Mobile Robot Localization and Navigation in Physical Environments Using Hand-Drawn Maps

A Thesis Presented
by

Kevin Michael Robb

to

The Khoury College of Computer Sciences

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Robotics

Northeastern University
Boston, Massachusetts

December 2023

Contents

List of Figures

List of Acronyms

A* Best-First Graph Search. One of the most common and simple path planning algorithms when using an occupancy grid map. The robot position and the goal are provided as cells on the map, and the algorithm uses best-first search to find the shortest path to connect the points. The result may not be unique if there are multiple equally optimal paths.

AMCL Adaptive Monte Carlo Localization. A popular particle-filter-based localization algorithm which uses a highly detailed occupancy grid map as well as measurements from a 360 degree planar LiDAR sensor to estimate the robot pose.

BEV Bird's Eye View. Term for the top-down representation of the world that we use to simplify our algorithms and the state representation to a 2-dimensional space. Since the robot drives around on the ground, we can treat it as a 2D point moving on the plane.

CMN Coarse Map Navigation. Overall process for this project, in which a rough, low-resolution map is used in combination with local occupancy measurements to run localization, path planning, and navigation tasks.

DBF Discrete Bayesian Filter. Bayesian filters are a class of iterative algorithms in which the current state estimation depends only on the previous estimate, some propagation term (such as a robot motion model and the last velocity commands), and one or more incoming measurement sources. A discrete Bayesian filter estimates robot position as a cell on the map, so the state of this filter is an array of probabilities of the same dimension as the map being used. Whichever cell has the highest (normalized) probability is taken as the position estimate, and we say the filter has "converged" when there is a single clearly most probable cell.

FOV Field of View. A camera such as the RealSense series has a limited area it can observe. There is a horizontal and a vertical field-of-view for each data stream, and generally we care more about the horizontal, since we project data down onto the plane.

LiDAR Light Detection and Ranging. A common sensor on robotics platforms, in which beams of light are emitted, and their return times measured, in order to measure distances to objects in the environment. A 2D planar LiDAR spins in a circle to generate measurements in all directions.

ROS Robot Operating System. A popular open-source set of software libraries and tools to streamline robotics projects. It handles the networking and low-level communication so that developers can simply create nodes in Python or C++ that publish and subscribe to topics in order to pass information between programs and devices. In our case, the robot publishes odometry and sensor data, and subscribes to velocity commands. Both our host PC and the robot are using ROS Noetic on Ubuntu 20.04.

SLAM Simultaneous Localization and Mapping. General name for a class of algorithms in which a robot uses its sensors to build a map of the environment while also estimating its own position in that map.

Abstract of the Thesis

Mobile Robot Localization and Navigation in Physical Environments Using Hand-Drawn Maps

by

Kevin Michael Robb

Master of Science in Robotics

Northeastern University, December 2023

Dr. Lawson L.S. Wong, Advisor

For a mobile robot to operate autonomously, it must have some knowledge of the environment. A platform with ample sensors and compute power is able to perform Simultaneous Localization and Mapping (SLAM) online to create a map and estimate its position on that map. A less powerful robot could be provided with a map of the environment, which it can use along with its sensors to estimate its position. What if a decent map is not available, though?

Enter Coarse Map Navigation (CMN) [1], a pipeline for using a rough, low-resolution map of an environment to perform localization and navigation. This map can be a descaled floor plan, a manually-created low-res image, or even a hand-drawn sketch. Scale and proportional accuracy are unneeded, as this method uses the local topology to understand the world.

We implement CMN on a physical robot, and explore several extensions to make it more generalizable and effective outside simulation. These include expanding the discrete localization to estimate both orientation and position, modularizing the perception stage to compare several different sensors, and developing a continuous CMN algorithm to estimate poses in SE(2) and directly command robot velocities.

Chapter 1

Introduction

1.1 Problem Statement

A mobile robot is one that can move freely in the environment, and is not anchored to a single point like a robot arm is. To accomplish a task, the robot must be able to understand where it currently is, where the goal/task location is, and how it can navigate between the locations [6]. One of the most classic methods to give the robot knowledge of the world is to use an occupancy grid map [4]. The robot could create this map itself using SLAM, or it could generate the map in advance. For a reasonably small and static environment, the latter should work well. If the map specifies the goal location, and the robot knows its position on the map, it can use a simple heuristic planner such as A* to generate a trajectory between the points. However, even if the robot knows its starting location on the map, the real world always has noise, so the robot may deviate from the planned trajectory, and require some kind of feedback (i.e., localization) to correct this. Commonly, a mobile robot will have odometry information generated from its wheel encoders and/or IMU data, which can be used to some extent. However, if the odometry isn't entirely accurate, or in the case where the robot's starting pose is unknown, a more complex filter is required for localization.

With this project, we would like to target usability on as lightweight a system as possible, so complex SLAM algorithms during navigation are out of the question. So, we will focus on the localization-only case. Generally for localization-only, the robot has access to a detailed map of the environment, usually a highly granular occupancy grid that may have been generated previously by some form of LiDAR-SLAM (such as Cartographer-ROS [2]). This type of map can be very large, on the order of 1000+ pixels per side, with each pixel corresponding to around a centimeter. (In a simulated project, it can use the ground truth occupancy map, which can be specified at any

resolution.) The robot would then use its sensors to extract measurements from the environment, such as LiDAR scans, depth measurements, or RGB images. By moving around in the environment and observing different parts of it, the robot is able to localize and determine its position and orientation relative to the given map. One of the most common algorithms to do this is Adaptive Monte Carlo Localization (AMCL) [3], which uses a particle filter and performs raycasting to evaluate LiDAR scans against expected poses on the map.

What if we can't get access to a detailed metric map though? If the robot can't get into the space in advance to run the time-consuming mapping process, or the environment is rapidly changing, how can we localize and navigate? With a poor quality map, methods like AMCL will suffer; too many inaccuracies or a lack of features can throw off the localization result. Further, if the map scale is unknown or the environment has significantly changed since map creation, there is little chance of localizing correctly. A project experiencing this situation may move to performing online SLAM to update the map while running, but this requires a certain level of sensors and compute resources.

The goal of this project is to localize and navigate using a coarse, hand-drawn map that can be quickly created by a person with some basic knowledge of the environment in a few minutes, and easily updated when the environment changes, without needing to send the robot on a lengthy mapping excursion. These maps are characterized by high-level topological accuracy, meaning important features like walls and furniture are represented, but the scale is not specified, proportions may be wrong, and the resolution is very low.

By localizing using the topology of the robot's immediate surroundings, a wide open space gives just as much information as the robot being in a corner. A map could be as small as 20 pixels on each side, and we would be able to determine a set of feasible locations which would dwindle as the robot moves and detects a series of features in the form of a local occupancy grid centered on the robot. When the set of feasible locations has reduced to one, the filter has converged and we can perform path planning and navigation tasks. This approach would also make the filter invariant to a wide range of scale differences, so the provided map would not need any additional metadata such as its pixel-to-meter resolution.

1.2 Background

The basis for this project is a previous work by Xu [1], who created a proof-of-concept using the Habitat simulator. Our work extends this in several ways, namely by implementing it on a physical robot.

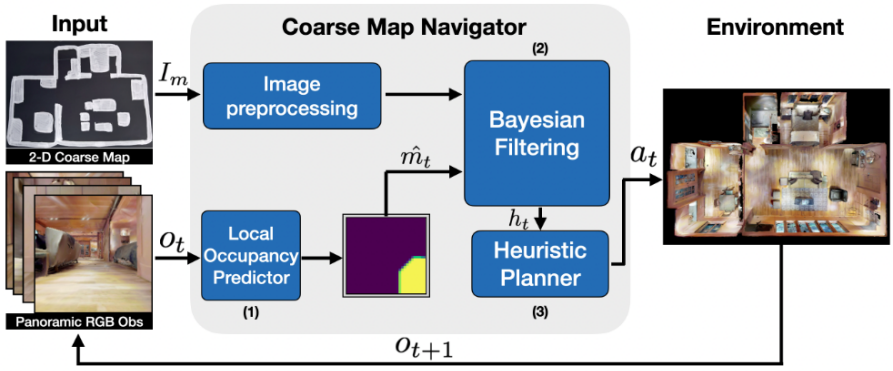


Figure 1.1: Architecture diagram from [1].

In the work by [1], the Habitat simulator is used to gather training data, consisting of pairs of panoramic RGB measurements and the ground-truth local occupancy map. These are perfect panoramas, and the local occupancy is a 128x128 pixel image corresponding to a 1.28×1.28 meter region centered on the robot, obtained from the ground-truth occupancy map. This data was used to train a deep perception model to predict local occupancy from an RGB panorama; we have access to this trained model, and will be discussing how it performs for our use case.

The state space and action space are discrete, with the robot pose being represented as a pixel location on the coarse map, and the set of actions being to turn left (CCW) or right (CW) by 90 degrees, or to move forward a set amount (0.15 meters). The robot orientation is assumed to be known, both during training and during all runs. A discrete Bayesian filter is used to estimate the robot location. From this cell, a topological graph of map regions is used to do path planning to the goal location.

Xu was able to show that this performs well in the Habitat simulator, outperforming baseline methods that aren't equipped to handle low-resolution or hand-drawn maps.

1.3 Related Work

There have been some other papers utilizing topological localization methods to subvert the problems of unknown map scale, proportional inaccuracies, or dynamic environments. Few projects use hand-drawn maps, but some still use low fidelity maps such as floor plans.

The main project comparable to CMN is from Behzadian et al, who were able to use MCL on hand-drawn maps with a reported 80% success rate [5]. In that work, they perform localization in

pixel coordinates on the hand-drawn map, similarly to CMN. However, they also extend the state space to estimate the local deformation of the hand-drawn map from the expected high-granularity metric map; this increases the dimensionality, since the map scale must be estimated. Another work by this group uses hand-drawn maps effectively, but assumes the initial robot location is known [14]. A recent paper explores using a SLAM algorithm to match hand-drawn maps to the currently observed metric local occupancy grid, which works well but also requires the initial robot position is known [15].

Another project from Koenig et al addressed this issue from a more ML-focused approach, using an unsupervised learning algorithm to learn sensor and control models so the robot can interact with the environment using only an abstract topological map [7]. This approach essentially annotates a provided topological map with distance information to solve the unknown scale problem, which they were able to demonstrate effectively when a good topological map is available.

Ito et al used sparser floor plan maps for localization, so furniture and other temporary obstacles are absent [8]. They also used WiFi information to be able to treat the state space as Gaussian, which led to an improvement over the non-Gaussian MCL solution. Another paper that uses blueprints is from Setalaphruk et al, who use a Voronoi diagram approach to split the metric floor plan map into topological landmarks such as intersections [10]. This method assumed the scale is known and the proportions are correct, so we cannot use the same method for our case.

Some other methods use the topological map to communicate more information to the robot to avoid needing to localize at all [9] [12] [11] [13]. These types of works do not address using hand-drawn maps, and cannot be used for more complex autonomy where the robot needs to be able to localize to adapt and act robustly in the environment.

Chapter 2

Implementation

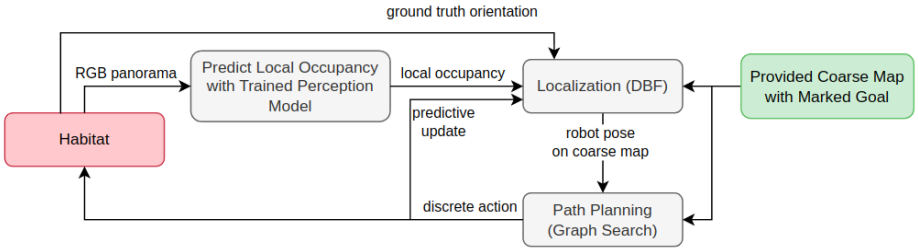


Figure 2.1: Our interpretation of the original CMN architecture.

