This resulted in a 30-40% reduction in runtime without increasing the error.
## Contents

1 Introduction ........................................... 1  
   1.1 Background .................................... 1  
   1.2 Problem Statement .............................. 1  
       1.2.1 Vehicle ................................. 2  

2 Literature Review ................................... 4  
   2.1 Particle Filters .................................. 4  
   2.2 SLAM ........................................... 5  
       2.2.1 FastSLAM ................................. 6  
       2.2.2 DP-SLAM ................................ 7  

3 Methods ............................................. 10  
   3.1 Localization .................................... 11  
       3.1.1 Map Representation ...................... 12  
       3.1.2 Initialization ............................ 12  
       3.1.3 Motion .................................. 14  
       3.1.4 Update .................................. 16  
   3.2 Planning and Integration ...................... 18  
       3.2.1 Goals .................................. 18  
       3.2.2 Procedure ............................... 18  
       3.2.3 Integration .............................. 19  
   3.3 Culling ........................................ 20  
   3.4 Analysis ...................................... 21  

4 Results ............................................. 22  
   4.1 Runtime ...................................... 22  
   4.2 Deviation ..................................... 25  

5 Conclusions and Future Work ....................... 28  
   5.1 On-Vehicle Localization ...................... 28  
   5.2 Future Work .................................. 29  
       5.2.1 Parallelization and Culling .......... 29  
       5.2.2 Discretization ......................... 29  
       5.2.3 Drift .................................. 29  

Appendix A Source Code ............................... 32  

Appendix B Analysis Code ............................. 64
# List of Figures

1.1 Mountaineer Mining Vehicle ........................................... 3
2.1 Particle Filter for Localization ......................................... 5
2.2 SLAM to Address Skew ................................................... 6
2.3 FastSLAM ................................................................. 7
2.4 DP-SLAM ................................................................. 9
3.1 Localization Flowchart ................................................... 11
3.2 Initial Map ................................................................. 12
3.3 Robot Pose ................................................................. 13
3.4 Swarm Initialization ...................................................... 14
3.5 Localization Move Flowchart ............................................ 15
3.6 Swarm Move/Update ...................................................... 15
3.7 Localization Update Flowchart .......................................... 16
3.8 Overall State Diagram .................................................. 19
3.9 Interprocess Communication ............................................ 20
4.1 Overall Runtime .......................................................... 23
4.2 1000 Particle Runtime .................................................... 23
4.3 8000 Particle Runtime .................................................... 24
4.4 64000 Particle Runtime .................................................. 24
4.5 Angular Deviation .......................................................... 25
4.6 Positional Deviation ....................................................... 26
4.7 Angular Deviation w/Culling ............................................. 27
4.8 Positional Deviation w/Culling ........................................... 27
Chapter 1

Introduction

1.1 Background

As humanity expands its reach beyond Earth and aspires to reach other planets, there is one crucial molecule that we need: water. Luckily, on both of the two most likely destinations for human exploration, the moon and Mars, there is water available. However, this water is in the form of ice crystals mixed in with the martian or lunar soil. To extract this water, we must first collect the soil. Collecting the soil manually would be a very laborious task. Thankfully, we have the ability to bring with us equipment that can assist, or even perform the majority of the collection for us. There are many challenges in building such equipment such as making it robust enough to get the job done while minimizing the weight for transport. However, in this paper we will focus on autonomous operations. Specifically, we will be discussing WVU’s entry into the 2014 NASA Robotic Mining Competition. Our vehicle was referred to as the Mountaineer Mining Vehicle or MMV.

1.2 Problem Statement

The objective of the NASA Robotic Mining Competition is to build a robot that can mine simulated martian soil. The simulated Martian soil (also called regolith) was a compound known as BP-1. The MMV will start in a random position and orientation within a designated starting area. The robot will have to traverse an obstacle field containing three obstacles such as large rocks, and two craters. It will then collect BP-1 in a designated mining area. Finally, the MMV must traverse back across the obstacle field and deposit the collected BP-1 into a collection bin behind the starting area. The vehicle can traverse the arena again to collect more BP-1 until 10 minutes are up.
Each 10 minute collection attempt is scored as follows:

- 1000 points for passing the safety and communications check
- 0 points for the first 10 kg of BP-1 mined, 3 points for each kg thereafter
- -1 point for each 50 kb/s of average data throughput used
- -8 points for each kg of robot mass
- 20 points for reporting energy usage
- up to 100 points for dust tolerant design and dust free operation
- up to 500 points for autonomous operations
  - 50 points for only crossing the obstacle field
  - 150 points for also excavating BP-1
  - 250 points for also returning across the field and depositing BP-1
  - 500 points for operating autonomously for the entire 10 minute run

### 1.2.1 Vehicle

In order to autonomously collect soil, several hardware and software components are needed. We start with the basics that are needed even for non-autonomous operations: sensors to provide feedback to the driver and motor controls to move the vehicle and operate the soil collection mechanisms. The MMV, shown in Figure 1.1, has four wheels, each with an independently controllable motor. Each of these motors has a Hall effect sensor that provides the ability to measure how far the motor turns. Additionally, the MMV has a dump bin with two linear actuators for raising and lowering the bin. Finally, at the front of the vehicle is an excavator bucket for collecting soil and depositing it in the dump bin. The excavator bucket has a chain driven axel to raise and lower it, as well as two small linear actuators that allow for control of the bucket angle.
The non-motor sensors included a forward looking camera and a Hokuyo UTM-30LX-EW LIDAR scanner. For autonomous operations only data from the LIDAR scanner was used. Using this data, we were able to localize the MMV within the competition arena. Once the position and orientation (pose) of the MMV was known, that information was used by a planning algorithm to transition the vehicle through the various stages of autonomous operations. After starting at one end of the arena, the vehicle had to first cross an obstacle field. Once it arrived on the other end of the arena, the vehicle collected the soil using the bucket and deposited it in the dump bin. The vehicle then had to return to the starting area, line itself up with the center of the wall, and then dump the collected soil into a collection bin.
Chapter 2

Literature Review

LIDAR is a sensing technology that allows its user to accurately detect the distance between the sensor and various objects in the sensor’s field of vision. This kind of data has many applications. For example, 3D LIDAR data can be combined with 2D images to build photorealistic 3D models [1]. Aerial LIDAR scans are even used in forestry research to determine various metrics of the health and biodiversity of the trees in the scanned area [6]. Of course, LIDAR is also well studied for use in robotics applications [2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13]. Successful path planning and obstacle avoidance relies heavily on accurate sensing capabilities. While obstacle avoidance can often be accomplished by simple analysis of raw sensor data that detects objects in front of the robot, path planning necessitates accurate localization of the robot platform [7, 10, 11].

2.1 Particle Filters

A particle filter is a simple method for estimating a hidden parameter based on observable data [8]. The pool of potential values is sampled as a set of particles, where each particle represents a guess at the hidden parameter’s value. Using the observed data, the fitness of each particle is evaluated. The fitness values of each particle are normalized so that they sum to one. In this way, the fitness of a particular particle represents the probability that particle is the correct value for the hidden parameter. The particles are then resampled by using the fitness as a weight. Highly likely particles will end up as the ancestor for many particles in the next generation. After resampling, each particle is randomly mutated to further explore the state space. The new swarm of particles is again filtered with the lastest observations and the process repeats. In this manner, the hidden parameter can be accurately tracked. In a Darwinian competition, only the fittest particles survive [11].
Particle filters have been used for localization within a known map. Figure 2.1 shows a particle swarm attempting to localize within a known map using very low resolution observations. Initially, the swarm is very spread out and the pose uncertainty is high. But after moving around and gathering more data about its environment, the robot is localized.

![Figure 2.1: As the robot moves around and gathers more data, the location within the static map is refined until the uncertainty is fairly low [11]. The actual position is labeled.](image)

2.2 SLAM

In addition to localization within a known map, the problem can be expanded to include localization relative to a dynamically generated map. In a particle filter based SLAM algorithm, each particle represents no only the current pose, but also some measure of previous observations assumed to originate from previous poses
This allows the algorithm to recover from small errors in a single pose. As shown in Figure 2.2, such an approach can help to address the skewing issues seen when building maps without evaluating the full path history.

Figure 2.2: Utilizing path information instead of just pose information can address skew when building maps.

The problem of simultaneous localization and mapping (SLAM) has been widely investigated over the past few decades. Some approaches utilize distinct landmarks to define the map of the robot’s environment [5, 8, 10, 11]. These methods work well in outdoor environments where there are generally sparse, distinct objects that can be identified. Other approaches consider grid-based maps [2, 3, 4]. Grid based maps work better for indoor environments where complex features such as walls and doorways are not modeled well using the circles or 2D gaussians common in landmark based approaches. For these grid based maps, the filter is over estimated submaps with an associated pose estimation. These maps are stitched together and evaluated for consistency with the current sensor data to weight the particle. While many approaches can be used to represent maps and to localize the LIDAR readings within the map, all approaches provide some description of the robot’s environment, as well as the vehicle’s position and orientation within that environment.

2.2.1 FastSLAM

FastSLAM [8] is possibly the best known of these approaches. As seen in Figure 2.3, FastSLAM is a particle filter over estimated landmark positions and current pose. FastSLAM uses landmark based maps to represent distinct obstacles, so it is well suited for outdoor environments.
FastSLAM uses particles consisting of the current pose and estimates for all observed landmarks [8]. Since many particles can have the same ancestor, a tree structure with reference counting is used to store landmarks in a way that allows sharing partial trees until a child particle needs to update it. This copy-on-write structure greatly reduces the memory requirements. For each potential path, the particle is scored by comparing the sensor data at the current position to the map that would be built by combining the observations from previous points along the path. As with all particle filters, the particles are resampled with weighting determined by their score.

FastSLAM has also been extended as FastSLAM 2.0 by adding an Extended Kalman Filter update step [8]. This EKF update is applied to all potential particles before they are evaluated. By improving each particle, diversity of the entire swarm is better preserved, increasing the robustness of the algorithm. However, FastSLAM represents the robot environment with a set of landmarks. While this representation is excellent in scenarios where the environment contains many separate landmarks, it is not ideal for high accuracy localization within an arena with four straight walls, or any environment that isn’t sparsely populated with distinguishable landmarks. Since such an area is the primary target environment, this leads to a preference for a grid based representation of the map.

### 2.2.2 DP-SLAM

The Distributed Particle SLAM family of algorithms (DP-SLAM) is one successful approach that utilizes grid based map representations. Like FastSLAM, DP-SLAM maintains a map estimate for each individual particle. Because grid based maps are large and many of the map estimates will contain much the same data, DP-SLAM uses careful accounting to share as much data between particles as possible [2]. By using a single global occupancy grid where each grid space contains a balanced tree of observations and an ancestry tree...
of particles, DP-SLAM achieves a bound that does not grow with the iteration number. Each observation is inserted by a single particle at a single iteration and the tree is sorted by a particle id number. This allows $O(ADP \log(P))$ localization where $A$ is the area that can be seen by the sensor, $D$ is the depth of the ancestry tree, and $P$ is the number of particles in the swarm. The pruning methods employed on the ancestry tree keep $D$ close to $\log(P)$ in practice, although it can theoretically be as high as $P$ in high uncertainty environments.

