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

Kevin M. Robb

M.S. Robotics Thesis Defense

Problem

A mobile robot needs to understand its relationship with the environment to do anything meaningful.

- "Go to X room"
- "Find Y feature"
- "Fetch Z object"

} "Go to a location"

Requires knowing:

- Where the robot is
- Where the task/goal is
- How to get there

We assume initial pose is unknown, i.e., the "kidnapped robot problem".

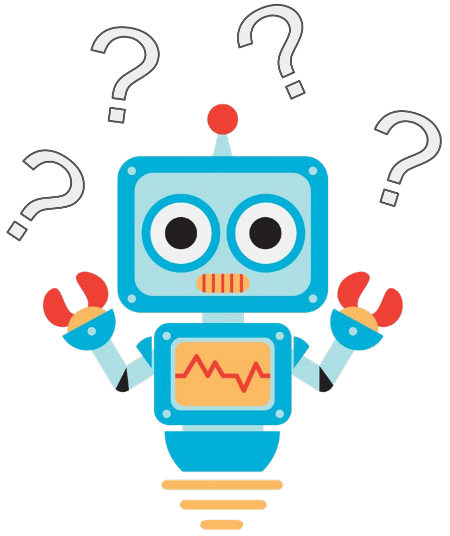
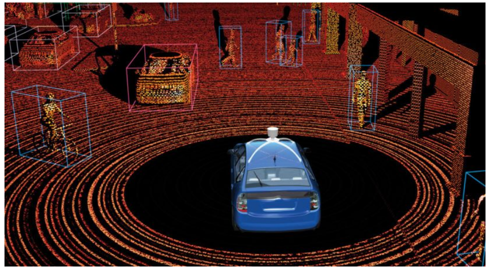


Image from wikimedia: link

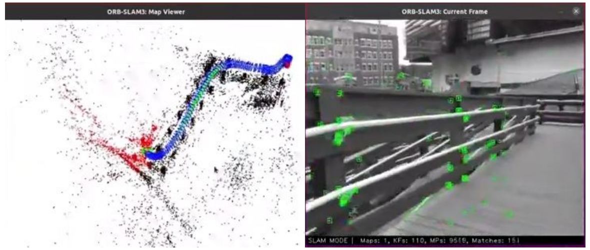
Common Solutions

Main two approaches used:

1. Online SLAM
   a. Requires good sensors, compute, power, and weight.
   b. Needed for huge or rapidly changing environment.
2. Map environment in advance, then localize in that map when performing the task
   a. Lower resource requirements.
   b. Map can be used many times.
   c. Only for reasonably small and static environment.

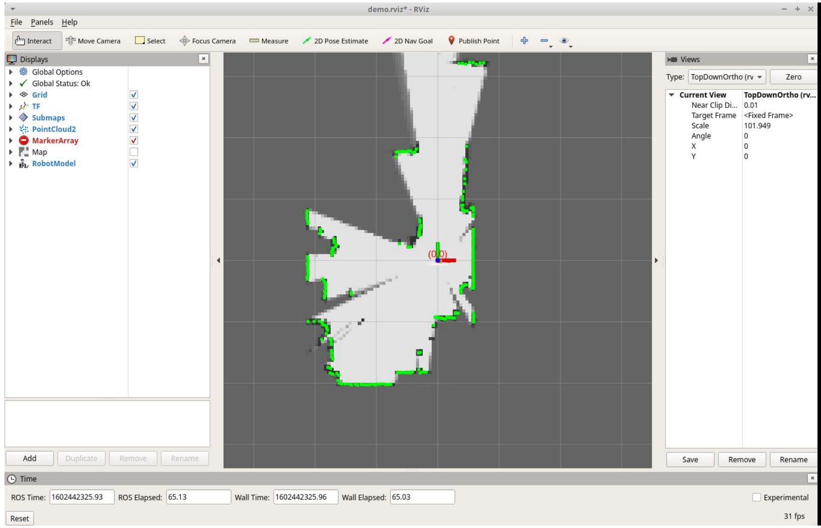


Autonomous cars can run online 3D LiDAR SLAM.
(Image from Small World Social: link.)



Offline 3D trajectory reconstruction from monocular RGB images using ORB_SLAM3.
(Images from my own prior implementation: link.)

Closest example

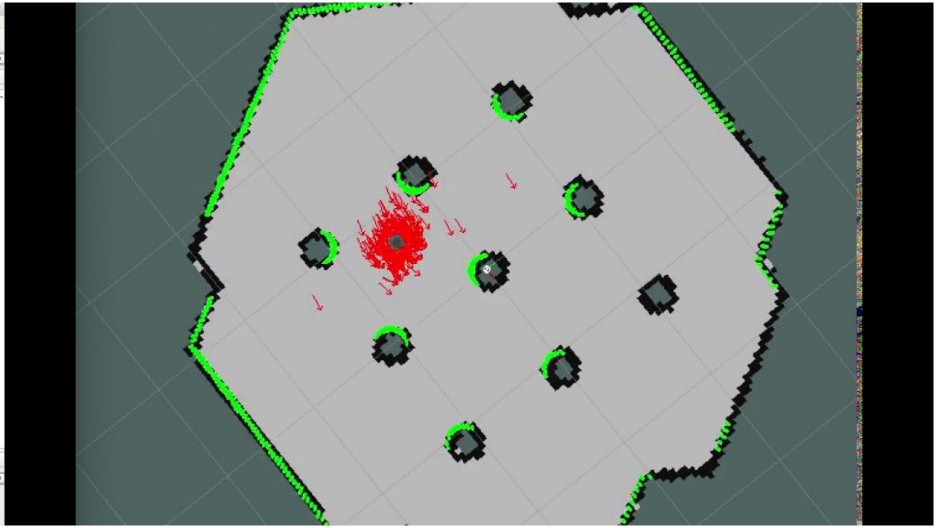


Type: TopDownOrtho (rv

Current View	TopDownOrtho (rv...
Near Clip Di...	0.01
Target Frame	<Fixed Frame>
Scale	101.949
Angle	0
X	0
Y	0

ROS Time: 1602442325.93 ROS Elapsed: 65.13 Wall Time: 1602442325.96 Wall Elapsed: 65.03 Experimental

31 fps



A turtlebot with a 2D planar LiDAR can run Cartographer-ROS to create an occupancy grid map, and then localize in it later with AMCL.

(Images from Robotics Weekends: link and Simon Bogh: link)

Remaining Problem

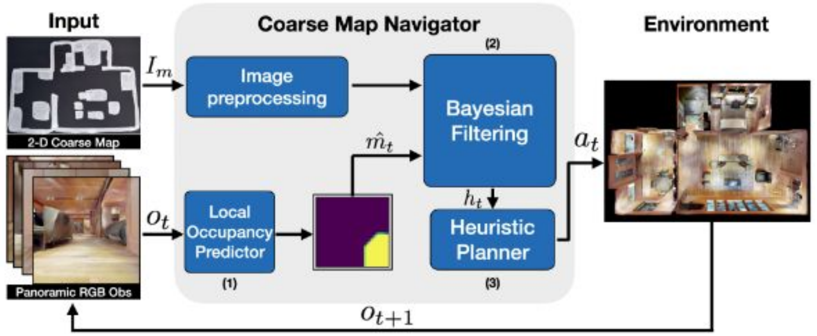
Robot itself must collect data and generate a high-resolution map of the environment, either online or in advance.

What if we have:

- a platform incapable of online SLAM
- **and** can't generate a map in advance?
    - Mapping is too time-consuming
    - Can't access the environment beforehand
    - Environment changes often, so new map is needed every time

Proposed Solution: CMN

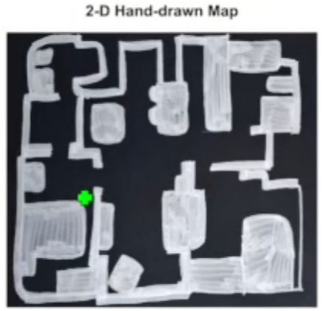
The Coarse Map Navigation (CMN) algorithm proposed by Xu [1] solves this problem.



Figures from Xu [1].

Someone quickly creates a coarse map, characterized by:

- Topologically accurate on a high-level
- Very low resolution
- Possibly wrong proportions
- Unspecified scale



This map can then be used for localization and navigation.

Xu was able to show CMN works well in the Habitat simulator.

[1] Chengguang Xu, Christopher Amato, Lawson L.S. Wong, *Autonomous Robot Navigation using Coarse Maps*, IEEE RA-L 23-0270.1 submission, 2023.