2.1 Simple Simulator

To verify the localization and navigation process, disjoint from the perception stage, we create a simple simulator that can be used in place of Habitat. Each box in the following architecture diagram is implemented as a Python script, which can be run in ROS when we need to communicate with the physical robot.

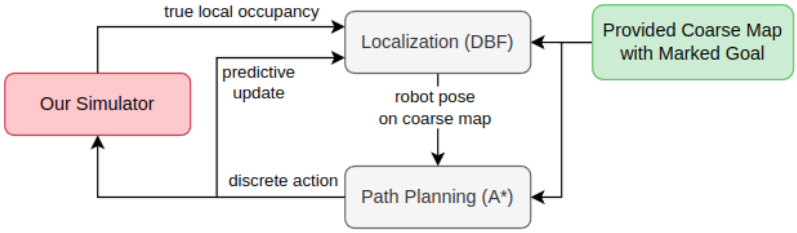


Figure 2.2: Our first implementation of the CMN architecture in a custom simulator. We maintain the discrete state and action spaces, and skip the perception stage (for now).

This simulator allows us to implement the localization and path planning components first, and test them in the best case scenario where the perception and motion planning stages are assumed to work perfectly. We are also able to progressively degrade the quality of the coarse map or local occupancy ”predictions” to test the limits of the method.

2.2 Discrete Localization

For our discrete localization method, we use the same approach as the base work [1], using a Discrete Bayesian Filter (DBF) to estimate the robot location as a cell on the coarse map. We initialize the filter with equal probability for all free cells on the coarse map. The first version will maintain the assumption that ground truth orientation is known, so we need only estimate position.

The predictive update stage will propagate cells on the belief map over by one, in the direction the robot is facing. Since the robot position is tracked at the cell level on the coarse map, but we assume the scale is unknown, there is a chance that when the robot moves 0.15 meters forward in the world/simulator, it is still within the same pixel on the coarse map. To account for this, the predictive belief map is a combination of the current ”stay” probability added to the shifted ”move” probability. We also stamp out cells that are marked as occupied on the coarse map to probability 0.

Note that the coarse map could be drawn with furniture such as tables marked as occupied, which a robot would be able to drive underneath; this means technically the robot is able to be within occupied cells on the coarse map. We could design the filter to account for this, but the approach we choose is to ensure the initial robot position and the goal cell are not in occluded cells, and to use incoming sensor data to prevent the robot from entering occupied cells at all times. We will see that this works sufficiently well, and in our lab environment we’re able to guarantee the robot always remains in areas of the coarse map that are marked as free.

After receiving the local occupancy prediction, the known robot yaw is used to rotate it to align with the global coarse map. Then, it is compared with every 3x3 region on the coarse map (using mean-squared-error) to evaluate the alignment for every location on the map; this gives a measurement likelihood map, where high probability is assigned to locations where the predicted local occupancy aligns well with the map.

The predicted belief map and measurement likelihood maps are combined and normalized to give an overall belief map. The cell with the highest probability is taken as the robot position estimate; since this is a non-Gaussian representation, no information about the initial robot position is needed.

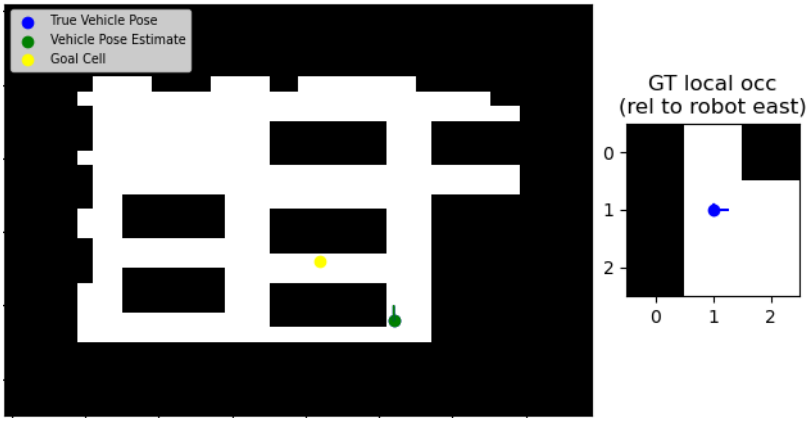
2.3 Discrete Action Selection

After the localization stage runs, we have not only a robot pose estimate, but a measure of confidence in this estimate; if the highest value in the normalized belief map is small (e.g., $p_{\max} < 0.05$), there are many similarly probable position estimates. If we plan a path from one of these estimates, it's likely the position estimate will keep jumping around, and the robot may end up driving back and forth in a loop. To avoid this, we attempt to explore the environment with a simple heuristic until the confidence is sufficiently high ($p_{\max} > 0.8$).

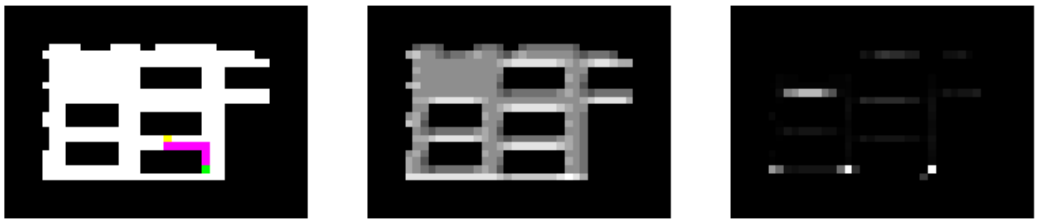
For exploration, the robot will drive forward until seeing a wall, upon which it will randomly turn left or right. This could lead to back-and-forth behavior in certain situations, such as coarse maps with isolated hallways, but is generally sufficient. We must be careful not to make the condition for ending exploration too strict, as there may be a cluster of likely cells which accurately represent the true robot pose, and yet the robot would keep driving around aimlessly instead of planning a path to the goal.

Once the filter estimate has converged sufficiently, we use an A* implementation to plan a path to the goal on the coarse map. It is fairly trivial to convert a global path to a series of turns and forward motions. If the filter estimate changes during traversal of the path, it is fine, as the path is replanned every iteration regardless. Since the coarse map is used for both localization and path planning, it is fairly low resolution and thus has no trouble running quickly each iteration.

On reaching the goal cell with the filter converged, the simulator will choose a new goal cell to allow the process to keep running.



(a) Coarse map with the true vehicle pose and the local occupancy.



(b) Coarse map showing the current planned path (left), the current observation belief map (center), and the current state of the overall belief map (right).

Figure 2.3: Our simulator with a run in progress. We can see that localization is working.

2.4 Processing Hand-Drawn Maps

The simplest way to generate coarse maps is to ”draw” them manually using a graphics program such as GIMP at the same low resolution they will be used at, using the pencil tool to prevent blending and anti-aliasing. For a map drawn with pen-and-paper, we need a pipeline to transform an image of that map into a low-resolution representation of it.

We convert the image to grayscale, and threshold all pixels closer to black as occupied, and those closer to white as free. We also flag transparent pixels as free, in case a format such as PNG is used that may mark white pixels as transparent. Finally, we downscale the image from its native resolution (which is likely very high if it was taken with a camera or scanned in) down to a more usable regime. For this, we use OpenCV’s built-in area-based interpolation to achieve the desired resolution specified in our config file.

We obviously do not require the exact pixel-to-meter resolution to be known, or for the

map to be proportionally accurate for that matter; a ballpark estimate of the scale is only provided so the map becomes reasonably useful, and is not compressed so much that all features are ruined. Since a person is drawing the map, we expect it is also reasonable for a person to spend a few seconds running the processing on a handful of different scale parameters to choose one that performs decently. We will discuss in the next section how robust the program can be where the scale is concerned. The person may also need to tweak the thresholding parameters if they have drawn the map in a color scheme other than black-on-white.

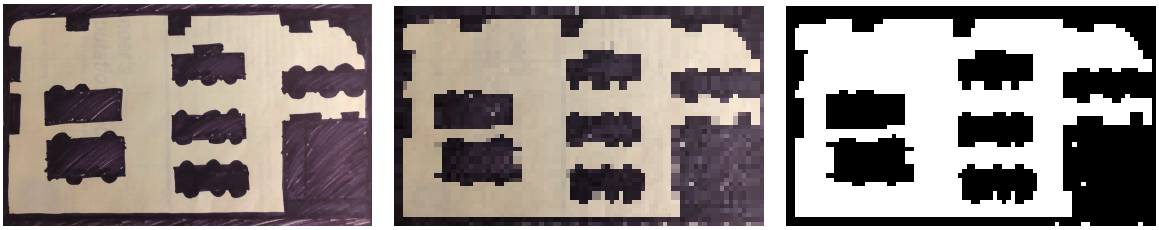


Figure 2.4: Processing a hand-drawn map into a usable binarized coarse map.

Some cleanup on the thresholded map gives us the final low-res map we will use and downscale further in the rest of the project. A less messy hand-drawn map could avoid the artifacts we see from stray sharpie lines and poor filling of large areas. CMN would probably work fine even without this extra cleanup step, but for the sake of neater diagrams and more direct path planning, we perform it regardless.

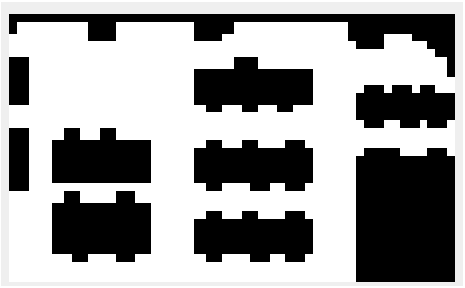


Figure 2.5: The final processed coarse map of our lab.

2.5 Coarse Map Scale Invariance and Limitations

We expect a local occupancy measurement to correspond to a 1.28×1.28 meter area centered on the robot, and it will be 128×128 pixels in size, so each pixel corresponds to 1 real-

world centimeter. When comparing to the coarse map, we extract a 3×3 pixel area centered on the robot cell, which is scaled up to 128×128 pixels so it can be compared element-wise to the local occupancy grid. If the coarse map resolution exactly matches the real-world, this means each pixel should correspond to $1.28/3 = 0.43$ meters.

However, the main purpose of CMN is to use whatever map we have, which could even be hand-drawn, so the dimensions being exactly this are unlikely. So, we can use our simulator to test a variety of different resolutions to explore the limits of our process as the resolution becomes very small or very large.

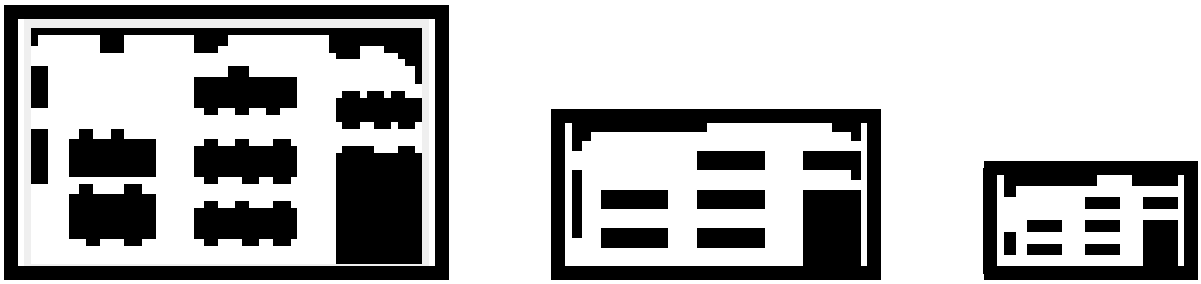
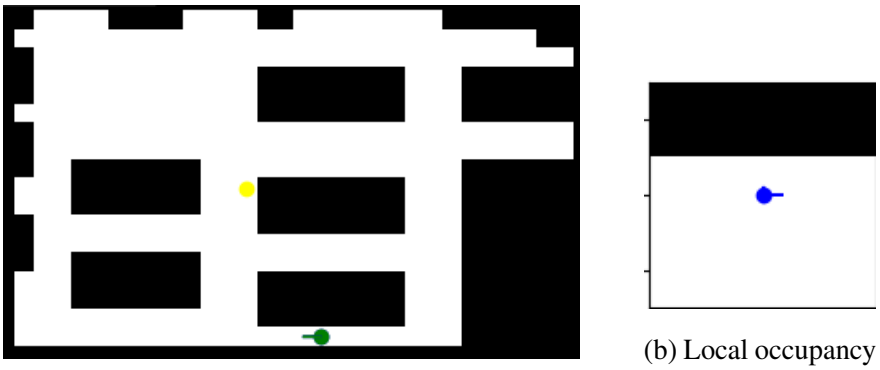


Figure 2.6: Maps of the same environment in a variety of granularities.