DP-SLAM 2.0 extends the capabilities of the DP-SLAM methodology by modeling the map using an opacity model and by improving the bound on localization to $O(AP^2)$ in the worst case of $D = P$. For the opacity model, they trace each laser cast through the map, to determine two parameters for each grid square the cast traverses. First, how far the laser travels through the grid square is tracked. Depending on the angle and where it enters the square, it can travel anywhere from 0 to $gs\sqrt{2}$ units where $gs$ is the length (or width) of a grid square. Additionally, if the laser cast stops in a grid square this is also marked. This gives an opacity estimator of $o = \frac{\text{stop}}{d_{\text{total}}}$ or the number of laser casts which stopped in this square divided by the total distance traveled by all laser casts that passed through this square.

To improve the localization time, instead of walking back up the ancestry tree for every particle to find the the most recent ancestor for a given particle, the ancestor tree is walked from the top down to process this information for all particles. Finally, the bound for localization is lowered to $O(AP)$ by creating a $O(1)$ access time cache for each particle. This cache can be created by two $O(AP)$ passes, one over the global grid, and one over the ancestry tree [4]. Utilizing this method very high resolution maps, such as Figure 2.4 can be built with no prior knowledge of the environment. To build this map, a robot was manually driven in a loop around the building.
Figure 2.4: Highly accurate maps can be built using grid based methods when filtering is done over submaps such as in DP-SLAM [4]
Chapter 3

Methods

In the field of robotics, one crucial task for autonomously moving around in the world is localization. This is the task of figuring out where exactly the autonomous platform is located relative to its surroundings. In our approach to localization, a Hokuyo UTM-30LX-EW scanning laser range finder was used as sensor input. This sensor is capable of specifying the range to the nearest solid object every 0.25° in a 270° arc. Using this sensor data, the problem of localization in a real world robotic mining competition using a predefined map was successfully addressed. While this localization was accurate enough to enable path planning for the autonomous robot operations, it was not robust enough to be able to generate accurate maps of the environment due to issues with skew. The exact methods used on the competition robot are described in Section 3.1 below.

Additionally, this pose information was used in an autonomous planning algorithm. The goal of the algorithm was to traverse the mining arena, collect simulated regolith, return to the opposite end of the arena, and deposit the material into a collection bin. At each stage of this process, the pose was used to determine how to move next, and at what point the next phase of the plan could be enacted. We will describe the planning algorithm in detail in Section 3.2.

While the pose estimation algorithm was robust enough to feed the autonomous planning algorithm and successfully complete the automated mining task, it was not robust enough to achieve full SLAM. When attempting to generate new maps based on projecting the distance data from the pose estimates, we experienced severe skewing issues that built up over time. However, we were able to use the data to refine the map used for navigation in the competition arena. While the initial map was a perfect rectangle, and the arena was built very exactly, our refined maps did contain a notch in one corner. Inspection of the arena
showed that this corner did indeed have a piece of wood that protruded into the arena.

### 3.1 Localization

Because sensor data can be noisy and kinematics models will fail to fully account for wheel slippage, a particle filter approach was used to tackle the task of localization. A swarm of particles is initialized (Section 3.1.2), moved according to a kinematics estimate from the robotics platform (Section 3.1.3), then compared with LIDAR data (Section 3.1.4) to determine the best pose estimate for the sensor in a known map. The map used is a high resolution grid-based map, described in Section 3.1.1. This position is then used in the goal based planning, a motor activation is performed, and the resulting kinematics estimate is passed into the particle filter. After another sensor update and evaluation, the loop continues until autonomous operations end. Figure 3.1 illustrates this overall procedure.

![Figure 3.1: Robot-Integrated Localization Flowchart](image-url)
3.1.1 Map Representation

For simple localization in an approximately known map, a simple and fast map provides quick lookups for fast evaluation of many potential particles. Maps are stored as a rectangular grid of locations. Each location in the grid holds a count of the number of times that particular location was seen or unseen. A location is seen when a sensor distance is projected from a particle into that location. A location is unseen when it lies between the sensor and the seen location. Because the sensor detected something beyond this point, we can assume there was empty space between the sensor and the object that generated the return. For simple localization in a roughly rectangular area, a simple rectangular border can be used for the initial map, as seen in Figure 3.2.

![Figure 3.2: This initial map shows a probability gradient between black and white where black represents empty space (unseen) and white represents filled space (seen)](image)

3.1.2 Initialization

In our particle filter, each particle represents a single pose for the LIDAR sensor, which is defined as the sensor position and orientation. This is simplified to the 2D case since the robot can only move along the floor and any small change in elevation due to terrain should not change the walls the robot sees. Within the rectangular initial map we have our position variables, $x$ and $y$, and our orientation angle $\theta$, as shown in Figure 3.3.
The initial particle swarm is distributed to match the rules of the NASA Robotic Mining Competition as shown in Figure 3.4. Initially, the vehicle is centered at one of two possible locations in the starting area of the arena (the left side in the top-down views). Additionally, the vehicle’s orientation will be chosen randomly. In order to thoroughly cover the possible sensor starting positions, the initial particles must be distributed around two circles. The initial location of each particle is chosen by first randomly choosing one of the two possible starting positions for the vehicle center. A random orientation is then chosen for the particle. Using the location and orientation for the particular particle, the distance between the center of the robot and the sensor is used to determine the approximate particle location. For certain orientations, it can be impossible to determine from the initial scan which of the possible locations the robot is in because it can not see far enough along the longer wall to differentiate it from the smaller wall. For these situations, the robot has to rotate slowly and collect observations until it is able to get enough information to determine its actual position.
3.1.3 Motion

After the autonomy controller uses the current pose to make a decision and move the vehicle, it gives the localization algorithm an estimated motion vector. This vector specifies the expected change in $x$, $y$, and $\theta$ for the robot platform, as calculated from the skid-steer kinematics model. The motion vector describes the change in pose relative to the center of the robot, but the particle pose is relative to the sensor. The motion vector is adjusted to account for this using the distance from the center of the robot to the sensor. Since the skid-steer model will likely have accuracy issues due to wheel slippage and imperfect Hall effect sensor data, not all particles have the motion vector applied. All particles receive a normally distributed random perturbation for each of $x$, $y$, and $\theta$. By using this procedure (Figure 3.5), the swarm covers the potential positions of the sensor and is ready for evaluation and resampling. The left side of Figure 3.6 shows the state of the swarm after the application of the motion vector and random perturbations.
Figure 3.5: Motion procedure that will be applied to each particle

Figure 3.6: The particle swarm is shown before and after the sensor data is evaluated. A shows the initial swarm covering the potential starting positions. B shows the next step after the swarm is resampled and the move command is applied.
3.1.4 Update

The update step of the localization process is broken down in several steps. As seen in Figure 3.7, the incoming observation is first used to evaluate each particle, then the swarm is normalized and sorted. The standard deviation of the swarm is calculated to check for convergence, and then the entire swarm is resampled based on the particle weighting. First we will discuss the evaluation procedure used to evaluate each particle's fitness, including the use of log probabilities as well as an explanation of the normalization procedure used to ensure that all particles' probabilities sum to 1. Next, we will review the convergence criteria used to check that we have some certainty about our position. Finally, we will explain how the resampling is performed.

Figure 3.7: Localization update procedure

3.1.4.1 Evaluation

The particle filter proceeds by evaluating each particle's fitness. It does this by comparing the LIDAR range data to an approximately known map, assuming the pose of the given particle is accurate. Using the landmark map for comparison, each direction of the LIDAR data is projected from the current particle and the probability it is unseen between the sensor and the distance reading is calculated. This is calculated by sampling the \( \text{unseen} \) probability of the map at several locations between the sensor and the distance at which an obstacle was detected. This probability is multiplied together with the probability that the map should be seen at the correct distance. The \( \text{unseen} \) probabilities are calculated for a given landmark as \( \frac{\text{unseen}}{\text{seen}+\text{unseen}} \) and \( \text{seen} \) probabilities are calculated as \( 1 - \text{unseen} \). \( \text{Unseen} \) probabilities are bounded between 0.05 and 0.95 so that neither probability is ever 0. The fitness of all particles is normalized to create probabilities. To avoid underflowing the floating point number representing the probabilities, log probabilities were used. Instead of multiplying all the calculated probabilities together, the logarithm of the probability is calculated, and the negative of this result is added to the posterior we are calculating. The prior probability for a given
particle is also incorporated into the score for each particle. The scores of the particles are normalized in three steps. The log probabilities are first normalized by subtracting the smallest value from all of the log probabilities, making the log probability of the most likely particle equal to 0, which represents a probability of 1. Next, the log probabilities are converted back to regular probabilities via exponentiation. Finally, the probability of each particle is divided by the sum of all particle probabilities. This normalization makes it so that the scores sum to 1 and the individual scores represent the probability that a given particle is the best choice. After normalization, the particles are ordered by probability using quicksort with a pivot chosen from the center of the list. This choice of pivot ensures we have fast sorting times with nearly sorted lists, which becomes important when using culling (Section 3.3).

3.1.4.2 Convergence

Getting lost in the arena would lead to incorrect inputs for path planning. After evaluating the particles, but before resampling, the standard deviations for \((x, y, \theta)\) are calculated for the current particles. These deviations are checked to make sure the particles cluster together tightly. After resampling (Section 3.1.4.3), the higher level algorithm can check this convergence flag. If the particles have converged, the algorithm can proceed to path planning. If the particles haven’t converged, the swarm is “moved” with a kinematics estimate of \((0, 0, 0)\) and the update process is run again.

3.1.4.3 Resampling

Those particles with higher probabilities should be chosen more often than those with lower probabilities. In order to resample, first a random number \(p\) between 0 and 1 is chosen. The fitness of the individual particles is summed, starting with the highest probability particle, until the total is larger than the chosen random number \(p\). Resampling continues by increasing \(p\) by \(\frac{1}{N}\) where \(N\) is the number of particles. This new \(p\) is compared to the total fitness and the appropriate particle is chosen as the first particle which causes the sum to exceed \(p\). If \(n\) becomes greater than 1, it is reduced by 1 to keep the sum bounded between 0 and 1 and the resampling proceeds. The right side of Figure 3.6 shows the state of the swarm after it has been resampled. At this point, it only contains particles that closely match the sensor data. The highest probability particle is saved, projected to the robot’s center, and made available to the autonomy controller as the current pose.
3.2 Planning and Integration

3.2.1 Goals

The high level autonomy planning was centered around a set of goals as laid out by the rules of the competition. The first objective was to traverse the obstacle field. We were able to define a lane for the robot to use to traverse the obstacle field. Because of the robust design of the Mountaineer Mining Vehicle (MMV), we were able to use the center of the arena as our lane without scouting for obstacles ahead of time. The next objective was to collect regolith. This objective was accomplished by fully lowering the bucket, driving forward a small amount, and then raising the bucket to deposit the regolith into the onboard dump bin. The final objective was to return to the starting area and deposit the regolith in the collection bin. Traversing back across the obstacle field was done by driving the robot in reverse down the chosen lane. Lining up with the collection bin required fairly accurate positioning of the MMV. The collection bin was centered at the edge of the arena. Because the MMV’s dump bin was not much narrower than the collection bin, the MMV had to be square and centered before it could make the deposit.