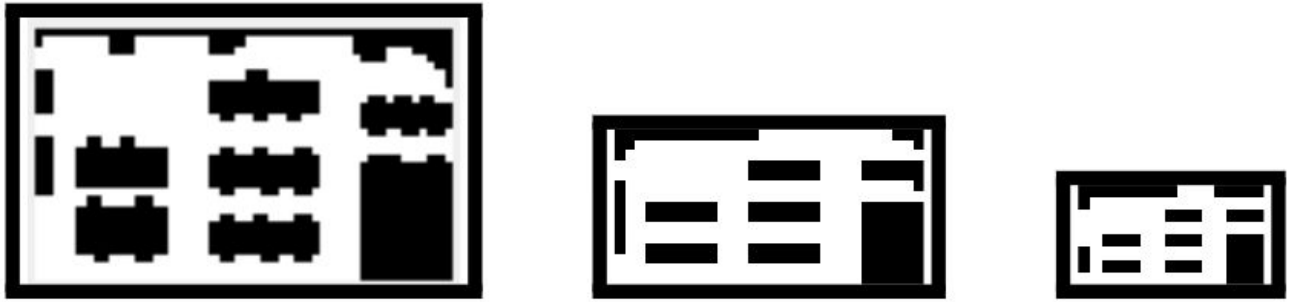
Getting a Coarse Map

We can use two methods of generating coarse maps. The second is much faster/easier.

1. Draw a map on paper and process it down.
2. Draw a map in GIMP with the pencil tool directly in low-res.

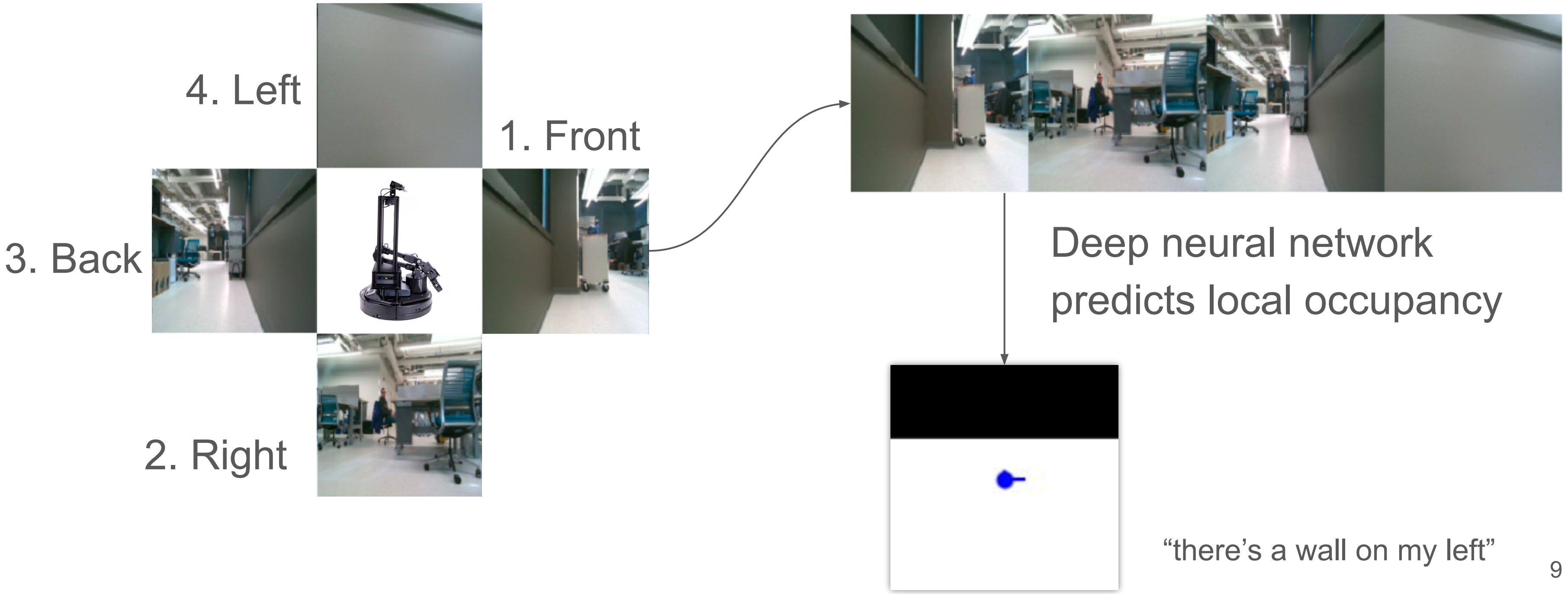
Coarse Map Variability

The same environment should be navigable with a wide range of coarse map resolutions.



CMN Components: Perception

Robot takes RGB image on all four sides to create panorama.

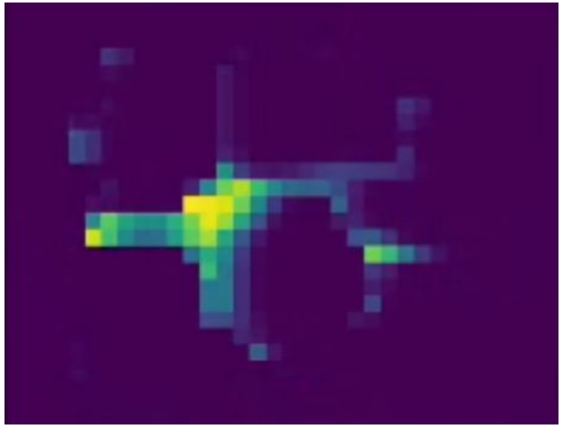


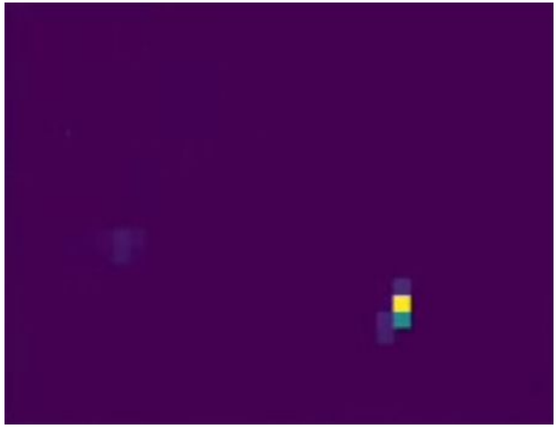
4. Left
1. Front
3. Back
2. Right
Deep neural network predicts local occupancy
"there's a wall on my left"

CMN Components: DBF Localization

A discrete Bayesian filter (DBF) runs directly on the coarse map to estimate robot position as a cell.

Orientation is assumed known, and is one of North, East, South, West.


Belief on the coarse map


Convergence

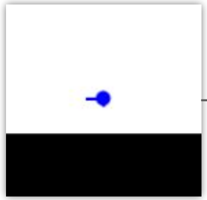
Figures from Xu's CMN

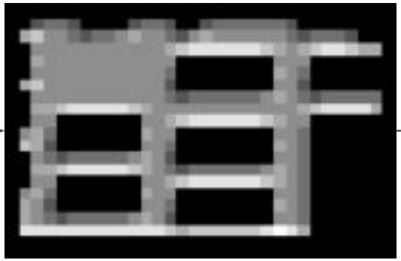
DBF: Measurement Update and Scale Invariance

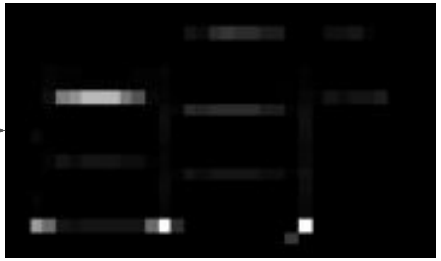
Local occupancy has known size of 1.28 x 1.28 meters.

For measurement likelihood, this is matched to 3x3 regions on the coarse map. Implicitly, this assumes the coarse map is at least ~0.43 m/px.

This is fine though, due to topological accuracy of coarse map.

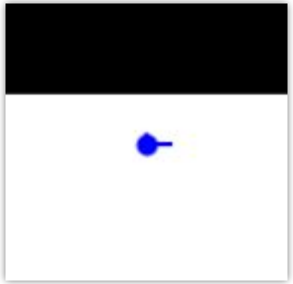

Local occupancy measurement.

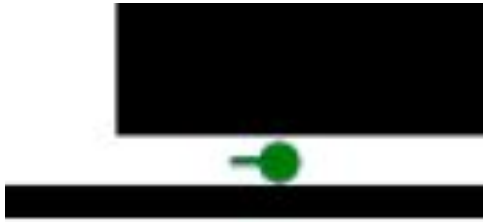

Measurement likelihood map. Robot is known to be facing West.