It seems the algorithm works well despite scale differences, so long as the scale is reasonably close. When the coarse map is egregiously small, such that the topology of the environment itself changes (i.e., passageways or small obstacles disappearing), the local occupancy grid will differ substantially from the true location on the coarse map, making localization with this method extremely difficult. Additionally, path planning may become impossible if doorways or aisles become sealed off by downscaling, segmenting the environment into unreachable regions. Even aside from these drastic changes, a robot in a wide hallway can only see one wall at a time, but a coarse map that represents the hallway as 1 pixel wide will expect it to be able to observe both walls in a single measurement.



(a) Map with a very coarse resolution that has compressed some hallways to 1 pixel wide.

(b) Local occupancy showing the robot is next to a wall.

Figure 2.7: Example of a map so rough that the local topology is not preserved in some places.

On the other hand, there seems to be no limit to how large the coarse map can be. Consider the robot driving alongside a wall; the local occupancy grid will simply reflect the fact that one side of the robot is occupied. This means the belief map will have high likelihood in the cells near a wall. Since our predictive update step in the filter will stretch the belief across multiple cells (due to the combined ”move” and ”stay” probabilities), it’s fine for there to be many cells on the coarse map within the area captured by the local occupancy.

The only artifact of an extremely granular coarse map is that the robot pose estimate may be pulled closer to walls than it perhaps should, but this has minimal impact on path planning effectiveness.

Since large-scale features like walls and corners are the primary contributors to localization, this means we’re able to effectively converge on a reasonable robot pose estimate in most cases.

2.6 Necessary Additions for CMN on a Physical Robot

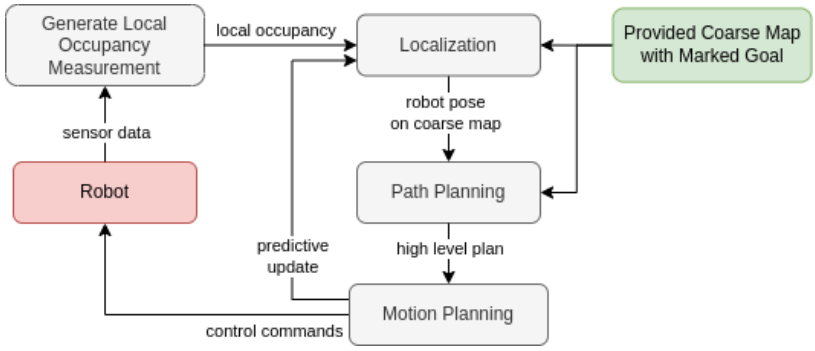


Figure 2.8: High-level architecture for the project.

We use the LoCoBot from Trossen Robotics, which is a turtlebot platform with a Kobuki base, a RealSense RGB-D camera, a 360 degree planar LiDAR, and a 5-DOF manipulator arm (which we won't be using).

Figure 2.9: Robot we're using for the project.

Some aspects of the original CMN by [1] were fine in Habitat, but will require additional attention to work on a physical robot. This includes the discretized state and action spaces, deterministic actions, knowledge of the ground-truth orientation, and easy access to panoramic RGB measurements.

Additionally, we will be testing the same deep perception model, which was trained in Habitat, and may not translate well to real-world data.

2.6.1 Discrete Motion Planner

The CMN algorithm uses the set of discrete actions ”move forward”, ”turn left”, and ”turn right”. The forward action has a set distance, and the turns are 90 degree in-place pivots. In a simulator, it’s simple enough to choose a discrete action and assume it has completed perfectly. This means the robot can always be perfectly represented in a discrete context, and its orientation is always locked to a cardinal direction. In the physical world, there is noise in all aspects, in particular the commands sent to a robot will not be executed perfectly. Assuming they are is called ”dead reckoning”, and is one of the most simple yet ineffective control methods, as the robot will rapidly accumulate drift until the real and estimated trajectories differ so drastically that further path planning is meaningless. Additionally, our robot cannot understand the discrete actions directly, so we must convert these into continuous velocity commands to achieve the desired motion. We want our other modules to be able to request discrete actions and have confidence they were completed properly, so we introduce the motion planner node as an intermediary control layer.

To command a relative motion, we subscribe to the internal robot odometry and save the current pose. Then, we begin commanding velocities by publishing Twist commands at a set frequency. We keep receiving robot odometry, and command zero when the motion has completed. At its most basic level, this has problems with inexactness of motion, which we need to minimize as much as possible to maintain a discrete representation.

2.6.1.1 Pivot Controller

For an in-place pivot, there will be two main types of drift:

- Overturning due to sudden stop when the goal angle is achieved. Also, slip due to sudden acceleration at the start.
- Drift in the robot odometry itself due to slippage/drag and other noise factors.

Turning very slowly for the entire duration will basically remove the first source of noise, and make slippage less likely, but we want this to be as practical and usable as possible. So, we define a velocity curve which ramps up the speed from 0 to the max velocity, and then ramps it back down to 0 by

the end of the motion. This is done with a simple P-controller, which is essentially an acceleration limiter. We specify a minimum and maximum angular velocity in the config file.

$$\omega_k = p \cdot \omega_{k-1} + (p-1) \cdot \frac{|\Delta\theta_{remaining}|}{|\Delta\theta_{command}|} \cdot \omega_{max}$$

The twist command sent to the robot is then

$$v = 0$$
$$\omega = \max(\omega_k, \omega_{min}) \cdot \text{sign}(\Delta\theta_{command})$$

The minimum speed is needed because commanding too small of a velocity will not provide enough torque for the motors to actually move the robot.

We have an additional check for overturning; if

$$\text{sign}(\Delta\theta_{command}) \quad != \quad \text{sign}(\Delta\theta_{remaining}),$$

we have pivoted past the goal, so immediately halt by setting $\omega = 0$.

This method allows faster turning while minimizing slippage and overturning. This also reduces the influence of the particular floor surface the robot is on, as these sources of drift depend heavily on the floor (i.e., slip is much greater on smooth tile, but the caster wheel drags on carpet).

Even with this velocity controller, we still see drift over time as the number of turns in the run increases. This is because the robot yaw at any given time is not highly controlled, and commanding relative pivots over and over again causes any noise in the starting angle to build up rather quickly. So, another improvement for our discrete pivot controller is to use global yaw as a control input, rather than remaining distance to turn.

Since we know the robot angle is represented in our algorithm as a cardinal direction, we can compute the global yaw equivalent to this. Assuming the robot is initially facing East, we reset the robot odometry when our algorithm starts; i.e., we take the first robot odometry message, and use it as a transformation on all future odom messages to use it as the origin. This guarantees the first pose angle is 0, which by the radians convention is East.

$$\theta_{init} = 0$$

Now let's say our algorithm has progressed for several iterations, and the robot is facing East again. The odometry could be a few degrees off from 0 in either direction,

$$\theta_t = 0 + \epsilon, \quad \text{for small } \epsilon \in \mathbb{S}$$

If we want to turn left by commanding a 90 degree turn, $\Delta\theta_{command} = \pi/2$, we will end up a few degrees off from exact North.

$$\theta_{t+1} = \theta_t + \Delta\theta_{command} = \pi/2 + \epsilon$$

If instead of commanding 90 degrees, we command the exact angle needed to reach North,

$$\Delta\theta_{command} = \theta_{desired} - \theta_t = \pi/2 - \epsilon,$$

then the resulting angle will be

$$\theta_{t+1} = \theta_t + \Delta\theta_{command} = \pi/2 - \cancel{\epsilon} + \cancel{\epsilon} = \pi/2$$

This method ensures that after every turn, the robot orientation will be as close to the desired cardinal direction as it can be. Any drift remaining comes from drift in the Kobuki base's internal odometry calculations, which is fairly small; further improving this would require additional filtering with other sensor data, so we leave it here for this project.

2.6.1.2 Forward Motion Controller

For a forward motion, it is not as important to directly achieve the desired motion. Recall that we assume the scale of the coarse map is unknown, and to compensate we stretch the belief over both the current and shifted belief maps. Due to this, it makes very little difference if the robot moves a few centimeters more than expected.

However, it is important that during a forward motion, the yaw is preserved. If we immediately jump up to max linear speed, and halt immediately to 0 at the end, it is very likely there will be some angular slippage as the robot brakes unevenly. To prevent this, we also ramp the linear speed up from 0 to max speed, and back down to 0 at the end.

$$v_{target} = \begin{cases} v_{max} & \text{if } \frac{|\Delta x_{remaining}|}{\Delta x_{command}} < 0.5 \\ v_{min} & \text{otherwise} \end{cases}$$

$$v_k = p \cdot v_{k-1} + (p-1) \cdot v_{target}$$

The twist command sent to the robot is then

$$v = \max(v_k, v_{min})$$

$$\omega = 0$$

Similarly to pivots, a minimum speed ensures the motors will have enough torque to achieve the motion.

With this control layer, we are able to treat the physical robot as a discrete agent, and we can control it in the same manner as the simulated Habitat agent in the original project.

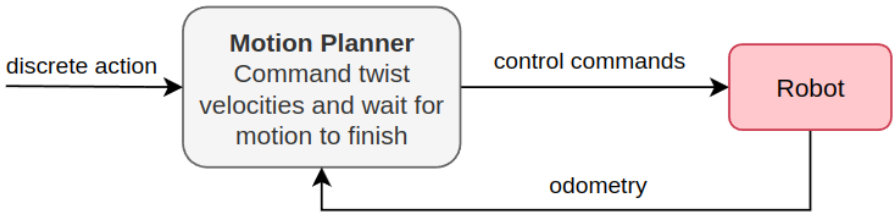


Figure 2.10: Motion planner interface allows other nodes to command discrete actions directly.

2.6.2 Panoramic Measurements

In the Habitat simulation, it's possible to extract a perfect RGB panorama at any time, centered on the robot. This is assuming the agent had a 90 degree FOV RGB camera on each side, which it could get a measurement from at any time and concatenate into a panorama. With the physical robot, we cannot support the data feed of four RealSense cameras at once; even if we had the compute onboard the turtlebot, having four dedicated USB 3.0 ports to handle the sensor inputs from each RealSense (including RGB, left+right grayscale, and depth feed) would be a heavy ask. This is an unreasonable requirement for this project, which aims to be usable by a computationally lightweight system.

We use a single RealSense D455 camera, which has nearly a 90 degree horizontal field-of-view. To generate a panoramic measurement, we perform four 90 degree turns to capture an image for each cardinal direction. This must be done every iteration to generate the input for our algorithm, which is another reason it was so important to create a minimal-drift-controller which is also as fast as possible without sacrificing accuracy. The panorama is the concatenation of the four measurements, in the order (front, right, back, left).

When the robot pivots in-place, we can use the previous measurement shifted to the new orientation, rather than slowly pivoting again. However, we are forced to collect a new panorama after every forward motion. This makes a run with the physical robot much slower than in simulation.