3.2.2 Procedure

Once initialized, the first step of the autonomy routine is to localize the robot’s initial position and to make sure all accessories are in the appropriate position. After determining the initial location and raising the digging bucket, the TraverseClearPath state is activated. The purpose of this state was to traverse the arena in a chosen lane. Because the vehicle was able to cross pretty much anywhere, the center was usually chosen because it would put us closest to where we would need to be when depositing the regolith in the collection bin. To accomplish the traversal, a simple algorithm was followed:

\[
\text{while } x_r < x_{\text{mining-area}} \text{ do}
\]

\[
\text{if } y_r > y_{\text{lane-top}} \text{ then}
\]

\[
\text{turn to } -20^\circ
\]

\[
\text{else if } y_r < y_{\text{lane-bottom}} \text{ then}
\]

\[
\text{turn to } 20^\circ
\]

\[
\text{else}
\]

\[
\text{turn to } 0^\circ
\]

\[
\text{end if}
\]

\[
\text{move forward } \min(300 \text{ mm, distance to mining area})
\]
Once the mining area is reached, the autonomy moves to the MineRegolith state. In the MineRegolith state, the vehicle lowers its digging bucket and then drives forward a small amount. Next, it raises the bucket all the way to deposit the collected regolith in the onboard dump bin. After this step is complete, the routine moves on to the ReturnToDeposition state. This state is simply the reverse of the TraverseClearPath state. The vehicle reverses across the arena within the appropriate lane until it is within the starting area. The next state to become active is the DepositRegolith state. The first phase of this state is responsible for ensuring the robot’s y position is centered to align with the collection bin. If necessary, the vehicle turns to either 90° or −90° as appropriate and then moves backward the distance it needs to be aligned. After rotating back to a 0° orientation, the robot moves backwards until it is seated against the wall. At this point, the dump bin actuators are fully raised to deposit the regolith, and then fully lowered to return the bin to the lowered position. If mining is to continue, the algorithm returns to the TraverseClearPath state.

The overall state diagram for the robot can be found in Figure 3.8.

![State Diagram](image)

Figure 3.8: The goal states and the transitions between them are shown.

### 3.2.3 Integration

The majority of the software for the Mountaineer Mining Vehicle is written in C#. This included the remote control software, the autonomy algorithm, and the motor control. The localization algorithm is written in C. In order to integrate the two, a daemon was written to handle the localization. The C# side of the software communicates with this daemon to send motion estimates, pass sensor values, check convergence, and retrieve the current best estimate. The communication is performed using a combination of shared memory segments to pass data and semaphores for flow control. The state diagram for this communication shown in Figure 3.9 illustrates how the C# code makes a function call into the C code.
Figure 3.9: Shared memory was used to communicate between the particle filter daemon and the robot’s planning and control processes. This figure shows a single method call to the C daemon.

There were 7 functions exposed to the C# code that would allow interaction with the daemon. *swarm_init* is the first function that is called. As the name implies, *swarm_init* initialized the particle swarm. *swarm_move* is the function used to pass a kinematics-derived motion estimate to the daemon. *swarm_update* is used to pass a set of sensor reading to the swarm. This function kicks off the next iteration of the particle filter, but returns as soon as it has read off the sensor data. *swarm_converged* is a flag used to test if the swarm has converged on a location yet. Once *swarm_converged* returns true, *swarm_get_best_x*, *swarm_get_best_y*, and *swarm_get_best_theta* are called to retrieve the updated localization.

### 3.3 Culling

One optimization that was not utilized during the competition, but shows promise when it was explored later was particle culling. This is the idea that rather than fully evaluating each particle’s fitness, we first sample a few of the readings at various angles. After processing a percentage of the available angles, many of the particles are clearly not going to be accurate and we can avoid evaluating the remaining angles. A variable called the *culling factor* was used to control the level of culling. When evaluating each particle, we would use data from $\frac{m}{\text{culling factor}}$ angles where $m$ is the total number of angles with available distance measurements. A culling factor of 1 indicated no culling, and a culling factor of 5 meant that $\frac{1}{5}$ of the data
was evaluated before each cull. The data used would be evenly spaced so as to cover the entire range of angles. After evaluating this subset of the data, the particles were sorted by probability and the bottom $\frac{0.9 \times \text{particle count}}{\text{culling factor}}$ particles were removed from evaluation. Finally, the next $\frac{m}{\text{culling factor}}$ angles are considered and the process repeats until all data has been evaluated. In this way, only 10% of the total particles are fully evaluated.

### 3.4 Analysis

When evaluating the algorithm, 200 iterations were run for each set of parameters, using recorded sensor data from the competition arena. Each run produced a file listing a series of 25 poses. For each of these poses, a mean and standard deviation was caculated for $x$, $y$, and $\theta$ across all 200 runs. This gave an average pose and standard deviation for each of the 25 poses the robot sampled data at. The deviations for $x$ and $y$ were averaged together for all poses and termed as *positional deviation*. The deviations for $\theta$ were average together and termed as *angular deviation*.

Samples were collected with a culling factor of 1, 5, 10, and 20 for each of the following particle counts: 1000, 2000, 4000, 8000, 16000, 32000, 64000. For particle counts 1000 and 8000, data was collected for culling factors ranging from 5-80 to explore if the effect of the culling factor varies with different particle counts. Finally, for 64000 particles, culling factors ranging from 5-20 were also sampled.
Chapter 4

Results

While increasing particle counts result in much lower positional and angular deviation, it comes at the cost of a linear increase in runtime. While this linear association of runtime with particle count is unavoidable, utilizing culling can reduce the slope of the runtime growth. This is accomplished without negatively impacting either the positional or angular deviation measures.

4.1 Runtime

In Figure 4.1, we can see the impact on runtime of various particle counts and culling factors. There is a linear increase of runtime with increasing particle counts. However, even a low culling factor can improve the runtime significantly. With the highest particle count of 64000, a culling factor of 20 is 25516 ms faster on average than a without culling. This is a 40.8% reduction in runtime. Similarly, for the runs with 1000 particles, there is a 20.8% reduction in runtime going from culling factor 1 to 20, and a 37.8% reduction comparing the best culling factor of 25 with no culling (see Figure 4.2).
Figure 4.1: Average runtime for localization with various particle counts and culling factors.

Figure 4.2: Average runtime for localization with 1000 particles and various culling factors.
Figure 4.3: Average runtime for localization with 8000 particles and various culling factors.

Figure 4.4: Average runtime for localization with 64000 particles and various culling factors.
As can be seen in Figures 4.2 - 4.4, all three particle counts where a deep sampling of culling factors
was performed show a sharp drop off in runtime with fairly low culling factors. The majority of the benefit
arises from a culling factor as low as 5. Higher culling factors do continue to marginally reduce runtime up
to a point. However, because culling involves sorting the particles, eventually the \( n \log n \) behavior of the
sort can start to overpower the linear scaling inherent to the particle evaluation. For each particle count,
there is an optimal culling factor, after which higher culling factors increase runtime. Using a culling factor
of around 10 seems to work well for particle counts between 1000 and 64000. Higher culling factors only
provide marginal reductions in runtime.

4.2 Deviation

Decreasing runtime is good, but we need to be certain we are not negatively impacting the accuracy of the
localization algorithm by adding culling. First, we examine how positional and angular deviation respond to
changes in particle count. In Figure 4.5, we can see that the angular deviation reduces with the particle count
for culling factors 1, 5, and 20. However, even with very high particle counts, there was usually at least one
sample out of 200 where the algorithm gave a series of poses that was off by 90°. Even one rotated sample
causes an overall standard deviation around 10°. A mechanism for reducing the probability of particles that
vary too much from the expected motion could potentially help reduce the occurrence of such rotated poses.

![Angular Deviation vs Particle Count](image.png)

Figure 4.5: Deviation from the average pose's \( \theta \) component.
Figure 4.6 shows a similar reduction in positional deviation as particle counts increase. Without culling, 1000 particles yields a positional deviation of 58.2 centimeters and that decreases to a deviation of 5.9 cm for 64000 particles. For both angular and positional deviation, the reduction in deviation slows down as the particle counts get very high. Once there is good coverage of the potential pose space, the algorithm begins to run into its limitations, such as the 2 cm grid size and the 10 cm wide border. Additionally, there is potential error in the sensor data due to noise and a swinging gimbal. The wide border could potentially be addressed by allowing the algorithm to refine the map as it gathers data.

Figure 4.6: Deviation from the average pose’s positional components.

Figures 4.7 and 4.8 show that angular and positional deviation are not negatively influenced by culling factor. This holds true across the entire range of particle counts that were sampled.
Figure 4.7: Deviation from the average pose’s positional components across various culling factors.

Figure 4.8: Deviation from the average pose’s positional components across various culling factors.
Chapter 5

Conclusions and Future Work

Particle filters are an effective means of localizing a robot in a known, grid-based map. The localization only form of the algorithm was successfully used at the 2014 NASA RMC to provide pose input for higher level autonomy planning that accomplished arena traversal as well as automated mining. Utilizing culling can enable us to use 30-40% more particles for the same runtime cost. More particles yields a lower angular and positional deviation, decreasing the likelihood of a bad output pose.

5.1 On-Vehicle Localization

On-vehicle localization was tested in three environments. The RMC arena is specified to be 738 cm by 388 cm. First, it was tested at a test arena located at WVU. This arena was nearly the correct size, but one wall had a major bow in it and was between 14 and 21 cm away from the correct dimensions. Next, it was also tested in a makeshift arena constructed from wooden stakes and plastic. This arena was used for testing while we were at the competition, and while it was fairly close to the correct dimensions at the corners and the stakes were in a straight line, the plastic blew in the wind, causing noisy readings. Finally, the localization was used in the competition arena. This arena was pretty much exactly to the correct dimensions, with the exception of a small piece of wood in one corner that is visible on some maps reproduced from sensor data. The localization algorithm worked quite well in all three scenarios, demonstrating that it is able to adapt to environments that do not exactly match its assumed map.
5.2 Future Work

A number of improvements could be made to the localization algorithm. The localization algorithm has fairly high discretization error and drift is a concern when allowing the algorithm to update its internal map. Specifically in the case of the Mountaineer Mining Vehicle, there is a case for preferring 3D maps because the sensor quite frequently has large deviations from level. The sensor was mounted on a gimbal which did ensure that it eventually settled to level, but every time the vehicle moved the sensor would swing. More dampening on the gimbal could possibly have addressed this issue without needing to complicate the localization algorithm. Such an extension would double the dimensionality of our pose space from 3 degrees of freedom to 6, so a large number of particles would be needed to sufficiently cover the state space.

5.2.1 Parallelization and Culling

The most obvious way to improve runtime is with parallel evaluation of the particles. It should be possible to accomplish parallel evaluation with very little synchronization overhead since each evaluation is independent. Culling would increase this synchronization overhead since there would need to be a sort across all particles, so there will be a tradeoff space to explore. Additionally, more aggressive approaches to culling could be considered to further reduce the evaluation time for each particle.