Belief map after many iterations.

DBF: Failure Cases

1. Too low resolution → messes up local topology
   a. Wide hallways become 1 pixel wide.
   b. Doorways close off, making path planning impossible.
2. Too high resolution → delays convergence
   a. Causes many equally likely cells, since only adjacent pixels are checked.


From local occupancy, robot sees a wall on only one side


In coarse map, robot expects a wall on both sides, since the corridor is only 1 pixel wide

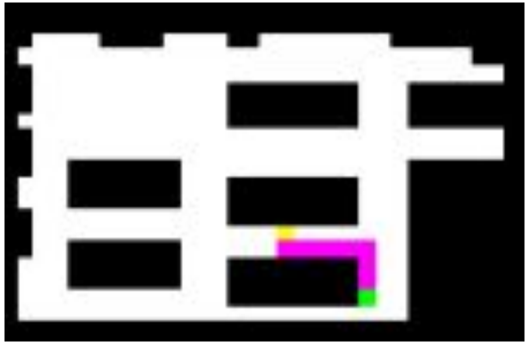
CMN Components: Planning

Plan a path to the goal with graph search (e.g., A*).

Choose a discrete action:

- Move forward (set distance, e.g., 0.15 m)
- Turn left 90 degrees
- Turn right 90 degrees

Action is commanded to robot, and reported to DBF for predictive update.


Planned path from robot estimate (green) to goal (yellow)

DBF: Predictive Update

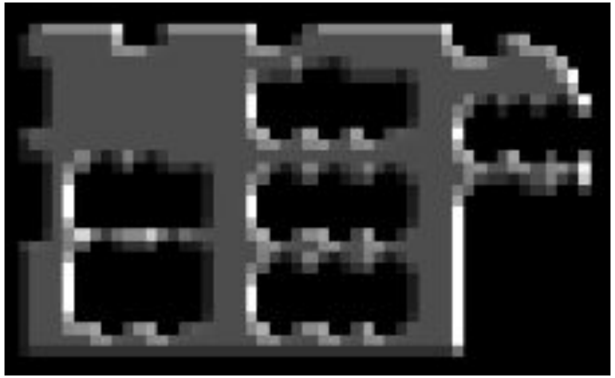
Orientation is known ⇒ only need to update for “move forward” actions.

Propagate entire map by 1 cell in direction of motion, accounting for walls.

Problems:

- Scale is unknown: Robot could move in real world, and still be inside same pixel.
- Proportions could be wrong: Walls can seem shorter/longer.

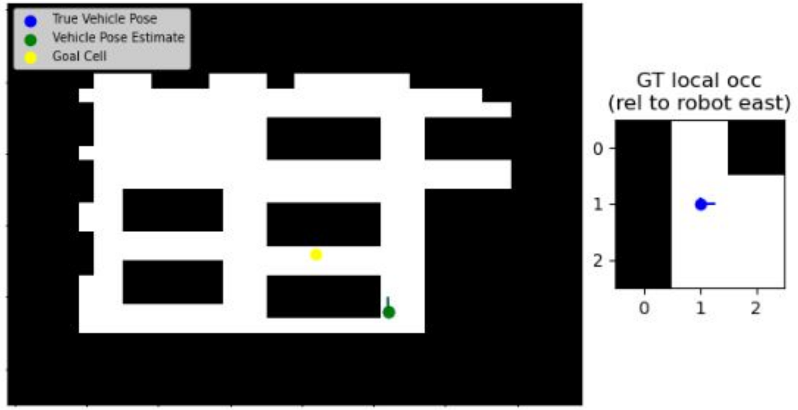
So, combine “move” (propagated) map with “stay” (current) map.

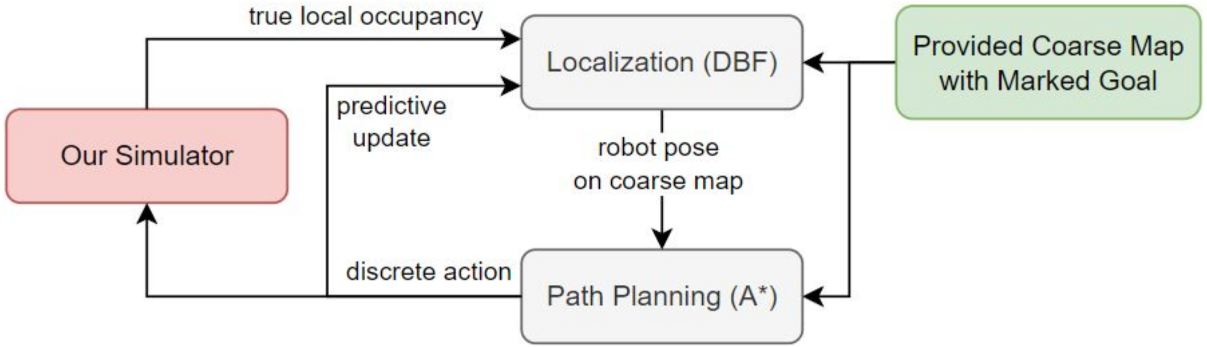


Belief map as the robot has been moving North with a wall on its right.

Our Implementation

- Simple simulator to test algorithm without perception step
- Localization using same method as original CMN, expanded for yaw
- Path planner basically the same





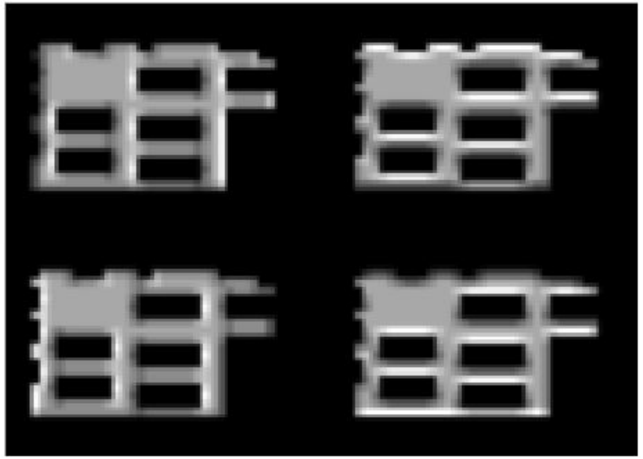
Extending DBF to Localize Orientation

In the original CMN, yaw was assumed known and not estimated.

We can easily expand the DBF to four layers (one for each possible orientation) to estimate position and yaw simultaneously.

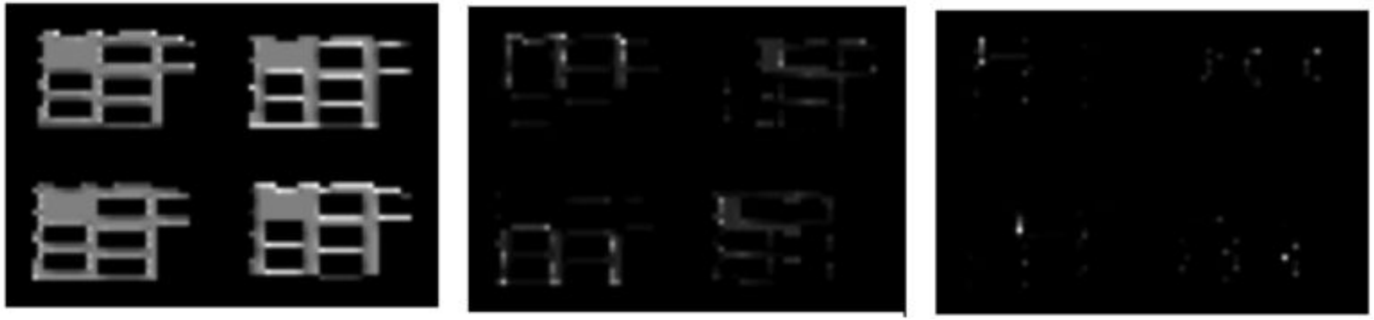
Measurement belief map is four layers, one for each rotation of the local occupancy.

Predictive update applies independently to each layer, with the motion direction dependent on the yaw corresponding to the layer. For rotations, the layers are cycled.