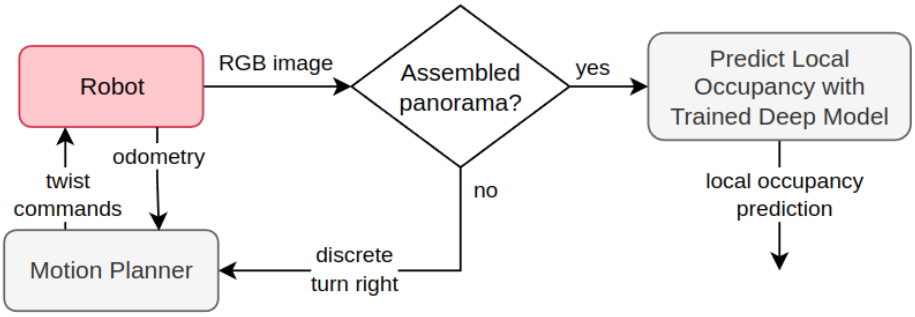


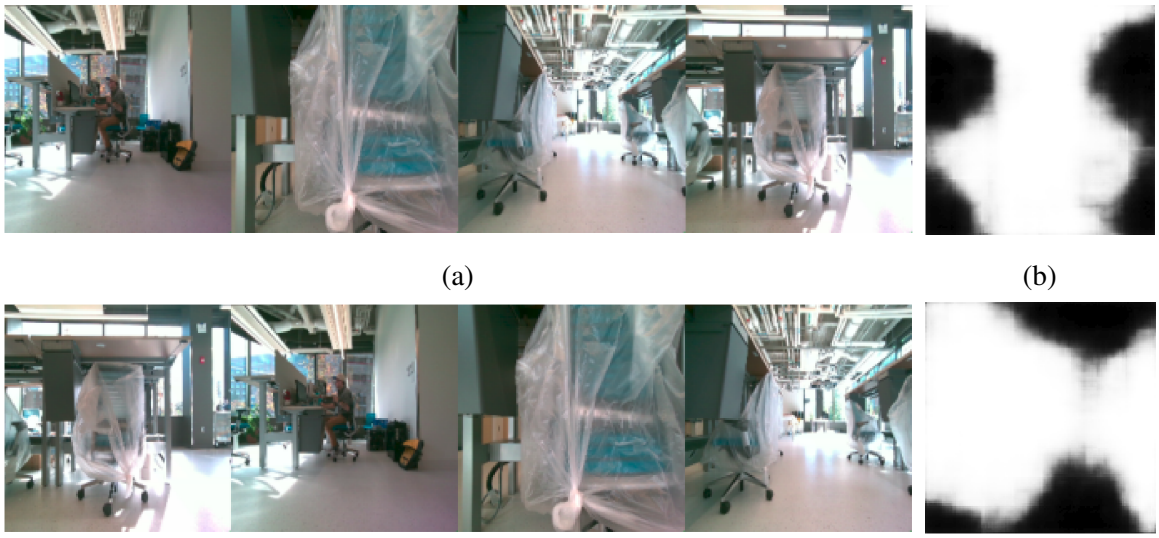
Figure 2.11: Process of predicting local occupancy on the physical robot.

2.6.3 Perception Model Performance

The original deep perception model from [1] was trained using images from the Habitat simulator, which are fairly realistic but not identical to reality, and not representative of all possible environments. Its performance will vary depending on the physical space we're trying to use; in our lab at Northeastern University, there are some main differences from Habitat, including a white floor and large wall-to-wall windows. We can accommodate this somewhat by lowering the shades, but in general this demonstrates the problem of domain shift, where the real-world environment is too different from the training data used for the model.

Running our CMN implementation with the existing perception model yields the results shown in the following figures.

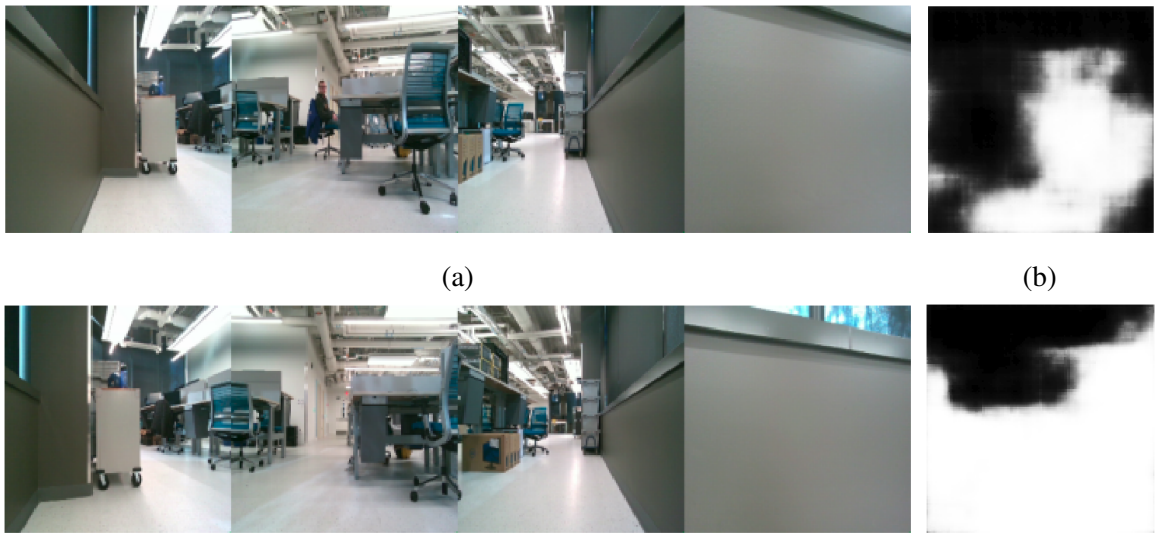
The following figures show examples of panoramas we were able to collect on the robot, and the resulting predictions. Figure 2.12 shows a panorama taken in between rows of desks with chairs. We see the predicted local occupancy is fairly good. Figure 2.13 shows two separate panoramas that were taken from the same location in the environment; despite their similarity, the predictions vary wildly in quality.



(a) (b)

(c) (d)

Figure 2.12: RGB panorama (a) and its predicted local occupancy (b). Shifting the panorama by 90 degrees (c) yields a new prediction (d), which is similar but not exactly the same as if the original prediction was rotated. This demonstrates imperfect equivariance in the model.



(a) (b)

(c) (d)

Figure 2.13: RGB panoramas and their poor local occupancy predictions.

If we want this project to run in a variety of environments and continue predicting local

occupancy from RGB panoramas, we will need to retrain this perception model with a more robust dataset. It may suffice to take data from many different Habitat (or other simulated) environments, emphasizing different floor and wall colors.

We do have an architecture to gather training data in the physical environment, but due to the slow pivots when gathering panoramas, it would be painfully slow. In any case, the process would involve first gathering a "ground-truth" occupancy grid of the environment, using something like Cartographer-ROS to perform LiDAR-SLAM. We can then drive the robot around in the environment, recording RGB panoramas, and use the known robot pose to extract a ground-truth local occupancy grid. This will give the needed input-output pairs to add to the training data for the perception model.

We will also explore other methods of generating local occupancy measurements that do not use the RGB perception model in Section 3.2

2.6.4 Avoiding Collisions

In simulation, the assumption is generally that when the robot would move into a wall, it will instead stay where it is. This is not so clean in the physical world, as the robot running into something will easily change its orientation, completely ruining the discrete, axis-aligned representation our controller is designed to preserve.

Instead of allowing the robot to run into something, where physics would prevent it moving into an occupied cell, we can use our local occupancy prediction to determine if the space in front of the robot is occupied. This is done by downscaling the local occupancy to a 3x3 and checking whether the value of the center cell on the side the robot faces is ≤ 0.75. If so, this region is at least 25% occupied, so we prevent the "move forward" action. Ideally our algorithm will not choose the "move forward" action in the first place; however, if that action is chosen, we will skip commanding the motion to the robot, as well as the predictive update stage in the DBF.

Since the local occupancy predictions are not completely accurate, as we have discussed, this could prevent motion when there actually isn't anything in front of the robot; we will address this in Section 3.2.1.

2.6.5 Tracking Robot Orientation

The original version of CMN requires that the ground truth yaw is always known. With this discrete architecture described so far, we can guarantee this by requiring the initial robot orientation

is always East. Given this starting condition, we can propagate the known robot orientation by each discrete pivot to ensure it is known at all times.

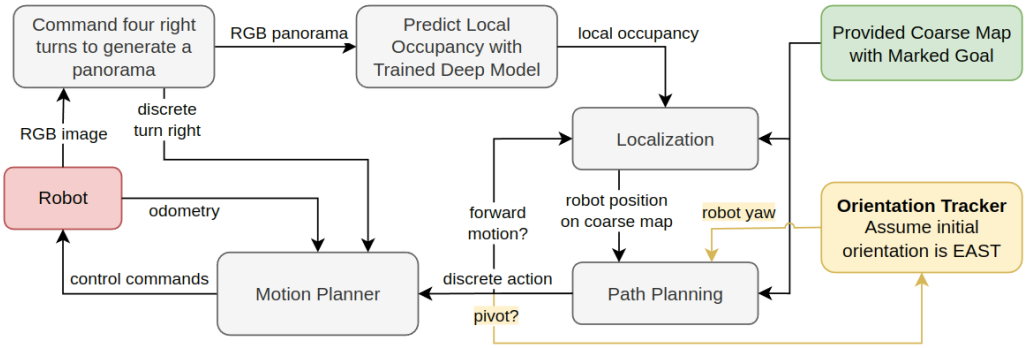


Figure 2.14: Architecture for running discrete CMN on the robot, assuming yaw is known.

With this architecture, we can run CMN on the robot and evaluate its performance.

(a) RGB panorama and local occupancy prediction (b)

(c) Measurement likelihood map (d) Overall belief map (e) Coarse map w/ estimate

Figure 2.15: Visualization while discrete CMN runs on the LoCoBot.

Chapter 3

Extensions

3.1 Extending Localization to Estimate Yaw

In the discrete state space, there are only four possible orientations: the four cardinal directions. So, we can add a dimension to the discrete Bayesian filter to estimate orientation in addition to position. To do this, we duplicate each step of the belief calculation onto four layers, with each layer corresponding to a different possible global robot orientation (north, east, south, west).

For the measurement update step, we take the local occupancy map, and compare each of its rotations to the coarse map to produce four observation belief maps. These are applied to the belief map's layers as usual.

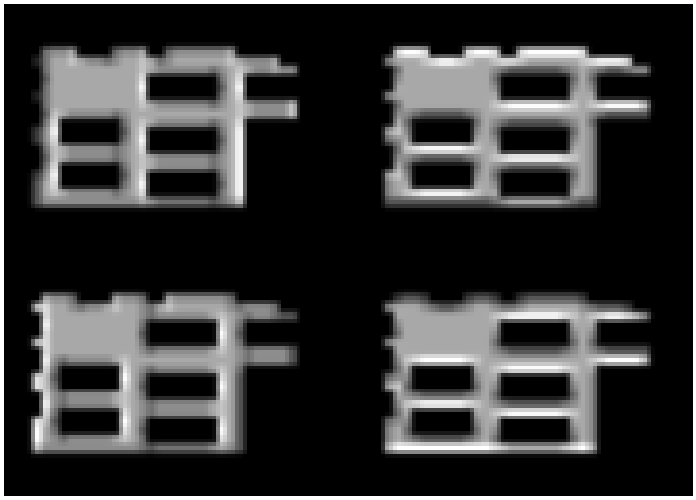


Figure 3.1: Observation belief for all four possible robot orientations.