5.2.2 Discretization

A more accurate model of the maps could be accomplished by incorporating the opacity model from DP-SLAM 2.0 [3]. The current seen and unseen counts that indicate whether or not the location was hit or passed through by the laser actually provide similar information, but the process of tracing the laser casts is not as fine grained. By reducing the granularity of the distance steps in the laser tracing process, an equivalent opacity can be calculated as \( \frac{\text{unseen}}{\text{seen} + \text{unseen}} \). Alternatively, a finer model could be used that can better fit the size and shape of the portion of the obstacle present in that grid space. A mean and standard deviation for x and y (a 2D Gaussian) would provide decent coverage of possible shapes. Adding a rotation angle would improve that even further, but each additional parameter adds additional cost to computing the probabilities used in localization.

5.2.3 Drift

The problem of drift is present with all SLAM algorithms. Thankfully, it can be corrected for with a hierarchical particle filter in much the same way as robot motion is corrected for. The current algorithm can optionally build a single map that is shared across all particles. Only the particle with the highest fitness
at each iteration contributes to refining the map. This works well as long as the best particle has a very accurate pose. One means of addressing this problem of drift is by maintaining a set of smaller maps for each potential particle under consideration. It is possible to stitch these submaps together using a similar particle swarm, similar to what is done in DP-SLAM [2]. Rather than a motion vector, the difference in assumed poses is used and the overlapping areas can be evaluated for consistency to weight the particles.
Bibliography


Appendix A

Source Code

Listing A.1: boxmuller.h

```c
#ifndef _BOXMULLER_
#define _BOXMULLER_

#include <math.h>

extern float ranf(); /* ranf() is uniform in 0..1 */

float box_muller(float, float);

#endif
```

Listing A.2: buffer.h

```c
#ifndef _BUFFER_H_
#define _BUFFER_H_

#include <math.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include "const.h"
#include "particle.h"

void buffer_set_arena_size(int, int);
int buffer_get_size();
int buffer_get_width();
int buffer_get_height();
uint8_t* buffer_allocate();
void buffer_attenuate(uint8_t*, double);
int buffer_index_from_x_y(int, int);
int x_from_buffer_index(int);
int y_from_buffer_index(int);
int index_protected(int);
int x_y_protected(int, int);
```
Listing A.3: const.h.template

#ifndef
#define _CONST_H_

// This file is used to produce const.h by vary_params.pl

#define PARTICLE_COUNT particle_count
#define INITIAL_POSITION_VARIANCE 150
#define INITIAL_ANGLE_VARIANCE 5
#define INITIAL_PARTICLE_FACTOR initial_particle_factor
#define CULLING_FACTOR culling_factor
#define CULLING_PERCENT 0.9
#define BUFFER_FACTOR 20
#define BORDER_WIDTH 100
#endif

Listing A.4: landmark.h

#ifndef
#define _LANDMARK_H_
#define _LANDMARK_H_

#include <assert.h>
#include <stdint.h>

#include "const.h"
#include "particle.h"

landmark_map* landmark_map_copy(landmark_map*);
void landmark_map_free(landmark_map*);
void landmark_map_reference(landmark_map*);
void landmark_map_dereference(landmark_map*);
landmark_map* landmark_map_init(int);
void landmark_set_seen(landmark_map*, int);
void landmark_set_seen_value(landmark_map*, int, int);
void landmark_set_unseen(landmark_map*, int);
void landmark_set_unseen_value(landmark_map*, int, int);
void landmark_write_map(landmark_map*, uint8_t*);
void landmark_write_map_subtree(landmark_map*, uint8_t*);
double landmark_seen_probability(landmark_map*, int);
double landmark_unseen_probability(landmark_map*, int);
int landmark_map_find_distance(landmark_map*, int, particle);
void landmark_map_simulate_scan(particle, int*, int);

#endif

Listing A.5: landmark_types.h

#ifndef
#define _LANDMARK_TYPES_H_
#define _LANDMARK_TYPES_H_

#include "const.h"

#include "const.h"

33
typedef struct landmark {
    unsigned int seen;
    unsigned int unseen;
} landmark;

typedef struct landmark_map {
    int references;
    landmark *map;
} landmark_map;

#endif

Listing A.6: lazygl.h

#ifndef LAZYGL_H
#define LAZYGL_H

#define GL_GLEXT_PROTOTYPES
#include <GL/glut.h>
#include <math.h>

void initGL(uint8_t*, uint8_t*, int, int, int, int);
void display();

extern uint8_t *buffer1, *buffer2;
extern int buffer_width, buffer_height, window_width, window_height;

#endif

Listing A.7: particle.h

#ifndef PARTICLE_H
#define PARTICLE_H

#include "landmark_types.h"

typedef struct particle {
    double p;
    int x;
    int y;
    int theta;
    double cos;
    double sin;
    double x_var;
    double y_var;
    double theta_var;
    landmark_map *map;
} particle;

#include "swarm.h"
#include "landmark.h"

particle particle_sample_motion(particle, int, int, int);
particle particle_sample_normal(particle);


define _RANDOM_H_
#include <stdlib.h>
#include <stdint.h>
#endif
#include <Windows.h>
#include <time.h>
#endif
#ifdef LINUX
#include <sys/time.h>

#include "boxmuller.h"

float ranf();
int rand_limit(int);
double rand_normal(int);
void rand_normal_init();
uint64_t utime();
#endif

#define _REPLAY_
#define _REPLAY_

void save_map();
void update_display();
#endif

#define _SENSOR_H_
#define _SENSOR_H_

#include "assert.h"
#include "pthread.h"
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

#include "random.h"
#include "urg_driver/urg_utils.h"
#include "urg_driver/urg_sensor.h"
#include "scip/scipUtil.h"
#define SENSOR_MIN 20
#define SENSOR_MAX_ETH 60000  
#define SENSOR_MAX_USB 5600  
#define RAW_SENSOR_DISTANCES_ETH 1081  
#define RAW_SENSOR_DISTANCES_USB 681  
#define MAX_SCANS 30  
#define SENSOR_RANGE_ETH 270.0  
#define SENSOR_RANGE_USB 240.0  
#define SENSOR_SPACING_ETH (SENSOR_RANGE_ETH/RAW_SENSOR_DISTANCES_ETH)  
#define SENSOR_SPACING_USB (SENSOR_RANGE_USB/RAW_SENSOR_DISTANCES_USB)  

// use max for all sensors  
typedef struct raw_sensor_scan  
{  
    int distances[RAW_SENSOR_DISTANCES_ETH];  
} raw_sensor_scan;  

pthread_t sensor_init_thread();  
void* sensor_init(void*);  
raw_sensor_scan sensor_read_raw();  
void* sensor_read_raw_n(void*);  
pthread_t sensor_read_raw_n_thread(int);  
raw_sensor_scan sensor_fetch_index(int);  

#endif  

Listing A.11: slamd.h  

#ifndef _SLAMD_H_  
#define _SLAMD_H_  

#ifndef LINUX  
#include <windows.h>  
#include <stdio.h>  
#include <tchar.h>  
#include "swarm.h"  

static LPCWSTR param_shm_name = L"slamd_parameters";  
static LPCWSTR return_shm_name = L"slamd_return";  
static LPCWSTR param_sem_name = L"slamd_parameters_sem";  
static LPCWSTR return_sem_name = L"slamd_return_sem";  
static LPCWSTR ready_sem_name = L"slamd_ready_sem";  
static HANDLE param_sem;  
static HANDLE return_sem;  
static HANDLE ready_sem;  
static HANDLE param_handle;  
static HANDLE return_handle;  
static int *params;  
static int *return_value;  

#endif  

typedef enum {SLAMD_INIT = 0,  
              SLAMD_MOVE = 1,  
...
SLAMD_UPDATE = 2,
SLAMD_X = 3,
SLAMD_Y = 4,
SLAMD_THETA = 5,
SLAMD_MAP = 6,
SLAMD_CONVERGED = 7} slamd_method;

Listing A.12: slam.h

#ifndef SLAM_H
#define SLAM_H
#include <math.h>
#include <pthread.h>
#include "buffer.h"
#include "landmark.h"
#include "particle.h"
#include "random.h"
#include "sensor.h"
#include "swarm.h"

void record_map_position(int, int, int, uint8_t);
#endif

Listing A.13: swarm.h

#ifndef _SWARM_H_
#define _SWARM_H_

#include <string.h>
define _USE_MATH_DEFINES
#include <math.h>

#include "const.h"
#include "particle.h"
#include "random.h"
#include "buffer.h"

#ifndef LINUX
_declspec(dllexport) void swarm_init(int, int, int, int, int); 
_declspec(dllexport) void swarm_move(int, int, int);
_declspec(dllexport) void swarm_update(int*);
_declspec(dllexport) void swarm_update_finalize();
_declspec(dllexport) void swarm_map(int*);
_declspec(dllexport) int swarm_converged();
_declspec(dllexport) int swarm_get_best_x();
_declspec(dllexport) int swarm_get_best_y();
_declspec(dllexport) int swarm_get_best_theta();
void swarm_init_internal(int, int, int, int, int);
void swarm_move_internal(int, int, int);
void swarm_update_internal(int*);
Listing A.14: boxmuller.c

/* boxmuller.c Implements the Polar form of the Box–Muller
Transformation (c) Copyright 1994, Everett F. Carter Jr.
Permission is granted by the author to use this software for any application provided this
copyright notice is preserved.*/

#include "boxmuller.h"

float box_muller(float m, float s) /* normal random variate generator */ /* mean m, standard deviation s */
{
        float x1, x2, w, y1;
        static float y2;
        static int use_last = 0;

        if (use_last) /* use value from previous call */
        {
                y1 = y2;
                use_last = 0;
        }
        else
{ do {
   x1 = 2.0 * ranf() - 1.0;
   x2 = 2.0 * ranf() - 1.0;
   w = x1 * x1 + x2 * x2;
   while ( w >= 1.0 ) ;

   w = sqrt( (-2.0 * log( w )) / w ) ;
   y1 = x1 * w;
   y2 = x2 * w;
   use_last = 1;

   } return( m + y1 * s );

Listing A.15: buffer.c

#include "buffer.h"
#include <assert.h>

static int buffer_size, buffer_width, buffer_height, arena_width, arena_height ;

void buffer_set_arena_size(int width_in, int height_in) {
    arena_width = width_in;
    arena_height = height_in;
    buffer_width = arena_width/BUFFER_FACTOR;
    buffer_height = arena_height/BUFFER_FACTOR;
    buffer_size = buffer_width*buffer_height ;
}

int buffer_get_size() {
    return buffer_size;
}

int buffer_get_width() {
    return buffer_width;
}

int buffer_get_height() {
    return buffer_height;
}

uint8_t* buffer_allocate() {
    uint8_t *buffer = malloc(sizeof(uint8_t)*buffer_size);
    memset(buffer, '\0', sizeof(uint8_t)*buffer_size);
    return buffer;
}

void buffer_attenuate(uint8_t *buffer, double factor) {
    int i ;
    for ( i = 0; i < buffer_size; i++)
        buffer[i] *= 0.75;
}
int buffer_index_from_x_y(int x, int y) {
    int i_x, i_y;
    i_x = x/BUFFER_FACTOR;
    i_y = y/BUFFER_FACTOR;
    if (i_x >= 0 && i_x < buffer_width && i_y >= 0 && i_y < buffer_height)
        return i_y*buffer_width + i_x;
    else assert(0);
}

int x_from_buffer_index(int index) {
    return (index % buffer_width)*BUFFER_FACTOR;
}

int y_from_buffer_index(int index) {
    return (index / buffer_width)*BUFFER_FACTOR;
}

int index_protected(int index) {
    int x, y;
    x = x_from_buffer_index(index);
    y = y_from_buffer_index(index);
    return x_y_protected(x, y);
}