Measurement belief map for a single local occupancy evaluation

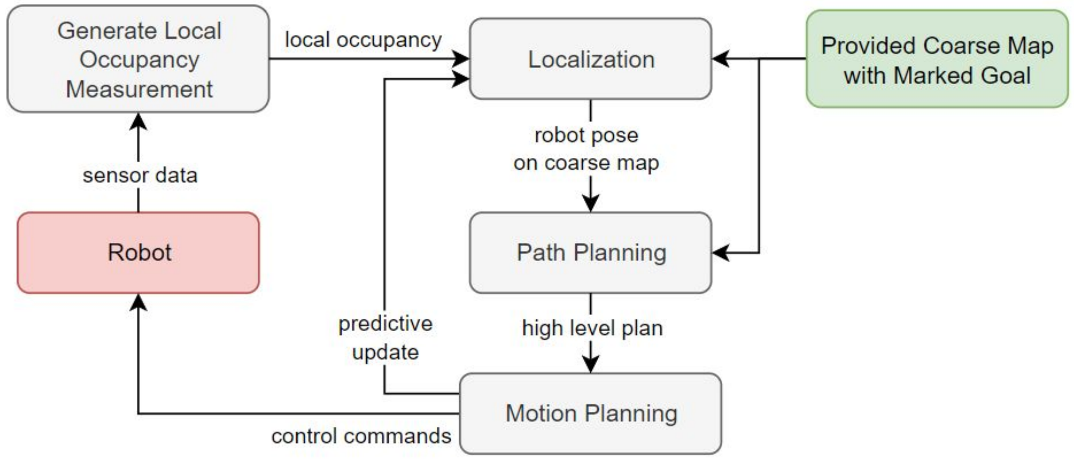
Extending DBF to Localize Orientation

This makes localization take more iterations on average, but works just as well unless the environment is highly rotationally symmetric.



Belief map in the 3D DBF after 1, 10, and 20 iterations.
Belief is normalized across all layers, not within each layer; so, there is still an overarching position estimate, and the layer it's on gives the yaw estimate.

Moving to the Physical Robot



Trossen LoCoBot, the robot we use for this project. It has a Kobuki base, RealSense RGB-D camera, 2D planar LiDAR, and a manipulator arm (unused).

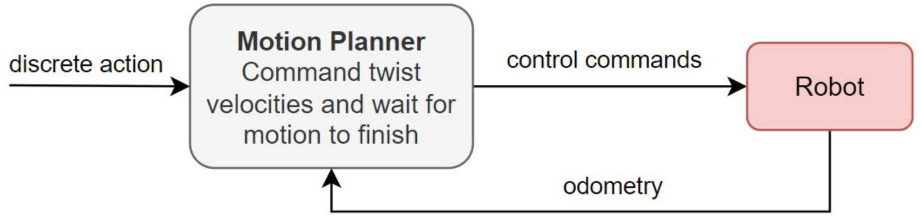
Challenges:

- Maintaining discrete state space
- Performing discrete actions
- Avoiding collisions
- Generating local occupancy measurements

Motion Planner to Keep Discrete State/Action Space

To keep the current CMN algorithm, the control layer must:

1. Command discrete actions to robot
2. Ensure the robot remains axis-aligned after each action



Motion Planner: Naive Pivot Controller

1. Record current yaw θ from odometry.
2. Command max angular speed.
3. Wait for odom to reflect that yaw is now $\theta+\pi/2$.
4. Command zero to halt.

This approach is terrible though, mainly due to two sources of drift:

- Overturning/slip due to sudden change in commanded velocity
- Drift in robot's internal odometry itself over time due to slip/drag/etc

Motion Planner: Better Pivot Controller

Minimize slip:

- Ramp speed up from 0 at set accel for first half, then back down to 0 by end.

Prevent drift accumulation:

- Realign to robot’s internal global odom every time.
- i.e., instead of turning from θ to $\theta+\pi/2$, compute angle to exactly reach desired cardinal direction.

Now the only drift is from internal kobuki base odom, which is fine for a long time.

Motion Planner: Forward Motion Controller

Since scale is unknown (and already handled), exact amount moved doesn’t matter.

Main goal is to prevent skewing off orientation.

So, speed ramps up and down similarly.

With this controller, we’re able to run the robot for a long time before the buildup of drift makes localization fail.

Generating Local Occupancy Measurements

In the base CMN work, the simulated robot could get an RGB panorama at any time, which was passed into a deep perception model trained to yield local occupancy predictions.

Figure from Xu's CMN in Habitat

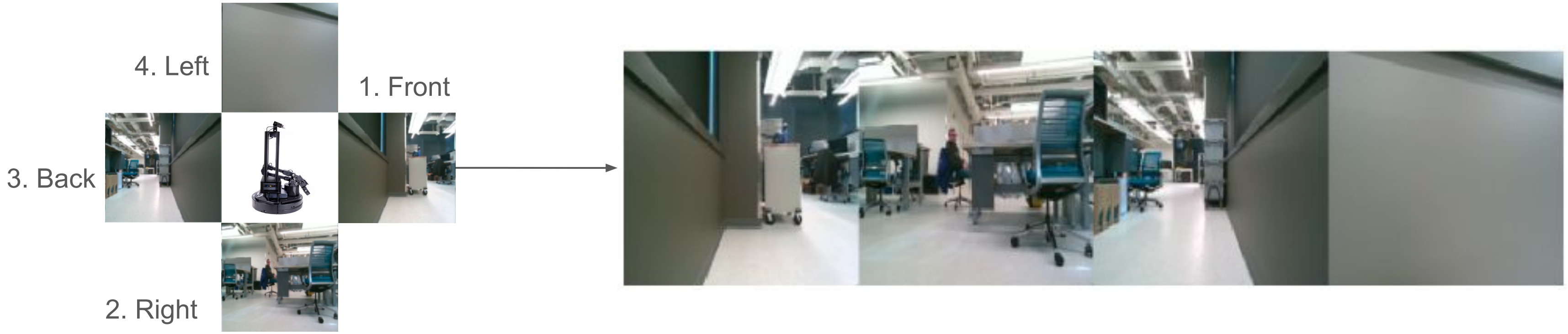
This presents two challenges for us:

1. Collecting a panorama each iteration
2. Domain shift between Habitat simulator and our real-world environment

Panorama Collection

Four cameras would be nice, but four USB-3.0 ports and RealSenses is impractical.

So, do a 90 degree pivot four times and concatenate the images into a panorama whenever one is requested.

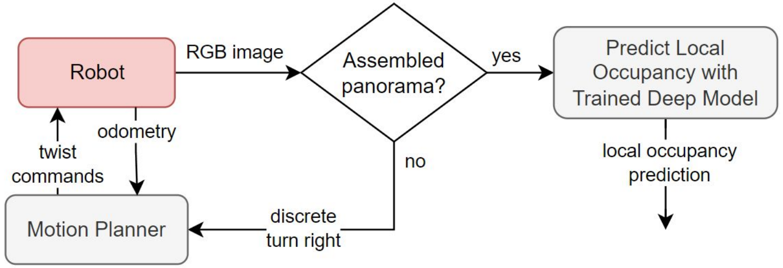


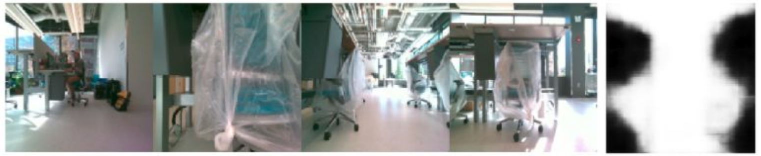
This is why minimizing drift is so important, as we will be pivoting 4 or 5 times every iteration. (It also makes CMN very slow.)

Predicting local occupancy

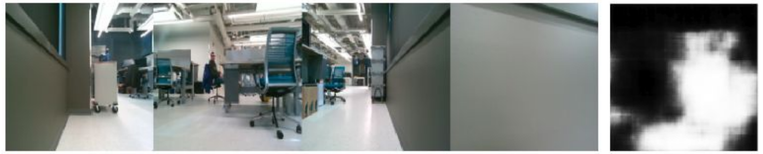
Panoramas are input to the same model as original CMN used. Results range from good to unusable.

As we feared, there is a substantial difference in the Habitat training environment and the lab in EXP we've been using.





Fairly good prediction, where the robot is between two rows of desks w/ chairs.



Prediction that captures the wall, but also contains large areas of false obstacle detection.

Improving local occupancy measurements

Some options to get better local occupancy measurements:

- Move to an environment closer to the trained domain. (the easy way out)
- Gather training data in the real world to improve the model.
    - This could be done by generating a high fidelity map (e.g., using Cartographer-ROS), then gathering pairs of RGB panorama + robot pose. Using the pose, extract the ground-truth local occupancy region from the map.
- Train in more Habitat environments to increase the domain.
- Use a different sensor to generate local occupancy without using the deep perception model.

Since this project is focused on implementation of CMN overall, we chose to explore other sensors, rather than diving into model improvement.