For the predictive update stage, if the robot moves forward, the usual forward propagation is applied to each layer of the belief map, with the propagation direction corresponding to the robot orientation for that layer. If the robot pivots in place, the layers of the belief map are cycled. So, if the robot turns left by 90 degrees, the belief map for East becomes that for North, North becomes West, etc.

Figure 3.2: The belief map after increasingly many iterations. We can see that after convergence, the belief map for South (shown in the bottom left quadrant) has the highest probability, so its position estimate is used, and the estimated orientation is South.

This increases the size of the state space, leading to slower convergence, but removes the requirement that the initial orientation always be East.

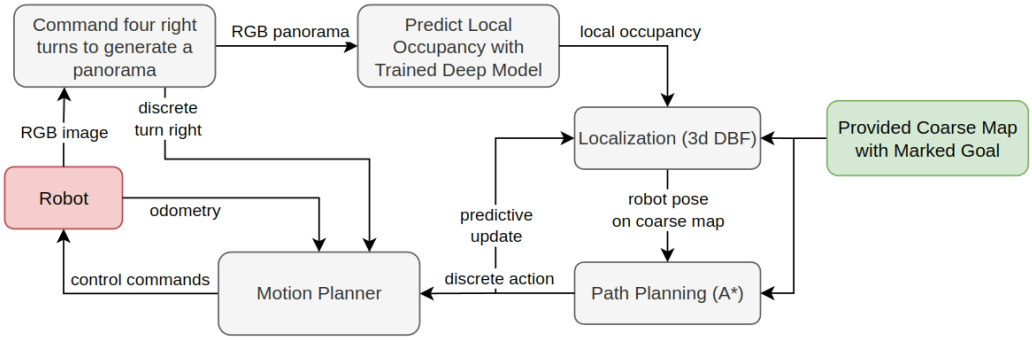


Figure 3.3: Architecture for running discrete CMN on the robot, estimating yaw as well as position.

3.2 Alternate Perception Methods

As we have discussed, the perception model which predicts local occupancy from RGB panoramas does not perform the best in our real-world environment. In this section we will explore

using alternate sensors to produce the local occupancy grid that is needed as the measurement input to the CMN algorithm.

3.2.1 Local Occupancy from LiDAR

The most obvious alternate approach to observing local occupancy is using the 360 degree planar LiDAR sensor. This will easily give us a measurement all around the robot without requiring the slow series of in-place pivots that were needed to generate RGB panoramas.

From the LiDAR, we receive LaserScan messages containing distance measurements at designated angle increments. By creating a grid of the same size and resolution as the expected local occupancy predictions, we can convert each hit in the LiDAR data into a cell on the local occupancy map.

We initialize a 128×128 grid of ones (i.e., free space). The robot position is the center, with indexes

$$(r_{center}, c_{center}) = (63.5, 63.5).$$

For each point in the LaserScan message, we have a corresponding angle ϕ in radians and distance d in meters. If this ray corresponds to a non-detection, the distance value may be set to 0, or to a very large distance, depending on the particular LiDAR device. So, we discard any rays with distance of 0 or greater than the max range that could appear on the local occupancy grid,

$$d_{\max} = \frac{\sqrt{1.28^2 + 1.28^2}}{2} \approx 0.9 \text{ meters.}$$

We convert each valid measurement into a cell on the local occupancy grid, such that the robot is facing East, with angles increasing counter-clockwise. We use the known resolution $r = 0.01$ meters/pixel to convert all distances to pixels.

$$r_{hit} = r_{center} + \frac{d}{r} \cdot \sin\phi$$
$$c_{hit} = c_{center} - \frac{d}{r} \cdot \cos\phi$$

We then compute the cell at the max distance along this ray,

$$r_{max} = r_{center} + \frac{d_{\max}}{r} \cdot \sin\phi$$
$$c_{max} = c_{center} - \frac{d_{\max}}{r} \cdot \cos\phi$$

All of these are rounded to the nearest integer to get particular cells on the grid. We then fill all cells along the ray between these two points, using the Bresenham's line algorithm. We set each cell's value to 0, marking it as occupied.

This gives a local occupancy grid that much resembles the predictions from the original CMN, as shown in the following figure.

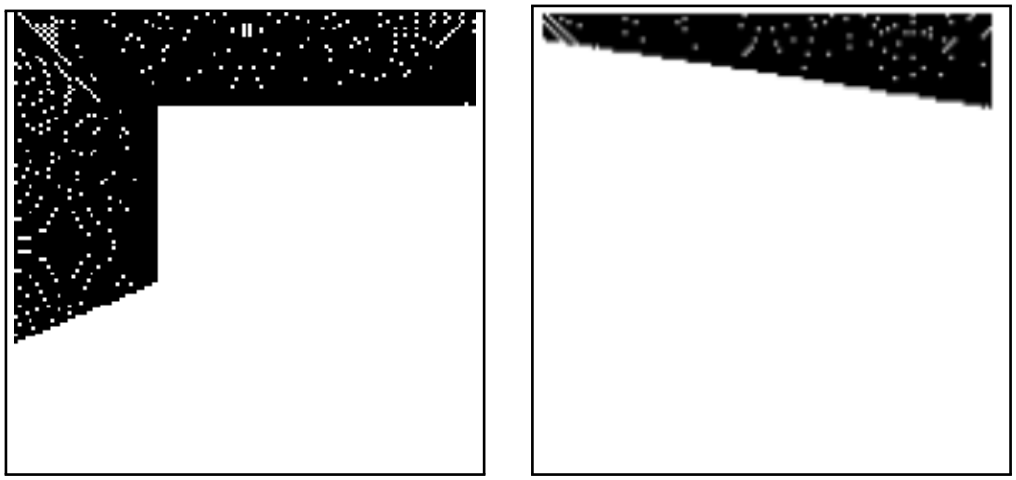


Figure 3.4: Local occupancy measurements generated from LiDAR data.

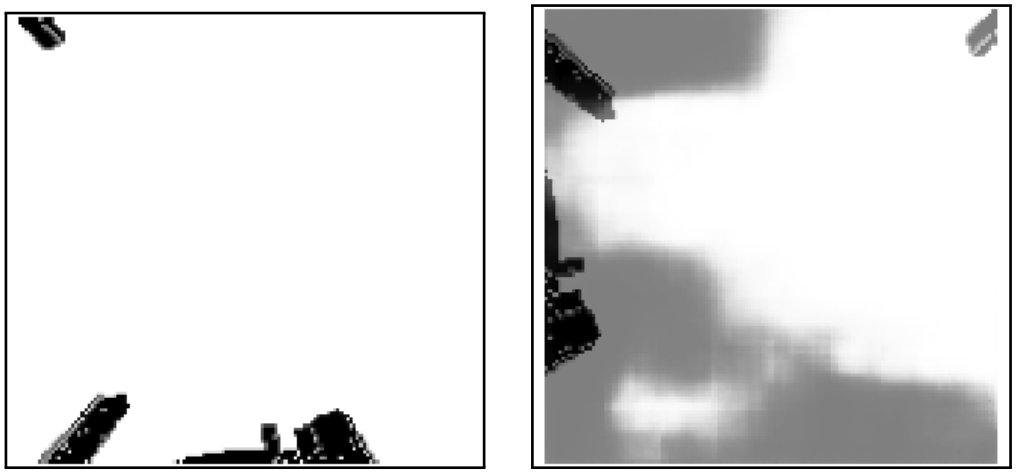
The speckles of white are due to our ray projection being linear instead of conical; a more accurate model may grow wider than 1 pixel as the distance increases, which would prevent these little gaps between adjacent rays. However, since the local occupancy will be evaluated element-wise using a mean-squared error, and the majority of pixels are properly marked as occupied, the inaccuracy due to these speckles is negligible.

When the robot is next to flat walls or similar surfaces, this method works great, giving a much more accurate local occupancy map than the RGB model was producing.

Figure 3.5: CMN running live on the physical robot, using LiDAR to generate local occupancy. The robot's true position and goal were near the north wall, so it was consistently able to get high quality measurements and quickly localize correctly, despite the map scale being unknown.

The major downside of this approach is that the LiDAR measurements all lie within its plane, at about knee height. This means it can only see the legs of tables and chairs, and basically cannot see empty shelves. So, in an area where most of the coarse map represents these types of furniture, this method fails because its view of the world is very different from the map.

The LiDAR data under-represents occupied space according to the coarse map, since it misses everything outside its height level. Conversely, the RGB predictions tend to over-represent occupied space, so perhaps combining the two feeds can be beneficial. This seems like an area worth exploring, but unfortunately does not seem to give any better results than using the RGB predictor alone.



(a) LiDAR measurement which sees only the legs of a table.

(b) LiDAR and RGB prediction averaged together.

Figure 3.6

Having this implemented, we can now use the LiDAR data to prevent the robot from moving forward into a wall or other obstruction, which is a big improvement to obstacle detection over the RGB predictions, which have many false positives. Similarly to the RGB predictions, we take the average occupancy of the middle-third of the East side of the local occupancy (i.e., the area in front of the robot). If this region is more than 25% occupied, we consider the region occluded and immediately halt robot motion. Note that this process can occur much faster than the time it would take to pivot in a circle, generate an RGB panorama, and predict the local occupancy, so we can use these LiDAR local occupancies even during motion. This makes it more reactive to poor understanding of the environment, as well as to dynamic obstructions.

3.2.2 Local Occupancy from Depth Camera

The RealSense camera we use for RGB measurements is also capable of giving depth measurements, which we can convert into a pointcloud and process into a local occupancy measurement. Using this will once again require the robot to pivot in a circle to accrue data on each side of the robot, but due to its vertical field-of-view, it may be able to overcome the weakness of LiDAR data.

We use the ROS `depth_proc` nodelet to generate pointclouds from the raw rectified depth images that the RealSense publishes. This is a slow process, generally taking 500-1000 milliseconds each time. We could avoid this computation time by directly using the depth image data, but we found this data hard to use effectively. Our approach was to treat each column of the depth image as corresponding to the same relative heading, and collapse it into the floor plane, taking the nearest depth measurement as the distance for that ”ray”; from here, the data is processed similarly to the LiDAR LaserScans. Even when excluding zeroes and non-detections (which appear as 65535 millimeters (i.e., max 16 bit integer)), or taking the mean instead of minimum distance, this method just does not produce a good representation of the environment. Adding further processing steps just reduces the benefit of using the raw depth image over the pointcloud, so we leave this thread here.

The following figures shows results obtained from processing the depth images alone.