// returns 1 if the position is "protected"
// protected pixels are the border pixels
// returns 0 if the position is not protected
int x_y_protected(int x, int y) {
    int protected = 0;
    if (x < BORDER_WIDTH || x > arena_width - BORDER_WIDTH ||
        y < BORDER_WIDTH || y > arena_height - BORDER_WIDTH)
        protected = 1;
    return protected;
}

Listing A.16: landmark.c

#include "landmark.h"
#include "buffer.h"

landmark_map* landmark_map_copy(landmark_map *parent) {
    landmark_map *head;
    if (parent == NULL) {
        head = landmark_map_init(buffer_get_size());
    } else {
        head = parent;
        landmark_map_reference(head);
    }
    assert(head != NULL);
    assert(head->map != NULL);
    return head;
}

void landmark_map_free(landmark_map *node) {
assert (node != NULL);
free (node);
}

void landmark_map_reference (landmark_map *node) {
    assert (node != NULL);
    assert (node->references > 0);
    node->references++;
}

void landmark_map_dereference (landmark_map *node) {
    assert (node != NULL);
    assert (node->references > 0);
    node->references--;
    if (node->references < 1)
        landmark_map_free (node);
}

landmark_map* landmark_map_init (int size) {
    int i;
    landmark_map *node = malloc (sizeof(landmark_map));
    node->references = 1;
    node->map = malloc (sizeof(landmark)*size);
    for (i = 0; i < size; i++) {
        node->map[i].seen = 0;
        node->map[i].unseen = 0;
    }
    return node;
}

void landmark_set_seen (landmark_map *node, int index) {
    assert (node != NULL);
    node->map[index].seen++;
}

void landmark_set_seen_value (landmark_map *node, int index, int value) {
    assert (node != NULL);
    assert (node->map != NULL);
    node->map[index].seen = value;
}

void landmark_set_unseen (landmark_map *node, int index) {
    assert (node != NULL);
    assert (node->map != NULL);
    node->map[index].unseen++;
}

void landmark_set_unseen_value (landmark_map *node, int index, int value) {
    assert (node != NULL);
    node->map[index].unseen = value;
}

// writes a byte buffer given the head of a landmark tree
void landmark_write_map (landmark_map *head, uint8_t *buffer) {

//
memset(buffer, '0', buffer_get_size() * sizeof(uint8_t));
landmark_write_map_subtree(head, buffer);
}

// writes a subtree to the given byte buffer
void landmark_write_map_subtree(landmark_map *node, uint8_t *buffer) {
    int i;
    assert(node != NULL);
    for (i = 0; i < buffer_get_size(); i++)
        buffer[i] = 255 * landmark_seen_probability(node, i);
}

double landmark_seen_probability(landmark_map *node, int index) {
    landmark l;
    double p, sum;
    // assert(node != NULL);
    l = node->map[index];
    sum = l.seen + l.unseen;
    // default to 5% probability if we have no data
    p = 0.05;
    if (sum > 0)
        p = l.seen / sum;
    // max p is 95% to avoid unseen probability of 0%
    if (p > 0.95)
        p = 0.95;
    // assert(p == 0.05 || p == 0.95);
    return p;
}

double landmark_unseen_probability(landmark_map *node, int index) {
    return 1 - landmark_seen_probability(node, index);
}

/*
 * returns distance in the direction specified by step in mm, according to the map
 */
int landmark_map_find_distance(landmark_map *node, int step, particle p) {
    int d, done;
    double x, y, c, s, degrees, theta;

    // forward is now 0 degrees, left -, right +
    // TODO: ETH
    degrees = -120 + step * SENSOR_SPACING_USB;
    theta = (degrees + p.theta) * M_PI / 180;
    c = cos(theta);
    s = sin(theta);

    done = 0;
    for (d = 0; !done; d++) {
        x = d * c + p.x;
        y = d * s + p.y;
    }

    if (d == 0)
        done = 1;
    else
        return d;
}

// make sure it is in bounds and we can see it
if (in_arena(x, y) &&
    landmark_seen_probability(node, buffer_index_from_x_y(x, y)) > 0.9)
    done = 1;
}
return d - 1;
}

void landmark_map_simulate_scan(particle p, int *distances, int m) {
    int i;
    for (i = 0; i < m; i++)
        distances[i] = landmark_map_find_distance(p.map, i, p);
} /*

Listing A.17: lazygl.c
#include "lazygl.h"

uint8_t *buffer1, *buffer2;
int buffer_width, buffer_height, window_width, window_height;

void initGL(uint8_t *b1, uint8_t *b2, int b_w, int b_h, int w_w, int w_h) {
    buffer1 = b1;
    buffer2 = b2;
    buffer_width = b_w;
    buffer_height = b_h;

    window_width = w_w;
    window_height = w_h;

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(window_width, window_height*2);
    glutCreateWindow("sensor");
}

void display() {
    glClearColor(GLCOLOR_BUFFER_BIT);
    glPixelStorei(GLUNPACK_ALIGNMENT, 1);
    glPixelZoom((double)window_width/buffer_width, (double)window_height/
        buffer_height);
    glWindowPos2i(0, 0);
    glDrawPixels(buffer_width, buffer_height, GLLUMINANCE, GLUNSIGNED_BYTE,
        buffer1);
    glWindowPos2i(0, window_height);
    glDrawPixels(buffer_width, buffer_height, GLLUMINANCE, GLUNSIGNED_BYTE,
        buffer2);
    glutSwapBuffers();
}

Listing A.18: particle.c
#include "particle.h"

43
particle particle_sample_motion(particle parent, int dx, int dy, int dtheta) {
    particle p = particle_sample_normal(parent);
    p.x += dx;
    p.y += dy;
    p.theta += dtheta;
    p.theta = p.theta % 360;
    return p;
}

particle particle_sample_normal(particle parent) {
    particle p;
    p.p = parent.p;
    p.x = parent.x + rand_normal(parent.x_var);
    p.y = parent.y + rand_normal(parent.y_var);
    p.theta = parent.theta + rand_normal(parent.theta_var);
    p.theta = p.theta % 360;
    p.x_var = parent.x_var;
    p.y_var = parent.y_var;
    p.theta_var = parent.theta_var;
    p.map = landmark_map_copy(parent.map);
    return p;
}

particle particle_init(int x, int y, int theta) {
    particle p;
    p.p = 0.0;
    p.x = x;
    p.y = y;
    p.theta = theta;
    // initial variance from const.h
    p.x_var = INITIAL_POSITION_VARIANCE;
    p.y_var = INITIAL_POSITION_VARIANCE;
    p.theta_var = INITIAL_ANGLE_VARIANCE;

    return p;
}

Listing A.19: random.c

#include "random.h"
#include <stdio.h>

//static float fn[128];
//static uint32_t kn[128];
static uint32_t seed;
//static float wm[128];

int rand_limit(int limit) {
    int r, d = RANDMAX / limit;
    limit *= d;
    do {r = rand();} while (r >= limit);
    return r/d;
}
double rand_normal(int variance) {
    return box_muller(0, (double)variance);
}

void rand_normal_init() {
    seed = utime();
    srand(seed);
}

// do not use for actual time due to platform differences, only relative microseconds
uint64_t utime() {
    uint64_t t = 0;
    #ifndef LINUX
        FILETIME ft;
        GetSystemTimeAsFileTime(&ft);
        t |= ft.dwHighDateTime;
        t <<= 32;
        t |= ft.dwLowDateTime;
    #endif
    #ifdef LINUX
        struct timeval tv;
        gettimeofday(&tv, NULL);
        t = tv.tv_sec*(uint64_t)1000000+tv.tv_usec;
    #endif
    return t;
}

float ranf() {
    return rand()/(double)RAND_MAX;
}

Listing A.20: replay.c

#include "slam.h"
#include "slamd.h"
#include "replay.h"

#include "lazygl.h"

#include <time.h>

long getMillis() {
    struct timespec spec;
    clock_gettime(CLOCK_MONOTONIC, &spec);

    return 1000*spec.tv_sec + lround(spec.tv_nsec / 1.0e6);
}

static int current_command;
FILE *output_file;

int main (int argc, char **argv) {
    int i, iterations, scan_count, parsed_line[723];
FILE *data_file;
ssize_t read;
char *int_string;
char *line = NULL;
size_t length = 0;
long start_time;

if (argc != 2) {
    printf("must specify output_filename\n");
    exit(1);
}

data_file = fopen("slamd_record.csv", "r");
assert(data_file != NULL);

output_file = fopen(argv[1], "w");
assert(output_file != NULL);

iterations = 0;
scan_count = 0;

while((read = getline(&line, &length, data_file)) != -1) {
    int_string = strtok(line, ",");
    if (strcmp(int_string, "init") == 0)
        continue;
    scanf(int_string, "%d", parsed_line);
    for (i = 1; i < 682; i++) {
        int_string = strtok(NULL, ",");
        scanf(int_string, "%d", parsed_line + i);
    }
    int_string = strtok(NULL, "\n");
    scanf(int_string, "%d\n", parsed_line + i);

    current_command = parsed_line[0];

    switch(current_command) {
    case SLAMD_INIT:
        swarm_init(parsed_line[1], parsed_line[2], parsed_line[3], parsed_line[4], parsed_line[5], parsed_line[6]);
        start_time = getMillis();
        break;
    case SLAMD_MOVE:
        swarm_move(parsed_line[1], parsed_line[2], parsed_line[3]);
        break;
    case SLAMD_UPDATE:
        i = 0;
        while (!swarm_converged() && i < 1) {
            i++;
            swarm_update(parsed_line + 1);
            if (!swarm_converged() && i < 5)
                swarm_move(0, 0, 0);
        }
        scan_count++;
    }
if(swarm_converged())
    swarm_map(parsed_line + 1);
break;
case SLAMD_X:
case SLAMD_Y:
case SLAMD_THETA:
    break;
}
iterations++;
}

long difference = getMillis() - start_time;
fprintf(output_file, "%ld", difference);

fclose(data_file);
close(output_file);
free(line);
return 0;
}

Listing A.21: sensor.c

#include "sensor.h"

const static int poll_time_eth = 25000;
const static int poll_time_usb = 100000;

static urg_t connection_eth;
static tPort *connection_usb;
static long *buffer_eth;
static int **buffer_usb;
static raw_sensor_scan_lidar_data;
static raw_sensor_scan n_scans[MAX_SCANS];
static int last_poll_usb, last_poll_eth, n, eth, usb;

pthread_t sensor_init_thread () {
    pthread_t t;
    pthread_create(&t, NULL, sensor_init, NULL);
    return t;
}

void* sensor_init(void *null_pointer) {
    int max_data_size;

    urg_connection_type_t type_eth = URGETHERNET;
    urg_measurement_type_t m_type_eth = URGMULTIECHO;
    char *device_eth = "192.168.0.10";
    char *device_usb = "/dev/ttyACM0";

    eth = 0;
    usb = 0;
if (urg_open(&connection_eth, type_eth, device_eth, 10940) < 0) {
    printf("Could not connect to the ethernet sensor\n");
} else eth = 1;

connection_usb = scipConnect(device_usb);
if (connection_usb == NULL) {
    printf("Could not connect to the usb sensor\n");
} else usb = 1;