Alternate perception methods

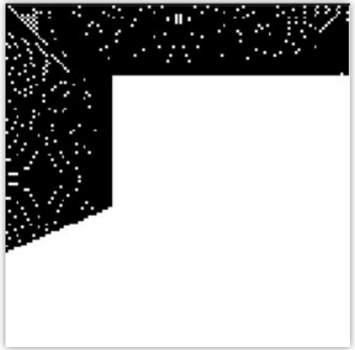
We have some other sensors available to use, each of which we will explore:

- 2D planar LiDAR → 360 degree FOV → instant panorama 😍
- RealSense D455 → ~90 degree FOV → still have to pivot for panorama

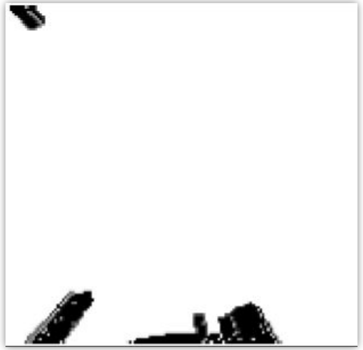
LiDAR local occupancy

Generating local occupancy from LiDAR is easy!

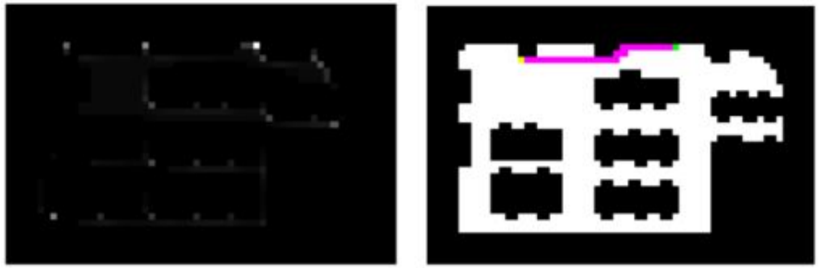
We get a LaserScan of hit distances at each angle; by marking everything beyond them as occupied, we can get a local occupancy around once every 0.1 seconds.



Local occupancy is excellent when the robot is near large flat walls



The LiDAR can only see the legs of tables, so underrepresents the environment compared to the coarse map



CMN works extremely well when the robot remains near walls that the LiDAR can easily detect

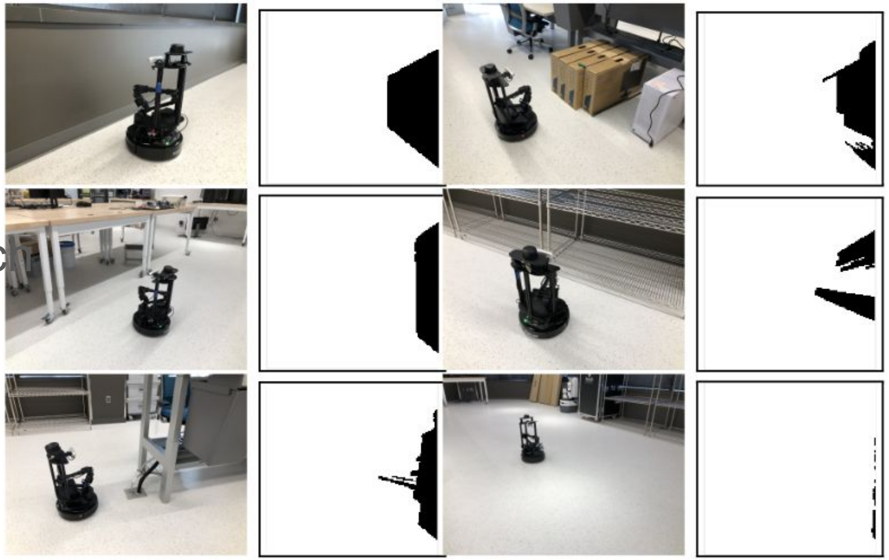
Depth local occupancy

The RealSense has a vertical FOV, so it can see short obstacles and tall tables!

Process depth image → pointcloud, then project all points down to the plane, and mark all cells behind them as occupied.

This adds a lot of computation time, so each measurement takes ~0.2 seconds, and we must pivot to gather four of them.

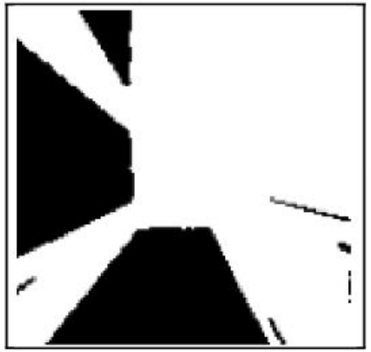
Despite being slow, this works great and solves the LiDAR’s problem.

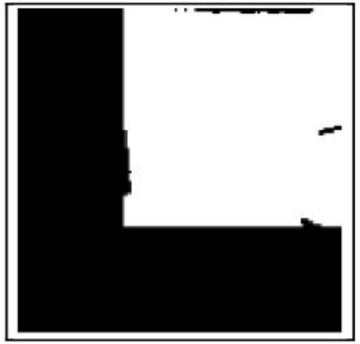


Depth local occupancy

Horizontal FOV is < 90 degrees, so this is insufficient for full coverage.

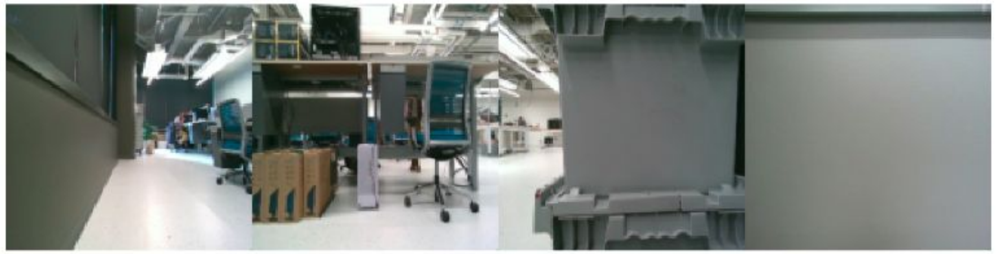
Could take more measurements at smaller increments, but that would make the process even slower. So, if the corner regions are mostly occupied, just stamp them as fully occupied.

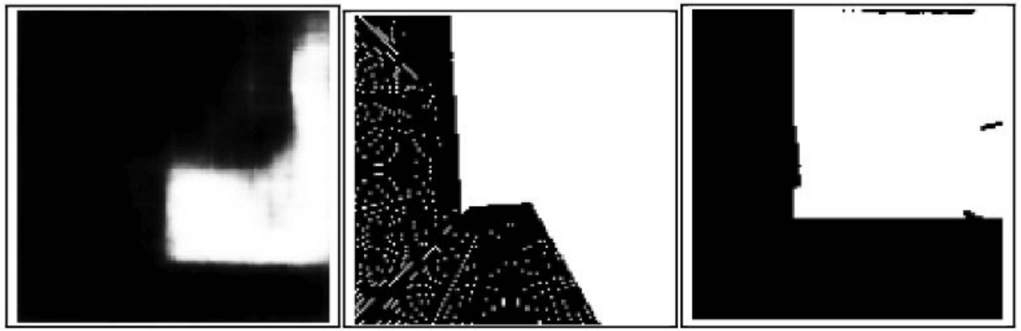




This corner filling method makes CMN work great on the robot in our lab environment!

Method comparison → Robot in a corner


(a) RGB panorama


(b) Predicted from RGB

(c) LiDAR

(d) Depth pointcloud

(Large false positive region)

(Both perfectly usable)

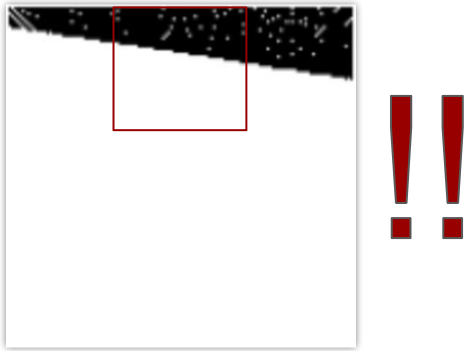
Avoiding collisions

If the robot runs into something, it can easily become skewed off from axis-aligned orientation, and all is lost for our discrete representation.