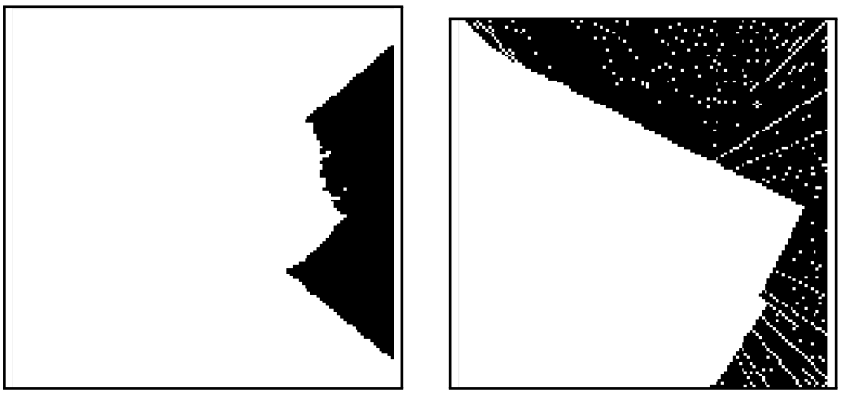
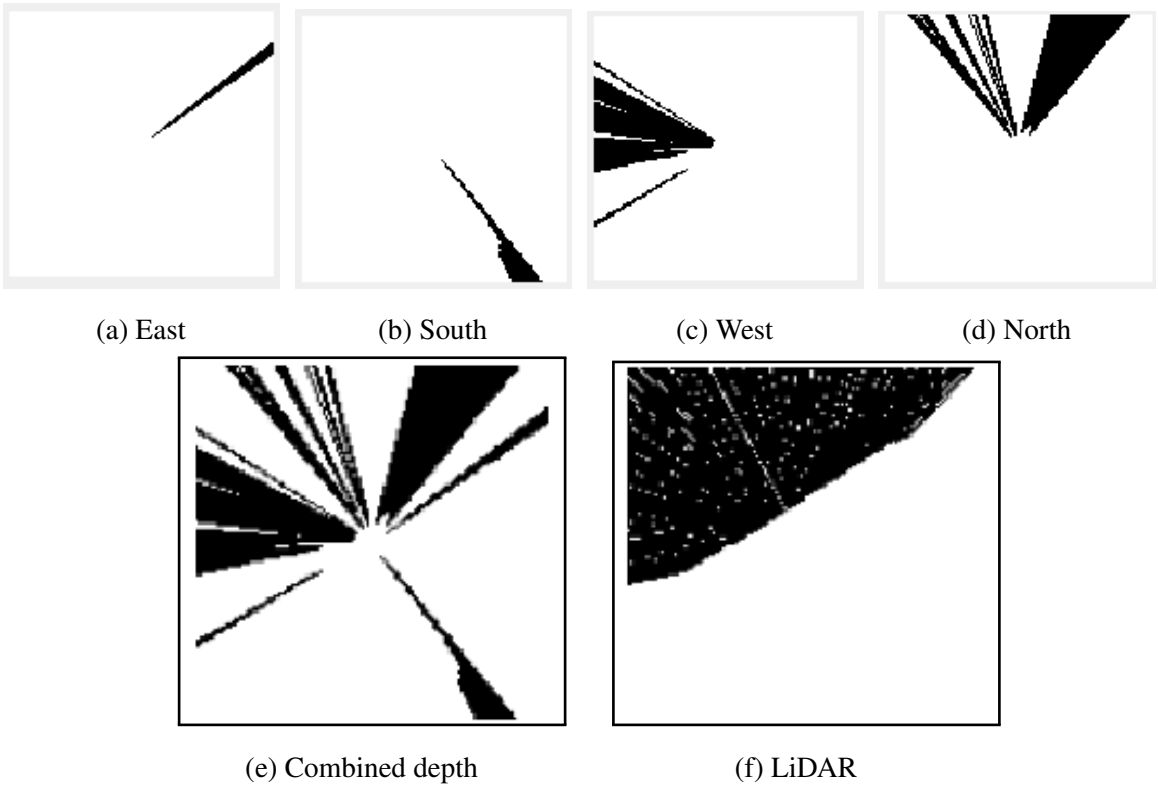


Figure 3.7: Local occupancy from a single raw depth image (left) and the simultaneous LiDAR grid (right) while the robot faces a corner.

We perform this process to generate a partial local occupancy grid, pivot 90 degrees, and repeat until the robot faces the original direction again. Since a value of 1 corresponds to free space, and 0 is occupied space, we can combine them into an overall occupancy grid by rotating each into the same coordinate system and taking the element-wise minimum of the set.



(a) East (b) South (c) West (d) North

(e) Combined depth (f) LiDAR

Figure 3.8: Four depth measurements are taken with a 90 degree pivot between each (a-d) and combined into a local occupancy grid (e). The LiDAR local occupancy at the same position in the environment is also shown (f). The general shape is recognizable through the noise, but overall the depth image result is much worse than the LiDAR.

Figure 3.9: Local occupancy while facing a corner, created by raw depth panorama (left) and LiDAR (right).

Given the poor results from this data source, we return to using the full depth pointclouds.

Since we have already set the robot up to pause after each 90 degree pivot to generate RGB panoramas, we simply add a step to receive a new depth pointcloud at each position. Note: For a runtime improvement, we could use separate ROS nodes so the processing of the pointcloud into local occupancy grid could be done while the robot slowly pivots, instead of one action at a time; we did not implement this, as it would require significant architecture changes, but someone implementing this project from scratch could easily set it up this way.

Similarly to the LiDAR data processing, we instantiate a 128×128 grid of ones (free space), with the robot position being the center at $(r_{center}, c_{center}) = (63.5, 63.5)$. Points in the depth cloud are given in (x, y, z), where $+x$ is to the right, $+y$ is down, and $+z$ is forward relative to the RealSense camera. We discard any points too close to be realistic ($z < 0.01$ m) or outside the distance that will appear in the local occupancy ($z > 0.9$ m or $|x| > 0.9$ m). For any points that remain, we convert them to pixels using the desired local occupancy resolution $r = 0.01$ meters/pixel.

$$\begin{aligned} \mathrm{d}\, r &= x/r \\ \mathrm{d}\, c &= z/r \\ \Rightarrow r_{hit} &= r_{center} + \mathrm{d}\, r \\ c_{hit} &= c_{center} + \mathrm{d}\, c \end{aligned}$$

We compute the angle of this "ray" as

$$\phi = \arctan\left(\frac{\mathrm{d}\, r}{\mathrm{d}\, c}\right),$$

allowing us to get a max-range cell,

$$\begin{aligned} r_{max} &= r_{center} + \frac{d_{\max}}{r} \cdot \sin\phi \\ c_{max} &= c_{center} + \frac{d_{\max}}{r} \cdot \cos\phi \end{aligned}$$

We round these to the nearest integer coordinates, use the Bresenham's line algorithm to get a list of all cells along this ray behind the hit point, and mark all as occupied by assigning them to 0 on the local occupancy grid.

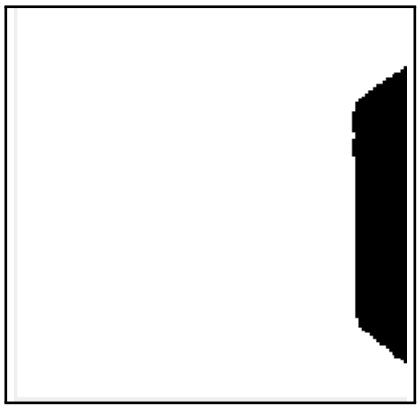


Figure 3.10: Partial local occupancy created from a depth pointcloud. Using the same method described for depth images, the robot will pivot four times to generate the full local occupancy measurement.

The effective horizontal field-of-view of the depth camera is less than that of the RGB camera, so we don't have true 90 degree coverage. This means when we pivot four times to generate the RGB panoramas, we end up missing depth data in the corner regions. Checking the average occupancy of each corner region allows us to infer if that region is likely occupied, and if so, we fill it in. Taking each corner region as $1/3$ of the side length, we can see that about 50% of the pixels are in the gap unobservable by the depth pointcloud. So, this means the maximum average occupancy for these regions is 50%. We introduce a heuristic check that will fill in the entire region with occupied space if the average exceeds 20%. This proves very effective, as the following figure shows.

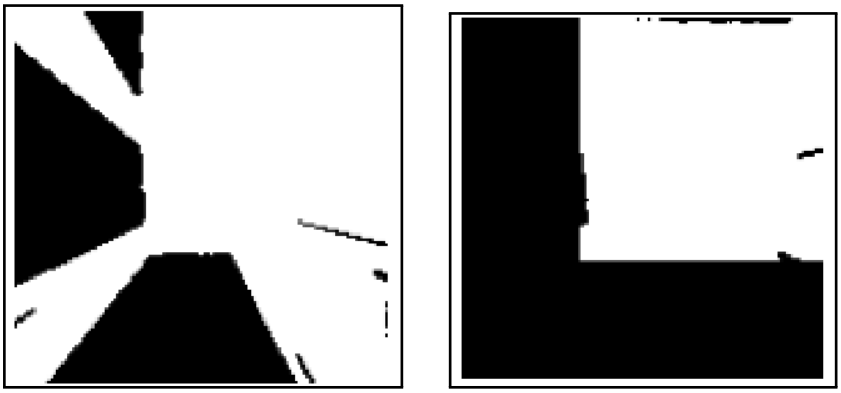
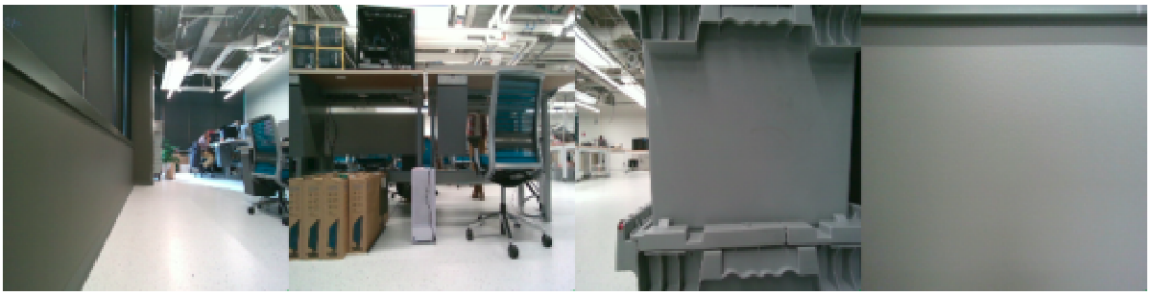


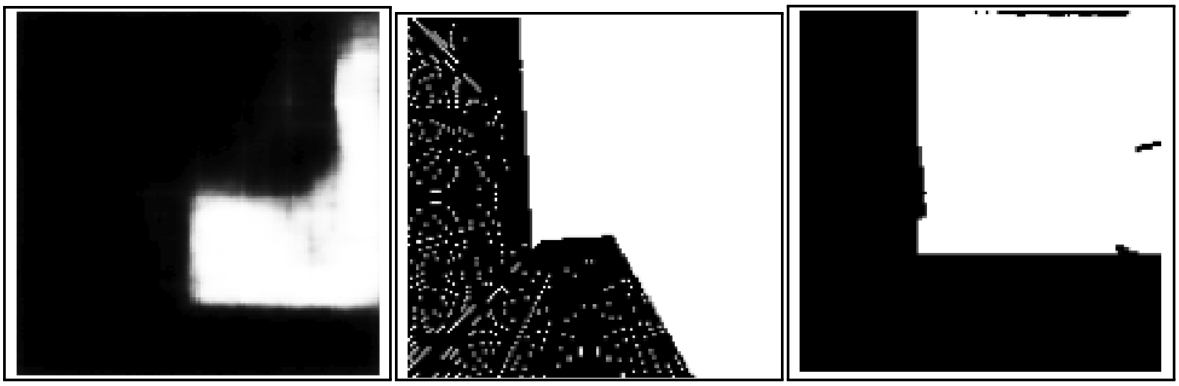
Figure 3.11: Local occupancy from depth pointclouds, with and without our corner-filling method.

We could instead combine the local occupancy from depth and LiDAR, which would fill in

the corner regions without this method of creating data from a heuristic assumption; however, it fails when the corners should be occupied, but are outside the observable plane of the LiDAR. This makes the combined local occupancy miss data in some cases, and basically equivalent most of the time, so we we will continue to use the corner-filling approach.



(a) RGB panorama



(b) Predicted from RGB (c) LiDAR (d) Depth pointcloud

Figure 3.12: Local occupancy measurements created simultaneously by all three primary methods. The LiDAR and depth maps are both great here, while the RGB prediction is unusable.

The local occupancy measurement generated from pointcloud panoramas is the most representative of the environment out of all methods we've attempted (as shown in the following figure), and gives the best performance of CMN. However, since it cannot be generated on the fly, requiring the robot to stop and pivot in a circle, we will continue using the LiDAR local occupancy for reactive obstacle detection.