// start measurement
if (eth) {
    max_data_size = urg_max_data_size(&connection_eth);
    buffer_eth = malloc(sizeof(int)*max_data_size);
    urg_start_measurement(&connection_eth, m_type_eth, 0, 0);
    last_poll_eth = utime() - poll_time_eth;
}

if (usb) {
    switchToScip2(connection_usb);
    scip2SetComSpeed(connection_usb,115200);
    last_poll_usb = utime() - poll_time_usb;
}

return NULL;
}

raw_sensor_scan sensor_read_raw() {
    // Initialize parameters for laser scanning
    int start_step = 44;
    int end_step = 725;
    int step_cluster = 1;
    int scan_interval = 0;
    int scan_num = 1;
    int step_num;
    long timestamp;

    int sleep_time = poll_time_usb - (utime() - last.poll_usb);
    if (sleep_time > 0)
        usleep(sleep_time);
    else printf("sleepless for %g seconds\n", -sleep_time/1000000.0);

    if (eth) {
        step_num = urg_get_distance(&connection_eth, buffer_eth, &timestamp);
        last_poll_eth = utime();
    }

    if (usb) {
        buffer_usb = scip2MeasureScan(connection_usb, start_step, end_step,
                                        step_cluster,
                                        scan_interval, scan_num, ENC_3BYTE, &step_num);
        last_poll_usb = utime();
    }
}
int i, distance;
for (i = 0; i < step_num; i++) {
    distance = buffer_usb[0][i];
    if ((distance >= SENSOR_MIN) && (distance <= SENSOR_MAX_USB))
        // Copy the data into the static variable
        lidar_data.distances[i] = distance;
}
// free usb_buffer, it is malloc'd every time we call scip2MeasureScan
for (i = 0; i < scan_num; i++)
    free(buffer_usb[i]);
free(buffer_usb);
return lidar_data;
}

void* sensor_read_raw_n(void *null_pointer) {
    int i;
    for (i = 0; i < n && i < MAX_SCANS; i++)
        n_scans[i] = sensor_read_raw();

    return NULL;
}

pthread_t sensor_read_raw_n_thread(int requested_n) {
    pthread_t t;
    n = requested_n;
    assert(pthread_create(&t, NULL, sensor_read_raw_n, NULL) == 0);
    return t;
}

raw_sensor_scan sensor_fetch_index(int index) {
    return n_scans[index];
}

Listing A.22: slamd.c
#include "slamd.h"

int main(int argc, char **argv) {
    int i, m, param_size, return_size, last_command;
    FILE *record;

    // set up shared memory
    param_sem = CreateSemaphore(NULL, 0, 1, param_sem_name);
    return_sem = CreateSemaphore(NULL, 0, 1, return_sem_name);
    assert(param_sem != NULL);
    assert(return_sem != NULL);

    // parameter space size is (sensor readings + 1) * sizeof(int)
    // first int specifies which function, remainder are params
    // TODO: fix constant 1081/721
    param_size = (721 + 1) * sizeof(int);
    param_handle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE,
0, param_size, param_shm_name);

// return space size is sizeof(int)
return_size = sizeof(int);
return_handle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, return_size, return_shm_name);

params = (int*)MapViewOfFile(param_handle, FILE_MAP_WRITE, 0, 0, param_size);
assert(params != NULL);

return_value = (int*)MapViewOfFile(return_handle, FILE_MAP_WRITE, 0, 0, return_size);
assert(return_value != NULL);

m = 0;
while (1) {
    WaitForSingleObject(param_sem, INFINITE);
    record = fopen("slamd_record.csv", "a");
    last_command = params[0];
    switch (params[0]) {
    case SLAMD_INIT:
        fprintf(record, "init\n");
        m = params[1];
        printf("calling swarm_init(%i, %i, %i, %i, %i)\n", params[1], params[2],
        params[3], params[4], params[5], params[6]);
        swarm_init_internal(params[1], params[2], params[3], params[4], params [5]);
        break;
    case SLAMD_MOVE:
        printf("calling swarm_move(%i, %i, %i)\n", params[1], params[2], params [3]);
        swarm_move_internal(params[1], params[2], params[3]);
        break;
    case SLAMD_UPDATE:
        printf("calling swarm_update(%i, %i, %i, %i, %i, ... )\n", params[1],
        params[2], params[3], params[4], params[5]);
        swarm_update_internal(params + 1);
        break;
    case SLAMD_CONVERGED:
        *return_value = swarm_converged_internal();
        printf("%i\n", *return_value);
        break;
    case SLAMD_X:
        *return_value = swarm_get_best_x_internal();
        printf("x_value: %i\n", *return_value);
        break;
    case SLAMD_Y:
        *return_value = swarm_get_best_y_internal();
        printf("y_value: %i\n", *return_value);
        break;
    case SLAMD_THETA:
        *return_value = swarm_get_best_theta_internal();
        printf("theta_value: %i\n", *return_value);
        break;
    }
    record = fclose(record);
    fprintf(record, "last_command: %i\n", last_command);
    fprintf(record, "m: %i\n", m);
    record = freopen("slamd_record.csv", "a", record);
    last_command = 0;
}

50
break;

// write to file
for (i = 0; i < m + 1; i++)
    fprintf(record, "%i," , params[i]);
    fprintf(record, "%i\n", return_value[0]);
fclose(record);

// UPDATE releases semaphore inline to return
// as early as possible while it normalizes,
// sorts, and resamples in the background
if (last_command != SLAMD_UPDATE)
    ReleaseSemaphore(return_sem, 1, NULL);

return 0;

Listing A.23: swarm.c
#include "swarm.h"
#include <stdio.h>
#include "slamd.h"

static particle particles[INITIAL_PARTICLE_FACTOR*PARTICLE_COUNT];
static particle previous_particles[INITIAL_PARTICLE_FACTOR*PARTICLE_COUNT];
static particle best_particle;
static landmark_map *map;
static int iterations = 0;
static int converged, m, sensor_degrees, long_side, short_side, start;
static double spacing;  // 550 mm
static int sensor_radius;

#ifndef LINUX
_declspec(dllexport) void swarm_init(int m_in, int degrees_in, int long_side_in, int short_side_in, int start_in, int radius) {
    int param_size, return_size;
    m = m_in;
    sensor_degrees = degrees_in;
    long_side = long_side_in;
    short_side = short_side_in;
    start = start_in;
    spacing = sensor_degrees/(double)(m);
    sensor_radius = radius;

    // set up shared memory
    param_sem = CreateSemaphore(NULL, 0, 1, param_sem_name);
    return_sem = CreateSemaphore(NULL, 0, 1, return_sem_name);
    ready_sem = CreateSemaphore(NULL, 0, 1, ready_sem_name);
    assert(param_sem != NULL);
    assert(return_sem != NULL);

    return

#endif
assert(ready_sem != NULL);

converged = 0;

// parameter space size is (sensor readings + 1) * sizeof(int)
// first int specifies which function, remainder are params
param_size = (m_in + 1) * sizeof(int);
param_handle = CreateFileMapping((HANDLE)0xFFFFFFFF, NULL, PAGE_READWRITE, 0, param_size, param_shm_name);
assert(param_handle != NULL);

// return space size is sizeof(int)
return_size = sizeof(int);
return_handle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, return_size, return_shm_name);
assert(return_handle != NULL);

params = (int*)MapViewOfFile(param_handle, FILE_MAP_WRITE, 0, 0, param_size);
assert(params != NULL);

return_value = (int*)MapViewOfFile(return_handle, FILE_MAP_WRITE, 0, 0, return_size);
assert(return_value != NULL);

params[0] = SLAMD_INIT;
params[1] = m_in;
params[2] = degrees_in;
params[3] = long_side_in;
params[5] = start_in;
params[6] = radius;

ReleaseSemaphore(param_sem, 1, NULL);
WaitForSingleObject(return_sem, INFINITE);

_declspec(dllexport) void swarm_move(int dx, int dy, int dtheta) {
    params[0] = SLAMD_MOVE;
    params[1] = dx;
    params[2] = dy;
    params[3] = dtheta;

    ReleaseSemaphore(param_sem, 1, NULL);
    WaitForSingleObject(return_sem, INFINITE);
}

_declspec(dllexport) void swarm_update(int *distances) {
    params[0] = SLAMD_UPDATE;
    memcpy(params + 1, distances, m*sizeof(int));
    ReleaseSemaphore(param_sem, 1, NULL);
    WaitForSingleObject(return_sem, INFINITE);
}
```c
void swarm_update_finalize() {
    assert(ready_sem != NULL);
    WaitForSingleObject(ready_sem, INFINITE);
}

void swarm_map(int *distances) {
    params[0] = SLAMD_MAP;
    memcpy(params + 1, distances, m*sizeof(int));
    ReleaseSemaphore(param_sem, 1, NULL);
    WaitForSingleObject(return_sem, INFINITE);
}

// returns 1 if standard deviations are low
// 0 otherwise
int swarm_converged() {
    params[0] = SLAMD_CONVERGED;
    ReleaseSemaphore(param_sem, 1, NULL);
    WaitForSingleObject(return_sem, INFINITE);
    return *return_value;
}

int swarm_get_best_x() {
    params[0] = SLAMD_X;
    ReleaseSemaphore(param_sem, 1, NULL);
    WaitForSingleObject(return_sem, INFINITE);
    return *return_value;
}

int swarm_get_best_y() {
    params[0] = SLAMD_Y;
    ReleaseSemaphore(param_sem, 1, NULL);
    WaitForSingleObject(return_sem, INFINITE);
    return *return_value;
}

int swarm_get_best_theta() {
    params[0] = SLAMD_THETA;
    ReleaseSemaphore(param_sem, 1, NULL);
    WaitForSingleObject(return_sem, INFINITE);
    return *return_value;
}

#ifdef LINUX
void swarm_init_internal(int m_in, int degrees_in, int long_side_in, int short_side_in, int start_in, int radius) {
#else
void swarm_init(int m_in, int degrees_in, int long_side_in, int short_side_in, int start_in, int radius) {
#endif
    int i, j, k;
    int x, y, theta;
```
```c
particle initial_map;
double t;
m = m_in;
sensor_degrees = degrees_in;
long_side = long_side_in;
short_side = short_side_in;
start = start_in;
spacing = sensor_degrees / (double)(m);
sensor_radius = radius;

rand_normal_init();

buffer_set_arena_size(long_side, short_side);

initial_map.map = landmark_map_copy(NULL);
map = landmark_map_copy(NULL);

// draw initial border
for (k = 0; k < long_side; k += BUFFERFACTOR)
  for (j = 0; j < BORDER_WIDTH; j += BUFFERFACTOR) {
    landmark_set_seen_value(initial_map.map, buffer_index_from_x_y(k, j), 10000);
    landmark_set_seen_value(initial_map.map, buffer_index_from_x_y(k, short_side - 1 - j), 10000);
  }

for (k = 0; k < short_side; k += BUFFERFACTOR)
  for (j = 0; j < BORDER_WIDTH; j += BUFFERFACTOR) {
    landmark_set_seen_value(initial_map.map, buffer_index_from_x_y(j, k), 10000);
    landmark_set_seen_value(initial_map.map, buffer_index_from_x_y(long_side - 1 - j, k), 10000);
  }

// initialize first round of particles
x = start/2;
for (i = 0; i < INITIAL_PARTICLE_FACTOR*PARTICLE_COUNT; i++) {
  y = short_side/4;
  if (rand_limit(2))
    y *= 3;
  theta = rand_limit(360) - 180;
  t = theta*M_PI/180;
  particles[i] = particle_init(x + sensor_radius*cos(t), y + sensor_radius*sin(t), theta);
  particles[i].map = initial_map.map;
  landmark_map_reference(particles[i].map);
}

best_particle = particles[0];
landmark_map_dereference(initial_map.map);
```