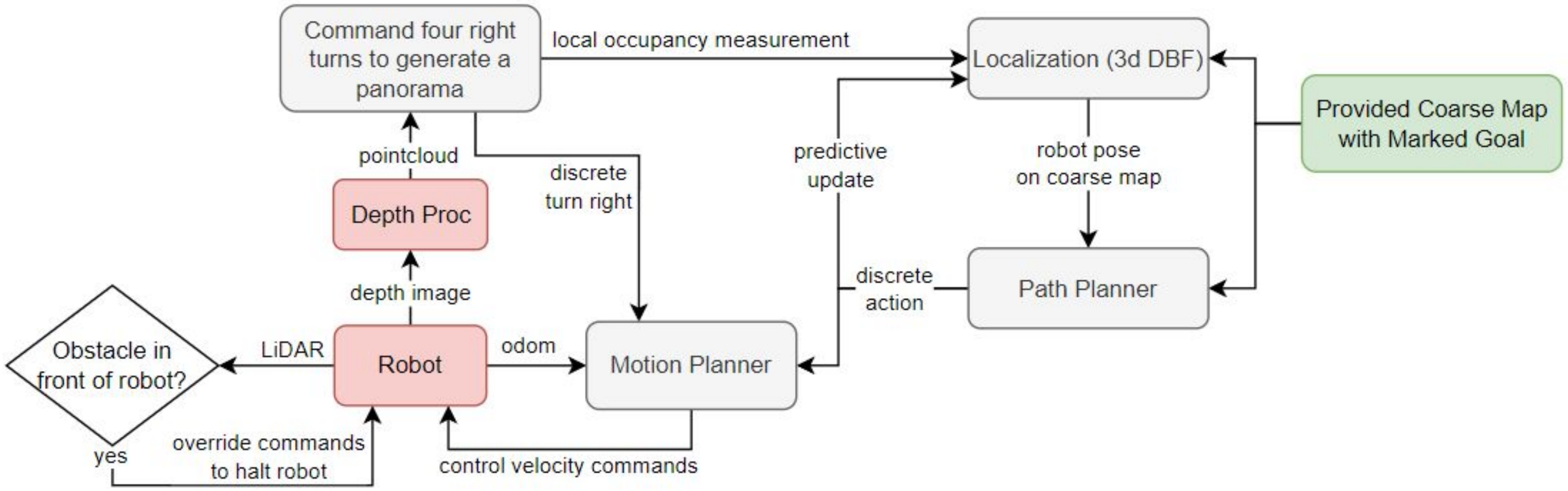
So, if the front region in the local occupancy is more than 25% occupied, the robot will halt and is prevented from taking the “move forward” action.

Check is applied in depth data when actions are selected. But, cannot use it during motion due to slow update.

So, always generate LiDAR local occupancy just for high frequency reactive obstacle detection.

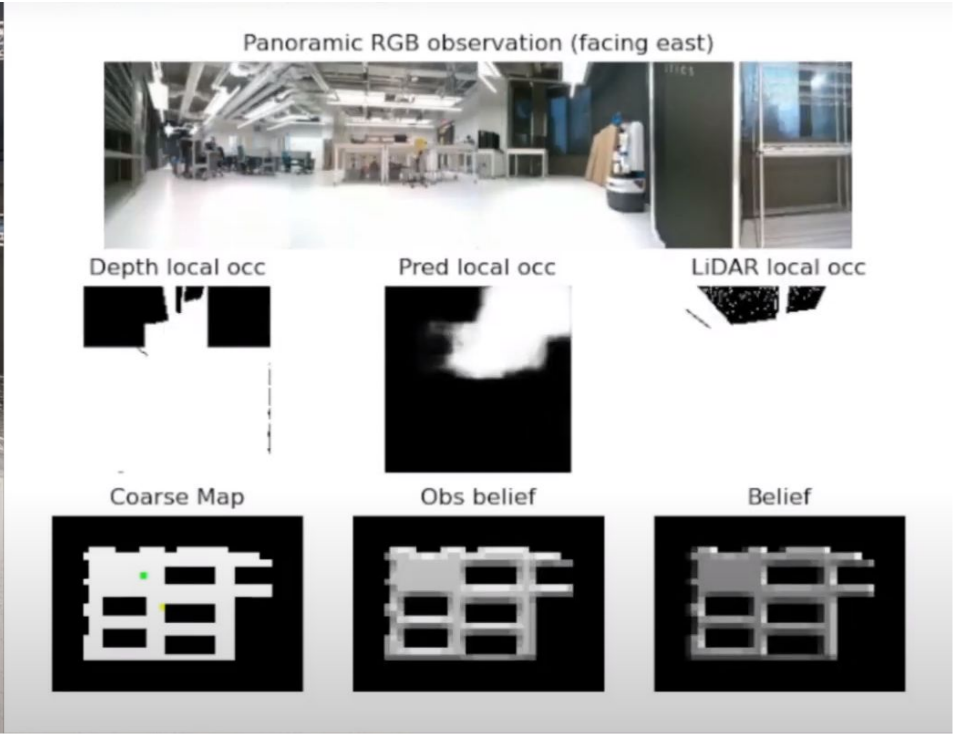


Architecture for Discrete CMN


Command four right turns to generate a panorama
local occupancy measurement
Localization (3d DBF)
Provided Coarse Map with Marked Goal
pointcloud
discrete turn right
predictive update
robot pose on coarse map
Depth Proc
discrete action
Path Planner
depth image
Obstacle in front of robot?
LiDAR
Robot
odom
Motion Planner
yes
override commands to halt robot
control velocity commands

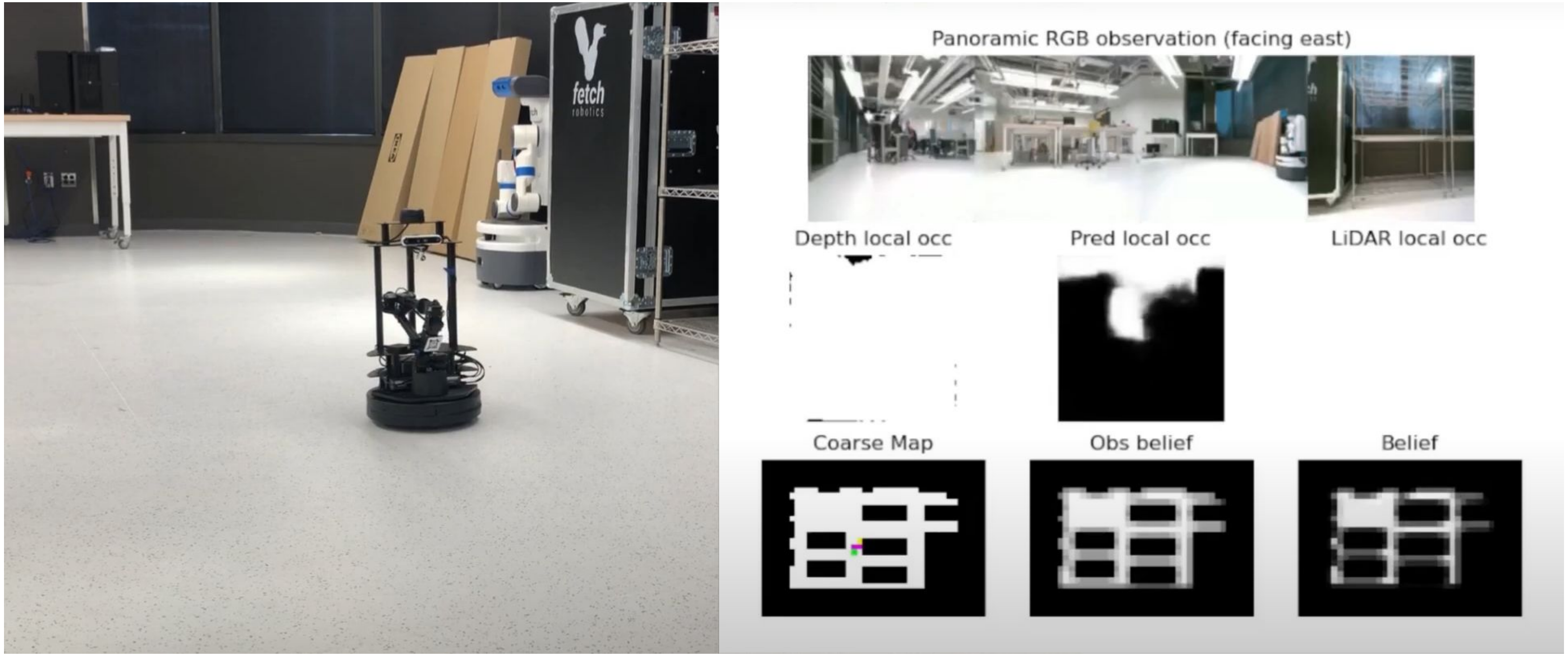
Discrete CMN Results (video)

Notable parts of the video (a)