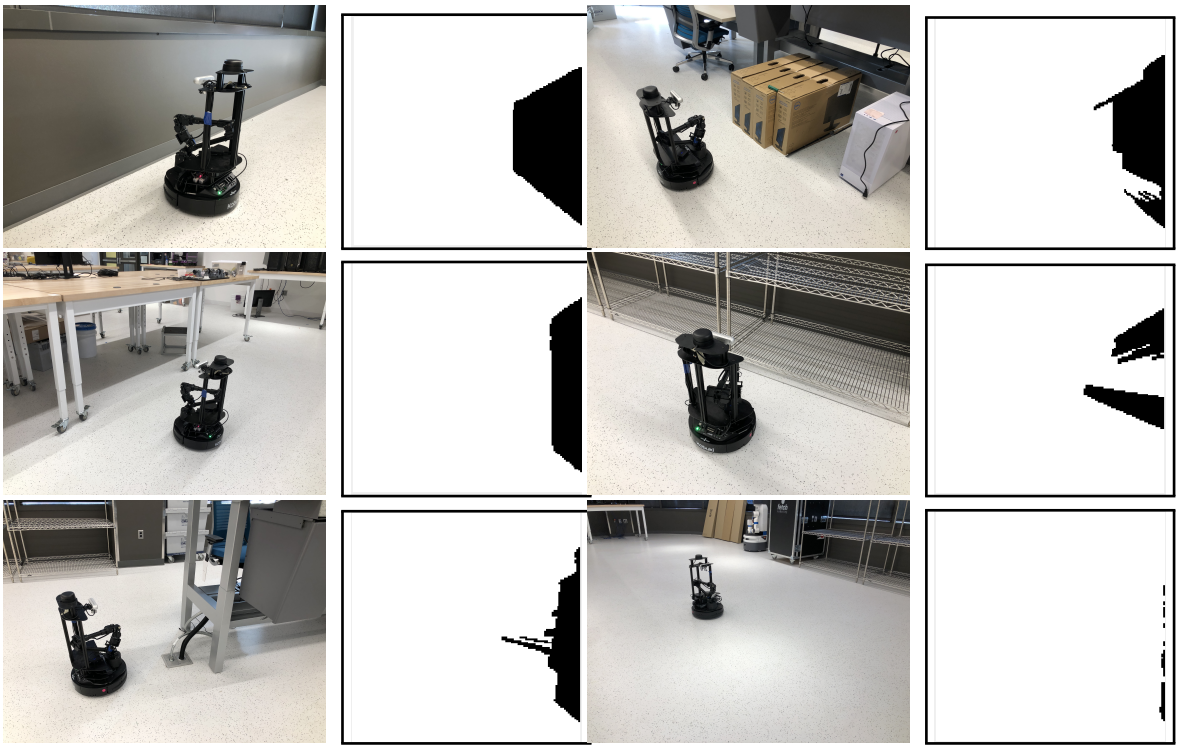


Figure 3.13: Pairs of partial local occupancy created from depth pointcloud, and the robot location in the environment. We can see that it performs great, detecting the boxes and table which are outside the plane of the LiDAR. The worst performance is with the empty wire shelf, but this is still better than the LiDAR result.

3.3 Extending CMN to a Continuous Domain

Even with our carefully designed discrete motion controller, the robot will eventually drift in yaw, making it unable to run forever. Changing the state and action spaces from discrete to continuous will allow us to integrate noise directly into the filter instead of abstracting it away, and will increase the project's generalizability to a wider variety of platforms and environments.

Generation of local occupancy measurements will be the same as previously discussed, with the robot using the discrete motion planner to generate a panorama of RGB or depth data, or using the LiDAR. This will happen once per iteration, and used as an input to the continuous filter.

3.3.1 Localization

Until now, the robot pose has been represented in a discrete space as

$$p = \left\{ \begin{pmatrix} r \\ c \end{pmatrix} \in \mathbb{Z}^2, \quad \theta \in [\text{North, East, South, West}] \subset \mathbb{S} \right\}$$

We can instead estimate it directly as an element of continuous space,

$$p = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2, \quad \theta \in \mathbb{S} \right\}$$

represented as an SE(2) matrix,

$$\begin{pmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{pmatrix} \in SE(2)$$

The SE(2) pose representation has some nice properties, including being identical in form to a transformation matrix. This means to apply a translation and rotation to a robot pose, we can simply multiply two matrices together.

For our continuous-space localization algorithm, we use a particle filter. We don't know the initial robot pose, so we scatter particles uniformly across the coarse map. Since we don't know the scale of the coarse map, we cannot directly convert between pixels and meters; so, we choose a reasonable guess for the scale, such as 0.5 meters/pixel. It need not be fully accurate, as we will see.

$$P_{init} = \{p_1, \ldots, p_N\}$$
$$\text{for } \begin{pmatrix} x_i \\ y_i \end{pmatrix} \sim \mathcal{U}\left(\begin{pmatrix} x_{\min} \\ y_{\min} \end{pmatrix}, \begin{pmatrix} x_{\max} \\ y_{\max} \end{pmatrix} \right)$$
$$\text{and } \theta_i \sim \mathcal{U}(\mathbb{S}).$$

In the prediction update stage, we could use either the last commanded twist of velocities, or the odometry feedback from the robot itself. We use the latter, propagating each particle forwards by the motion the robot experienced since the last iteration.

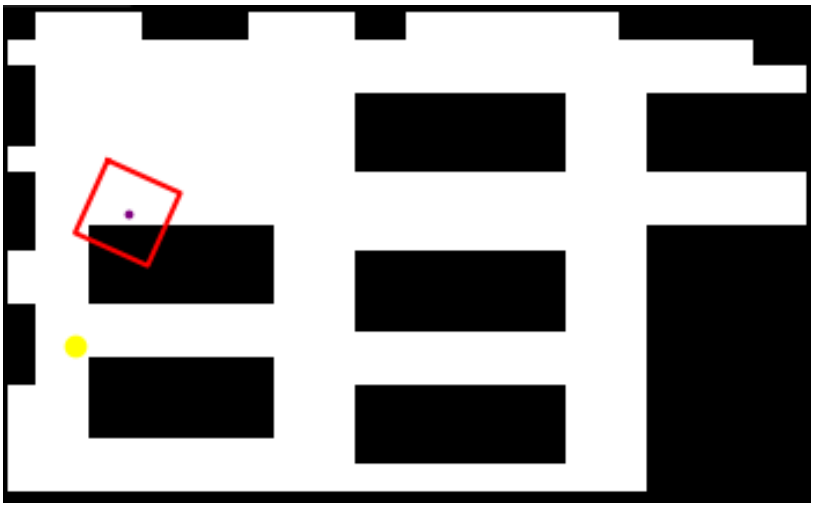
$$T = p_{odom,t-1}^{-1} \cdot p_{odom,t}$$

For each particle,

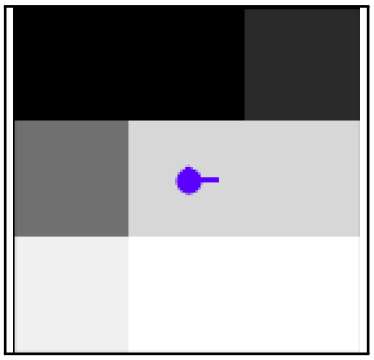
$$p_{i,t} = T \cdot p_{i,t-1}$$

In the measurement update stage, we receive a local occupancy map O_{meas}. For each particle, we extract a region surrounding it, O_i, and compare this to the local occupancy by taking the mean-squared-error (MSE). The lower the error between these, the more likely the particle is to represent the true robot pose. So, the likelihood L_i of particle p_i is

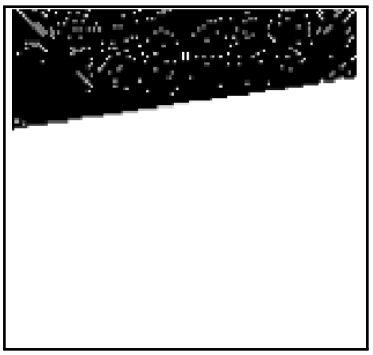
$$L_i = 1 - MSE(O_{meas}, O_i)$$



(a) Particle pose on the coarse map, with its local occupancy region outlined.



(b) Local occupancy extracted from the coarse map and rotated.



(c) Local occupancy measurement from LiDAR data.

Figure 3.14: The measurement update stage for a single particle. Its expected local occupancy is extracted from the coarse map, rotated to be relative to the robot facing east, scaled up to match the size of the measured local occupancy, and compared using MSE to compute this particle's likelihood.

Resampling is fairly standard, with particle weights being the normalized likelihoods computed from the measurement update stage. Particles are selected with replacement from the current particle set using these weights. A small noise matrix is sampled from $SE(2)$ for each and used to perturb particles, which avoids multiple particles being exact duplicates. Finally, a small

number of particles are sampled from the initial uniform distribution across the map to help prevent particle depletion.

Each iteration, the particle with the highest likelihood is considered the current filter estimate, and used for path planning. A more complex approach may be better for stable performance, but this is sufficient for this proof-of-concept.

3.3.2 Motion Planning

Path planning at the highest level is done the same way as in the discrete case, using A* from the nearest pixel to the current estimated robot pose to the goal cell on the coarse map. However, rather than converting this to one of the three discrete actions, we feed the path (converted from pixels to meters) and the current pose estimate through a pure pursuit controller to generate a twist command with linear and angular velocities for the robot.

The pure pursuit algorithm involves choosing a lookahead radius, and finding a point on the path which is this distance from the robot, as far along the path as possible. Since we ideally want a tight controller that will keep the robot close to the path, avoiding clipping walls as we pass them, we start with a small lookahead radius of $r = 0.2$ meters. Starting from the end of the path, we progress backwards until finding a point that has Euclidean distance to the robot smaller than r. If no point is found, we increase r by 25% and search again. This repeats until a point is found, growing the lookahead radius up to a maximum of 2 meters. So long as the map resolution does not exceed 2 meters/pixel, this guarantees a lookahead point will be found, since we always start the path from the pixel nearest to the current estimated robot pose.

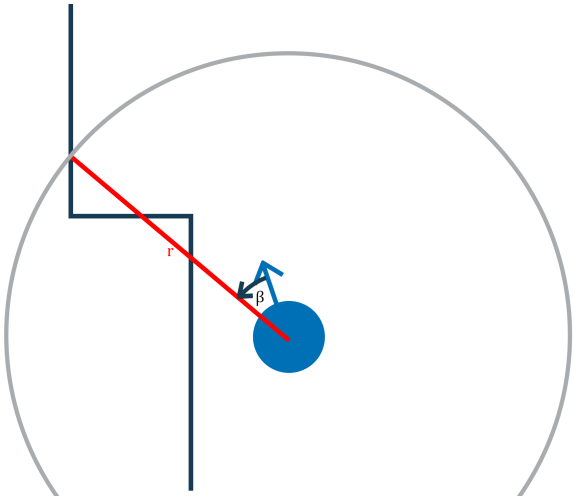


Figure 3.15: The relative angle β to the lookahead point for radius r, using the path and the robot pose estimate.

After a lookahead point p_{la} is chosen, we use it and the current robot pose estimate p_{cur} to compute the relative heading difference,

$$\beta = \arctan\left(\frac{y_{la} - y_{cur}}{x_{la} - x_{cur}}\right) - \theta_{cur}.$$

We then choose an angular velocity which is simply proportional to this relative heading,

$$\omega = k_p \cdot \beta.$$

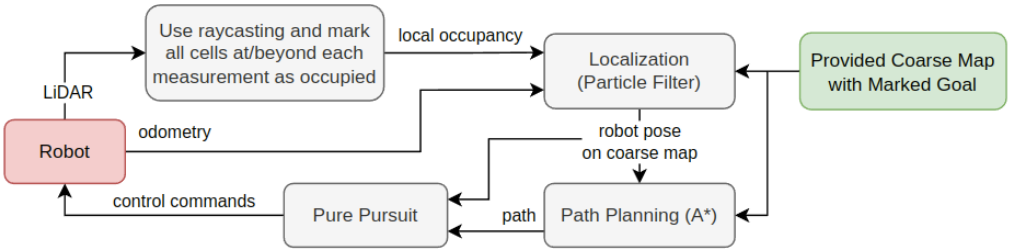
A PID could be used here to ensure ω does not change too rapidly between iterations, but for our platform this did not seem necessary. We use a coefficient of $k_p = 0.9$.