#ifndef LINUX
    return_sem = CreateSemaphore(NULL, 0, 1, return_sem_name);
    ready_sem = CreateSemaphore(NULL, 0, 1, ready_sem_name);
    assert(return_sem != NULL);
    assert(ready_sem != NULL);
#endif

#endif LINUX

void swarm_move_internal(int dx, int dy, int dtheta) {
#endif LINUX
void swarm_move(int dx, int dy, int dtheta) {
#endif

    int i, tries, p_count;
    double t_old, t_new;
    particle p;
    t_old = best_particle.theta*M_PI/180;
    t_new = (best_particle.theta + dtheta)*M_PI/180;

    // adjust dx/dy for dtheta using radius and old theta
    dx += sensor_radius*(cos(t_new) - cos(t_old));
    dy += sensor_radius*(sin(t_new) - sin(t_old));

    if (iterations < 1)
        p_count = INITIAL_PARTICLE_FACTOR*PARTICLE_COUNT;
    else
        p_count = PARTICLE_COUNT;

    // reset convergence if we have a non 0 move
    if (abs(dx) > 0 || abs(dy) > 0 || abs(dtheta) > 0)
        converged = 0;

    // add motion
    for (i = 0; i < p_count; i++) {
        p = particles[i];
        tries = 0;
        do {
            // ignore kinematics 20% of the time
            if ((rand()) / (double)RAND_MAX) < 0.8) {
                // sample motion distribution
                particles[i] = particle_sample_motion(p, dx, dy, dtheta);
            } else {
                // sample normal distribution
                particles[i] = particle_sample_normal(p);
            }

            // dereference old map, particle_sample_* copied it already
            landmark_map_dereference(p.map);
            tries++;
        } while (!in_arena(particles[i].x, particles[i].y) && tries < 5);
        // if we failed repeatedly, reuse previous pose
        if (tries >= 5)
            particles[i] = p;
    }

    // ...
#ifndef LINUX
void swarm_update_internal(int *distances) {
#endif
#endif
#endif

int i, j, k, l, cull, cut, cuts, cull_index, best_index, p_count;
int particle_trig_initialized = 0;
int x, y, count;
double mean[3], stddev[3];
double distance, degrees, s, c, total, min, p, step, p95, p5;
double cos_d, sin_d, factor, theta_d;
const double BIG = 10000000;

#ifndef LINUX
    ReleaseSemaphore(return_sem, 1, NULL);
#endif

    if (iterations < 1)
        p_count = INITIAL_PARTICLE_FACTOR*PARTICLE_COUNT;
    else
        p_count = PARTICLE_COUNT;

    p95 = -log(0.95);
p5 = -log(0.05);
factor = M_PI/180;
cuts = m/CULLING_FACTOR + 1;
cull_index = 0;
particle_trig_initialized = 0;
for (cull = 0; cull < CULLING_FACTOR; cull++) {
    // evaluate each direction for each particle
    for (cut = 0; cut < cuts; cut++) {
        j = cull + CULLING_FACTOR*cut;
        if (j >= m) {
            continue;
        }
        distance = distances[j];
        // skip any distances that are more than 8 meters in case we shoot over
        // the walls
        if (distance < 8000) {
            degrees = -sensor_degrees/2.0 + j*spacing;
            theta_d = degrees*factor;
            sin_d = sin(theta_d);
            cos_d = cos(theta_d);

            // evaluate the particle 's relative probability
            // skip culled particles
            min = BIG;
best_index = 0;
            for (i = cull_index; i < p_count; i++) {
                if (particle_trig_initialized == 0) {
                    theta_d = particles[i].theta * factor;
                }
            }
        }
    }
}
particles[i].sin = sin(theta_d);
particles[i].cos = cos(theta_d);
}

// forward is now 0 degrees, left − , right +
// theta = (degrees − particles[i].theta)*M_PI/180;
// sin(A − B) = sin(A)*cos(B) − sin(B)*cos(A)
s = sin_d * particles[i].cos − particles[i].sin * cos_d;
// cos(A −B) = cos(A)*cos(B) + sin(A)*sin(B)
c = cos_d * particles[i].cos + sin_d * particles[i].sin;

// check and record unseen every 200 mm, within 1 meter of obstacle
for (l = distance − 200; l > distance − 1000; l −= 200) {
  x = l*c + particles[i].x;
y = l*s + short_side − particles[i].y;

  // make sure it is in bounds
  if (in_arena(x, y)) {
    k = buffer_index_from_x_y(x, y);
    p = landmark_unseen_probability(particles[i].map, k);
    if (p == 0.05) {
      particles[i].p += p5;
    } else {
      particles[i].p += p95;
    }
  } else particles[i].p += p5;
}

// check and record seen
x = distance*c + particles[i].x;
y = distance*s + short_side − particles[i].y;

  // make sure it is in bounds
  if (in_arena(x, y)) {
    k = buffer_index_from_x_y(x, y);
    p = landmark_seen_probability(particles[i].map, k);
    if (p == 0.05) {
      particles[i].p += p5;
    } else {
      particles[i].p += p95;
    }
  } else particles[i].p += p5;

if (particles[i].p < min) {
  min = particles[i].p;
  best_index = i;
}
particle_trig_initialized = 1;
}

// skip for last iteration
if (cull != CULLING_FACTOR − 1) {
  quick_sort(particles, cull_index, p_count − 1);
}
best_index = p.count - 1;
// cull bottom

cull_index = CULLING_PERCENT*p_count*(cull + 1.0)/CULLING_FACTOR;
for (i = 0; i < cull_index; i++) {
    particles[i].p = BIG;
}
}

// clear old best, save new best, copy the map we are about to dereference
if (iterations > 0)
    landmark_map_dereference(best_particle.map);
best_particle = particles[best_index];
best_particle.map = landmark_map_copy(best_particle.map);

// normalize particle log probabilities, convert to normal probabilities for resampling
total = 0.0;
for (i = 0; i < p_count; i++) {
    particles[i].p -= min;
    particles[i].p = pow(M_E, -particles[i].p);
    total += particles[i].p;
}
for (i = 0; i < p_count; i++) {
    particles[i].p /= total;
}

// quicksort particles
quick_sort_reverse(particles, 0, p_count - 1);

// calculate standard deviation of top 90%
i = p_count - 1;
total = 0.0;
mean[0] = 0.0;
mean[1] = 0.0;
mean[2] = 0.0;
count = 0;
while (total < 0.99) {
    mean[0] += particles[i].x;
    mean[1] += particles[i].y;
    mean[2] += particles[i].theta;
    count++;
    total += particles[i].p;
    i--;
}
for (i = 0; i < 3; i++)
    mean[i] /= count;

stddev[0] = 0.0;
stddev[1] = 0.0;
stddev[2] = 0.0;
total = 0.0;
i = p_count - 1;
while (total < 0.99) {
    stddev[0] += pow(mean[0] - particles[i].x, 2);
    stddev[1] += pow(mean[1] - particles[i].y, 2);
    total += particles[i].p;
i--;
}

for (i = 0; i < 3; i++) {
    if (count > 1)
        stddev[i] /= count - 1;
    else
        stddev[i] /= count;
    stddev[i] = sqrt(stddev[i]);
}

// make sure 99% of particles are within 50 mm and 1 degree
if (sqrt(pow(stddev[0], 2) + pow(stddev[1], 2)) < 50 && stddev[2] < 2)
    converged = 1;
else
    converged = 0;

if (!converged || count > 1)
    printf("count: %d, max_prob: %g, standard deviations: %g, %g, %g, converged: %d\n",
           count, particles[p_count - 1].p, stddev[0], stddev[1], stddev[2], converged);

// save old particles before we resample
memcpy(previous_particles, particles, sizeof(particle)*p_count);

// resample with replacement
p = rand()/(double)RAND_MAX;
step = 1.0/p_count;
for (i = 0; i < p_count; i++) {
    p += step;
    if (p > 1.0) p = 1.0;
    total = 0.0;
    j = p_count - 1;
    while (j-- && total < p)
        total += previous_particles[j + 1].p;
    particles[i] = previous_particles[j + 1];
    particles[i].map = landmark_map_copy(particles[i].map);
}

// dereference previous particle maps
for (i = 0; i < p_count; i++)
    landmark_map_dereference(previous_particles[i].map);

// restore log probabilities
for (i = 0; i < p_count; i++)
    particles[i].p = -log(particles[i].p);
iterations++;
#ifndef LINUX
    ReleaseSemaphore(ready_sem, 1, NULL);
#endif

void pswap(particle particles[], int one, int two) {
    if (one != two) {
        particle tmp;
        tmp = particles[one];
        particles[one] = particles[two];
        particles[two] = tmp;
    }
}

void quick_sort(particle particles[], int start, int end) {
    if (start < end) {
        // partition
        double pivot = particles[(start + end)/2].p;

        int left = start;
        int right = end;

        while (left <= right) {
            while (particles[left].p > pivot) left++;
            while (particles[right].p < pivot) right--;
            if (left <= right) {
                pswap(particles, left, right);
                left++; right--;
            }
        }

        quick_sort(particles, start, right);
        quick_sort(particles, left, end);
    }
}

void quick_sort_reverse(particle particles[], int start, int end) {
    if (start < end) {
        // partition
        double pivot = particles[(start + end)/2].p;

        int left = start;
        int right = end;

        while (left <= right) {
            while (particles[left].p < pivot) left++;
            while (particles[right].p > pivot) right--;
            if (left <= right) {
                pswap(particles, left, right);
                left++; right--;
            }
        }
    }
}
#ifndef LINUX
void swarm_map_internal(int *distances) {
  #endif
  #ifdef LINUX
  void swarm_map(int *distances) {
  #endif
  int j, l, x, y, k;
  double distance, degrees, theta, s, c;