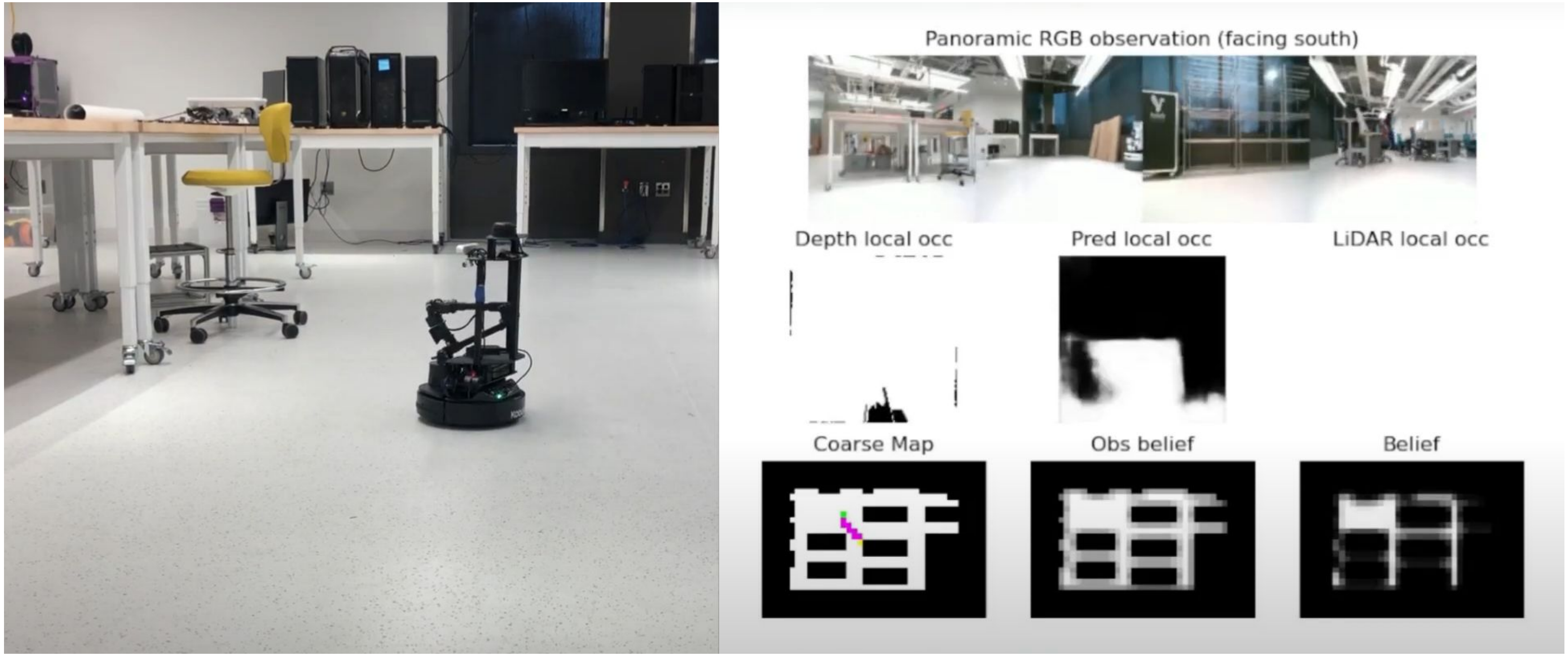
Initially, the belief map is very spread out, so the robot moves randomly to explore its surroundings.

Notable parts of the video (b)



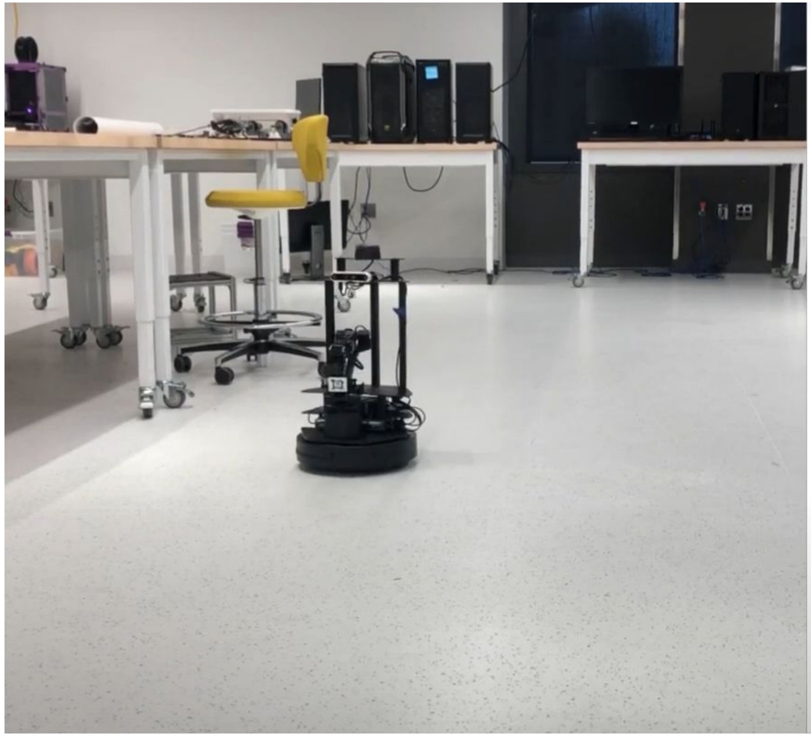
The highest likelihood cell in the belief map is incorrect, so the planned path is nonsensical.

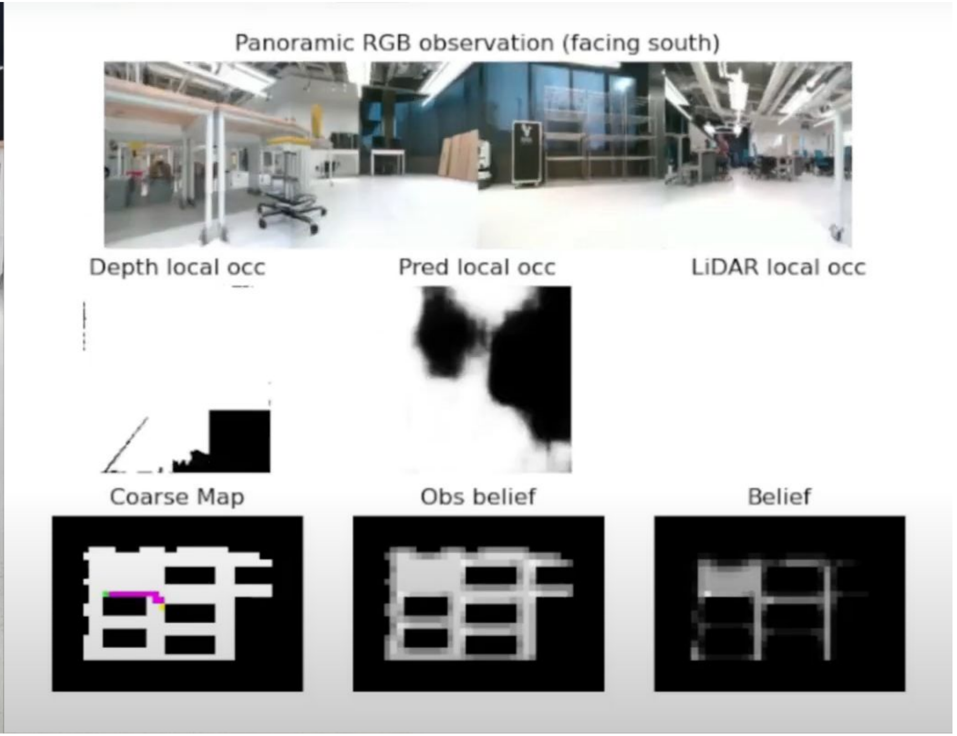
Notable parts of the video (c)



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.