The linear velocity is computed such that it's larger when the robot is more aligned with its goal heading (i.e., ω is smaller).

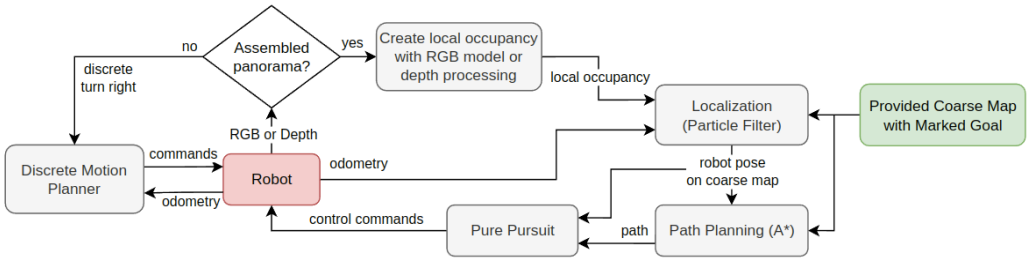
$$v = k_f \cdot (1 - |\beta/\pi|)^{k_e} + k_a$$

The exponential term causes attenuation in the linear velocity, so that v stays close to 0 until the robot is sufficiently aligned with its goal direction; we use $k_e = 5$. The additive term k_a serves as a lower bound on linear speed for a platform that can't perform zero-point-turns; our LoCoBot is capable of this, so we set $k_a = 0$. The forward multiplicative term helps to scale the result of the inner term to the feasible range of the physical platform, and was not needed for the LoCoBot, so $k_f = 0$ as well.

After these calculations, we have a twist of velocities (v, ω) to send to the robot, which will be applied for a configurable time period dt, after which a new measurement will be generated and the next iteration will proceed.



(a) Architecture for continuous CMN with LiDAR data.



(b) Architecture for continuous CMN with RGB or depth data.

Figure 3.16

Not shown on these architecture diagrams is that the LiDAR data is continuously processed into local occupancy in order to perform reactive obstacle detection. If an obstruction is present, control commands are replaced by zeros to halt the robot.

Chapter 4

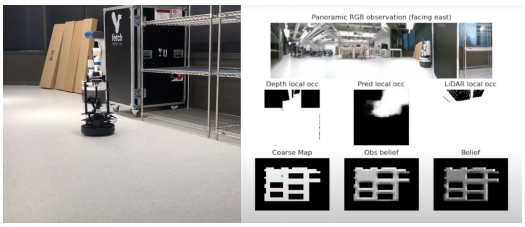
Conclusion

4.1 Discussion

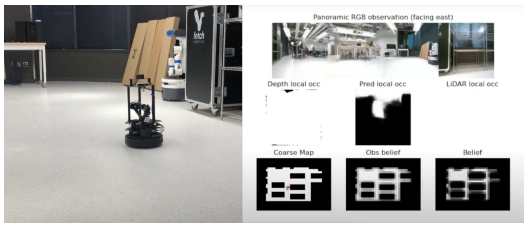
The goal of this project was to implement the Coarse Map Navigation pipeline on a physical robot and demonstrate its effectiveness, which we have been able to do with reasonable success. By changing the perception stage to the depth pointcloud-based method we've described, the robot is able to collect very good local occupancy measurements from the environment. Our extensions to the localization process and implementation of a motion controller allow the discrete state and action spaces to work well even in a physical environment. We found that CMN was able to succeed even with hand-drawn maps containing incorrect proportions and substantial scale differences from the local occupancy measurements.

Some downsides still limit the usability of CMN, however. Since the only perception method that works suitably well requires the robot to stop and pivot to gather a depth panorama, the process is painfully slow, and the platform must have the resources to power a RealSense RGB-D camera and process its data stream into a pointcloud. We had hoped to generalize to a more lightweight system by relying only on monocular RGB images, but we found them too poor for the algorithm to succeed. We have discussed potential avenues to improve the RGB method, which will be expanded upon in Section 4.2.

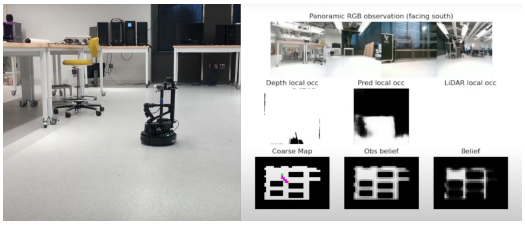
We have some videos of our CMN implementation running in simulation and on the robot, which are available at this link. Some notable points of a successful run on the robot are shown in the following figure. Our code is also available as open source on GitHub.



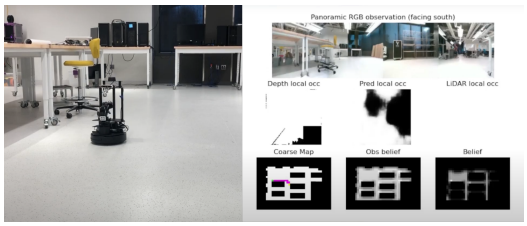
(a) Initially, the belief map is very spread out, so the robot moves randomly to explore its surroundings.



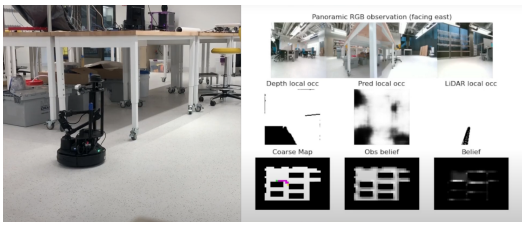
(b) The highest likelihood cell in the belief map is incorrect, so the planned path is nonsensical.



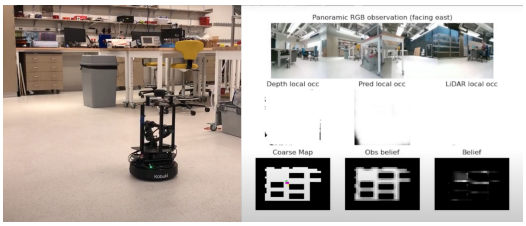
(c) The robot has observed empty surroundings after a couple of motions, so the belief map has mostly converged into the large free region of the map.



(d) The belief is now correctly focused alongside the table. This demonstrates the locally topological nature of our localization method, as the number of cells it plans to move is many more than will be necessary.



(e) As the robot first observes the end of the table when it enters the aisle, the belief map will change from a long spread alongside the table to be more focused in the true region.



(f) The localization is focused on the correct pose, and the robot approaches the goal to end the run.

Figure 4.1: Notable moments in a run of CMN on the physical robot. The full run video is available here.

4.2 Future Work

Our experimentation with the physical implementation took place entirely using the LoCoBot platform, and was done primarily in the same lab environment. To test for generalizability of our methods, more work should be done in different environments and with a greater variety of robotic platforms and sensors. We provide our codebase as open-source, and encourage others to implement this on their own robots.

A large undertaking that could be a project in it's own right would be to finish developing the pipeline to gather training data on the physical robot in a variety of environments, and use this to properly train the perception model. We believe its performance could be improved with this additional training, which would allow platforms that have only a monocular RGB camera to run coarse map navigation.

Similarly, it would be interesting to train a network to create the same local occupancy predictions from depth images, which could skip the slow computation stage of generating the pointclouds, only to then flatten them onto the plane.

This architecture could easily support the local occupancy region being in a different configuration; i.e., we have it centered on the robot, but this does not need to be the case. Another project could shift the local occupancy to the region in front of the robot, and use that for localization instead of the all-around grid we currently use. This would be great for speeding up the process on lighter platforms, since the measurements could be generated without the robot needing to stop and pivot in a circle to gather a panorama.

We developed the continuous domain as a proof-of-concept, but were not able to get the real-world performance to be as good as in the simulator, or as good as the discrete case. This is due in part to the methods we've used not operating well when the robot must completely stop moving each iteration to pivot and gather a panorama. Ideally, the measurements could be obtained continuously, while the robot is in motion, which would permit much smoother trajectories through the environment, and generally lead to many more iterations contributing to localization performance. Using the LiDAR for local occupancy generation helped on this front, but as we mentioned, it could not see tables or other obstacles outside its plane, so the localization was poor.

If CMN or a similar procedure were to be used frequently by anyone other than the primary developer(s), it would be very nice to have the hand-drawn map processing take place in a dedicated script with a GUI. This could allow for far more versatility, such as having a simple slider to tweak the granularity until it looks good, and allowing for thresholding adjustments and color inversion.

The processed map could then be saved in a tiny binary file format to be ported to a robot and used directly, rather than keeping the large, raw images and running the processing on every startup as we do currently.

Bibliography

[1] C. Xu, C. Amato, L.S. Wong, "Autonomous Robot Navigation using Coarse Maps," IEEE RA-L 23-0270.1 submission, 2023.

[2] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-Time Loop Closure in 2D LIDAR SLAM", in *Robotics and Automation (ICRA)*, 2016 IEEE International Conference on. IEEE, 2016. pp. 1271–1278.

[3] D. Fox, W. Burgard, F. Dellaert, S. Thrun, "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots", in *Proceedings of the National Conference on Artificial Intelligence*, Orlando, FL, USA, 18–22 July 1999.

[4] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer*, vol. 22, no. 6, pp. 46–57, 1989.

[5] B. Behzadian, P. Agarwal, W. Burgard, and G. D. Tipaldi, "Monte carlo localization in hand-drawn maps," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* . IEEE, 2015, pp. 4291–4296.

[6] D. Filliat and J.-A. Meyer, "Map-based navigation in mobile robots:: I. a review of localization strategies," *Cognitive systems research* , vol. 4, no. 4, pp. 243–282, 2003.

[7] S. Koenig and Reid G Simmons, "Passive distance learning for robot navigation," In *ICML*, 1996.

[8] S. Ito, F. Endres, M. Kuderer, G.D. Tipaldi, C. Stachniss, and W. Burgard, "W-RGB-D: Floor-plan-based indoor global localization using a depth camera and wifi," In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2014.

[9] K. Kawamura, A. B. Koku, D. M. Wilkes, R. A. Peters, and A. Sekmen, "Toward egocentric navigation," *International Journal of Robotics and Automation*, 17, 2002

[10] V. Setalaphruk, A. Ueno, I. Kume, Y. Kono, and M. Kidode, "Robot navigation in corridor environments using a sketch floor map," In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, 2003.

[11] D. Shah, J. Schneider, and M. Campbell, "A robust sketch interface for natural robot control," In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

[12] M. Skubic, S. Blisard, A. Carle, and P. Matsakis, "Hand-drawn maps for robot navigation," In *AAAI Spring Symposium, Sketch Understanding Session*, page 23, 2002.

[13] J. Yun and J. Miura, "A quantitative measure for the navigability of a mobile robot using rough maps," In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008.

[14] F. Boniardi, B. Behzadian, W. Burgard, and G. D. Tipaldi, "Robot navigation in hand-drawn sketched maps," in *2015 European conference on mobile robots (ECMR)* . IEEE, 2015, pp. 1–6.

[15] K. Matsuo and J. Miura, "Outdoor visual localization with a hand- drawn line drawing map using fastslam with pso-based mapping," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* . IEEE, 2012, pp. 202–207