  for (j = 0; j < m; j++) {
    distance = distances[j];
    // forward is now 0 degrees, left -, right +
    degrees = -sensor_degrees/2.0 + j*spacing;
    theta = (degrees - best_particle.theta)*M_PI/180;
    s = sin(theta);
    c = cos(theta);
    // check and record unseen every 10 mm
    for (l = 0; l < distance; l += 10) {
      x = l*c + best_particle.x;
      y = l*s + short_side - best_particle.y;
      // make sure it is in bounds
      if (in_arena(x, y)) {
        k = buffer_index_from_x_y(x, y);
        landmark_set_unseen(map, k);
      }
    }
    // check and record seen
    x = distance*c + best_particle.x;
    y = distance*s + short_side - best_particle.y;
    // make sure it is in bounds
    if (in_arena(x, y)) {
      k = buffer_index_from_x_y(x, y);
      landmark_set_seen(map, k);
    }
  }
  #ifndef LINUX
  int swarm_converged_internal() {
  #endif
  #endif
  #ifdef LINUX
  int swarm_converged() {
  #endif
    return converged;
  }
}
```c
#ifdef LINUX
int swarm_get_best_x_internal() {
#endif
#ifdef LINUX
    int swarm_get_best_x() {
#endif
t    return best_particle.x - sensor_radius*cos(best_particle.theta*M_PI/180);
}
#endif LINUX
int swarm_get_best_y_internal() {
#endif
#ifdef LINUX
    int swarm_get_best_y() {
#endif
t    return short_side - (best_particle.y - sensor_radius*sin(best_particle.theta *M_PI/180));
}
#endif LINUX
int swarm_get_best_theta_internal() {
#endif
#define LINUX
    int swarm_get_best_theta() {
#endif
t    int t = -(best_particle.theta + 180) % 360;
    if (t <= -180) t += 360;
    else if (t > 180) t -= 360;
    return t;
}
int swarm_get_x(int i) {
    return particles[i].x - sensor_radius*cos(particles[i].theta*M_PI/180) - BORDER_WIDTH;
}
int swarm_get_y(int i) {
    return short_side - (particles[i].y - sensor_radius*sin(particles[i].theta*M_PI/180)) - BORDER_WIDTH;
}
int swarm_get_theta(int i) {
    return -(particles[i].theta + 180) % 360;
}
void swarm_get_best_buffer(uint8_t *buffer) {
    landmark_write_map(best_particle.map, buffer);
}
void swarm_get_map_buffer(uint8_t *buffer) {
    landmark_write_map(map, buffer);
}
```
void swarm_get_map() {
    return *map;
}

void swarm_get_all_particles(particle **p) {
    *p = particles;
}

int in_arena(int x, int y) {
    if (x >= 0 && x < long_side && y >= 0 && y < short_side)
        return 1;
    else return 0;
}

Listing A.24: test_sensor.c

#include "urg_driver/urg_sensor.h"

int main(int argc, char **argv) {
    urg_t connection;
    urg_connection_type_t type = URG_ETHNET;
    char *device = "192.168.0.10";
    int direction_count, max_data_size, i;
    long timestamp;
    long *data;

    // open connection
    if (urg_open(&connection, type, device, 10940) < 0)
        return 1;

    max_data_size = urg_max_data_size(&connection);
    data = malloc(sizeof(long)*max_data_size);

    // start measurement
    // (connection, type, scan times (0 is keep going), skip)
    urg_start_measurement(&connection, type, 0, 0);

    while (1) {
        direction_count = urg_get_distance(&connection, data, &timestamp);
        for (i = 0; i < max_data_size; i++) {
            printf("%i", data[i]);
        }
        printf("\nmax_data_size:%i\n", max_data_size);
        sleep(1);
    }

    return 0;
}
Listing B.1: charts.pl

```perl
#!/usr/bin/env perl6

use Chart::Gnuplot;

my @results = "results.csv" . IO . slurp . chomp . split("\n") . split(",").map: -> @row {
  %(
    'particle-count' => @row[0].Int,
    'culling-factor' => @row[1].Int,
    'positional-deviation' => @row[2].Rat,
    'angular-deviation' => @row[3].Rat,
    'time' => @row[4].Rat/1000
  )
};

my $runtime_chart = start_chart('runtime', 'Runtime', 'Particle_Count', 'Time_{(s)}');
for 1, 5, 20 -> $factor {
  my @selected = @results . grep: *{'culling-factor'} == $factor;
  plot_one($runtime_chart, @selected, "Culling\_Factor\_$_factor", 'particle-count', 'time')
}
finish_chart($runtime_chart);

for 1000, 8000, 64000 -> $count {
  my $culling_runtime_chart = start_chart("culling-$_count-runtime", "$_count/1000\_Particles\_Runtime"
                                            , 'Culling\_Factor', 'Time_{(s)}', True);
  my @data = @results . grep: -> %row {
    %row{'particle-count'} == $count &
    ((%row{'culling-factor'} != 160 && $count != 64000) ||
     %row{'culling-factor'} < 21)
  };
  plot_one($culling_runtime_chart, @data, '', 'culling-factor', 'time');
  finish_chart($culling_runtime_chart)
```

64
for ( 'Positional', 'mm', 'Angular', 'degrees') -> $type, $units {
    my $accuracy_chart = start_chart("accuracy-{$type.uc}", "$type_Deviation_vs_Particle_Count",
        'Particle_Count', "$type_Deviation_($units)")
    ;
    for 1, 5, 20 -> $factor {
        my @selected = @results.grep: *{'culling-factor'} == $factor;
        plot_one($accuracy_chart, @selected, "Culling_Factor_$factor",
            'particle-count', "{$type.uc}-deviation")
    }
    finish_chart($accuracy_chart);

    my $factor_chart = start_chart("culling-accuracy-{$type.uc}", "$type_Deviation_vs_Culling_Factor",
        "Culling_Factor", "$type_Deviation_($units)"
    );
    for 1000, 8000, 64000 -> $count {
        my @selected = @results.grep: -> %row {
            %row{'particle-count'} == $count &
            ((%row{'culling-factor'} < 160 && $count != 64000) ||
                %row{'culling-factor'} < 21)
        };
        plot_one($factor_chart, @selected, "$count_Particles",
            'culling-factor', "{$type.uc}-deviation")
    }
    finish_chart($factor_chart)
}

sub start_chart(Str $filename, Str $title, Str $xLabel, Str $yLabel, Bool $noLegend = False) {
    my $gnu = Chart::Gnuplot.new(:terminal("png"), :filename("{$filename}.png")
    );
    $gnu.legend(:left) unless $noLegend;
    if $filename eq 'culling-accuracy-angular' {
        $gnu.yrange(:min(0), :max(120))
        $gnu.xrange(:min(0), :max(90))
    } elsif $filename eq 'culling-accuracy-positional' {
        $gnu.yrange(:min(0), :max(800))
        $gnu.xrange(:min(0), :max(90))
    } elsif $filename =~ /culling\-\-runtime/ {
        if $filename eq 'culling-64000-runtime' {
            $gnu.xrange(:min(0), :max(25))
        } else {
            $gnu.xrange(:min(0), :max(90))
        }
    }
    $gnu.title(:text($title));
    $gnu.xlabel(:label($xLabel));
    $gnu.ylabel(:label($yLabel));
    $gnu
sub plot_one($chart, @data, Str $title, Str $xLabel, Str $yLabel) {
    $chart.plot(:title($title), :vertices(extract(@data, $xLabel, $yLabel)),
               :style(’linespoints’), :ps(2.5));
}

sub finish_chart($chart) {
    $chart.dispose
}

sub extract(@rows, Str $xLabel, Str $yLabel) {
    @rows.map: -> %row {
               [%row{$xLabel}, %row{$yLabel}]
    }
}

Listing B.2: gather.pl

#!/usr/bin/env perl6

my @output = ();

for (dir) -> $dir {
    if $dir ~~ /−/ {
        try {
            my ($count, $factor) = $dir.path.split(’−’);
            my @summary = ”$dir/summary.csv” . IO.slurp.chomp.split(”\n”).map: *.
                          split(’,’);
            my $time = @summary.pop[1];
            @summary.pop;
            @summary.map: -> (%$x, %$y, %$x_sd, %$y_sd, %$t_sd) {
                ($$x_sd, $$y_sd, $$t_sd)
            }
            my @zipped = [Z] @summary;
            @zipped = (@zipped[0].flat, @zipped[1].flat).flat, @zipped[2];
            my @means = @zipped.map: -> @l {
                @l R/ [+] @l
            };
            my @sd = @zipped.kv.map: -> $i, @l {
                my $mean = @means[$i];
                sqrt (@l - 1) R/ [+] map (* - $mean)**2, @l
            }
            @output.push(”$count, $factor, @means[0], @means[1], $time”);
        }
    }
}
(@output.sort: -> $one, $two {
    my $one = $one.split(”,”);
    my $two = $two.split(”,”);
    if $one[0].Int == $two[0].Int {
        $one[1].Int > $two[1].Int
    } else {
        $one[0].Int > $two[0].Int
    }
})
Listing B.3: process.pl

```perl
#!/usr/bin/env perl

sub MAIN(Str $folder) {
    if $folder eq 'test' {
        run_tests;
        exit
    }
    my $foldername = $folder;
    $foldername = "${folder}/" unless $folder =~ /\$/;
    my @files = $foldername.IO.dir;
    my @times = ();
    my @samples = gather for @files -> $file {
        next if $file.path =~ /summary/;
        my @positions = $file.slurp.chomp.split("\n").map: *.split("\,");
        @times.push(@positions.pop[0]);
        take @positions;
    }
    my $t_mean = mean(@times);
    my $t_dev = dev(@times, $t_mean);
    my @summaries = map &mean_and_std, ([Z] @samples);

    my @output = @summaries.map: -> @list { @list.flat.join("\," ) };
    @output.push("iterations ,{@samples.elems}" );
    @output.push("time ,{ $t_mean },{ $t_dev }\n");
    spurt "{ $foldername }summary.csv", @output.join("\n");
}

sub mean(@l) {
    @l R/ [+] @l;
}

sub dev(@l, $mean) {
    sqrt (@l - 1) R/ [+] map (* - $mean)**2, @l;
}

sub mean_and_std(@position) {
    my @zipped = [Z] @position;
    my @means = map &mean, @zipped;
    my @stddevs = @zipped.kv.map: -> $i, @l {
        my $mean = @means[$i];
        dev(@l, $mean)
    };
    (@means, @stddevs);
}

sub run_tests {
    use Test;
    my @case = ((1, 2, 3, 4, 5, 6, 7, 8, 9, 10), (2, 3, 4, 5, 6, 17, 8, 9, 10, 11),
                (3, 4, 14, 6, 7, 8, 9, 10, 11, 12));
}
```

my @expected = ( 5.5, 7.5, 8.4, 3.0276503541, 4.503085362, 3.5023801431);
my @results = mean_and_std([ Z ] @case).flat;
for @expected.kv -> $i, $v {
    ok (@results[$i] - $v).abs < 0.0000001, "position_$i"
}

Listing B.4: vary_params.pl

#!/usr/bin/env perl

my $original_template = 'const.h.template'.IO.slurp;

my @particle_counts = (1000, 2000, 4000, 8000, 16000, 32000, 64000);
my @culling_factors = (1, 5, 10, 20);
my $iterations = 200;
my $iteration_offset = 0;

for @particle_counts -> $particle_count {
    for @culling_factors -> $culling_factor {
        my $initial_particle_factor = 1;

        my $template = $original_template;
        $template .= subst(/ particle_count /, $particle_count.Str);
        $template .= subst(/ initial_particle_factor /, $initial_particle_factor.Str);
        $template .= subst(/ culling_factor /, $culling_factor.Str);

        spurt 'const.h', $template;
        shell 'make';
        my $dir = "output/{$particle_count}-{$culling_factor}";
        mkdir $dir;

        for (1+$iteration_offset)..($iterations+$iteration_offset) -> $iteration {
            ($particle_count, $culling_factor, $iteration).join("-").say;
            shell "./replay$dir/run-{$iteration}.csv"
        }

        shell "./output/process.pl_$dir";
    }
}