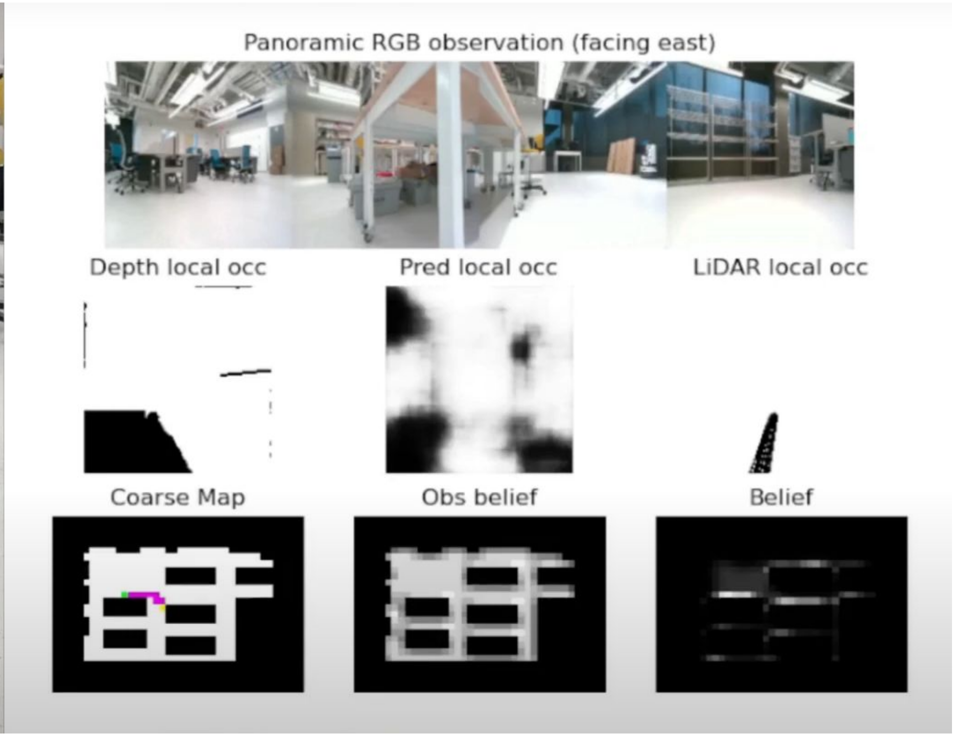
Notable parts of the video (d)





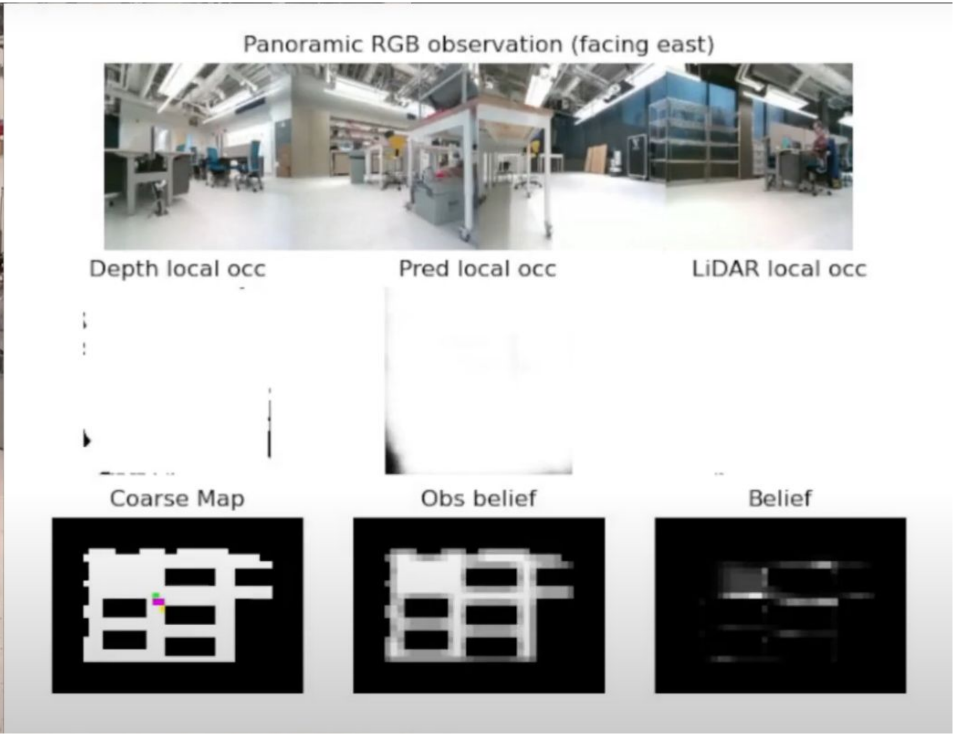
The belief is now correctly focused alongside the table. This demonstrates the locally topological nature of our localization method, as the number of cells it plans to move is many more than will be necessary.

Notable parts of the video (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.

Notable parts of the video (f)



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

Extension of CMN to 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 to continuous allows noise to be included in the filter itself, instead of abstracted away.

This also increases the generalizability to more platforms and environments, and can be used on non-differential-drive systems that can’t pivot in-place.

Local occupancy measurements will be generated the same as in discrete CMN, with the robot stopping to pivot and collect a panorama each iteration.

Particle filter localization

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

In the continuous domain, the robot pose is $p = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2, \quad \theta \in \mathbb{S} \right\}$

which can be represented as a matrix, $\begin{pmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{pmatrix} \in SE(2)$

Particle filter localization

Still don’t know coarse map scale, yet now we’ll work in meters instead of pixels. How?

Simply assume the correspondence is known, as say 0.5 m/px. Any inaccuracy will be handled by the filter as noise.

- This sort of thing is already done in Bayesian filters all the time, such as using an oversimplified constant-velocity motion model.

Now we can convert a pose in meters to a cell on the map, and CMN can operate continuously.

Particle filter localization

Initialize particles uniformly across the free space of the coarse map.

$$P_{init} = \{p_1, \ldots, p_N\}$$

$$\text{for } \begin{pmatrix} x_i \\ y_i \end{pmatrix} \sim \mathcal{U}\left(\begin{pmatrix} x_{\min} \\ y_{\min} \end{pmatrix}, \begin{pmatrix} x_{\max} \\ y_{\max} \end{pmatrix}\right)$$

$$\text{and } \theta_i \sim \mathcal{U}(\mathbb{S}).$$

The propagation stage simply records the change in odometry since the last iteration, and applies it as a transformation to each particle.

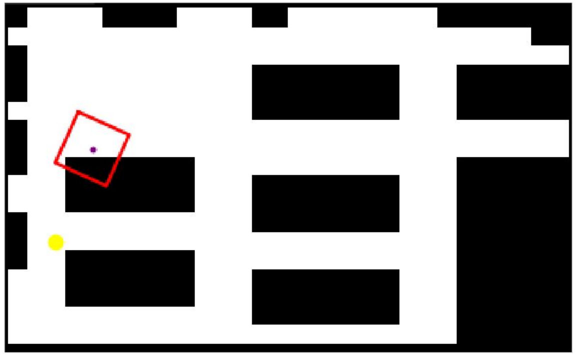
$$T = p_{odom,t-1}^{-1} \cdot p_{odom,t}$$

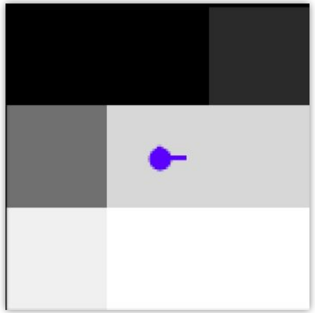
$$p_{i,t} = T \cdot p_{i,t-1}$$

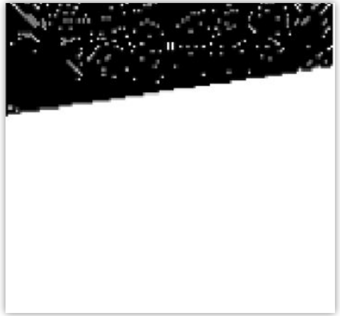
Particle filter localization

In the measurement update stage, we get a local occupancy map O_{meas}

For each particle, extract the region surrounding it, rotate to align with the measurement's expected orientation, and take the mean-squared-error. The particle likelihood is the inverted error. $L_i = 1 - MSE(O_{meas}, O_i)$


A particle to evaluate (facing WSW), with its local occupancy region that will be extracted.


Local occupancy extracted from the coarse map around one particle and rotated relative to the particle's yaw.

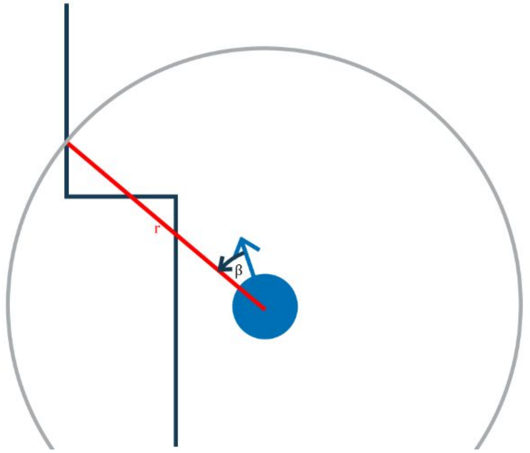

Local occupancy measurement from LiDAR, which is compared to all particles.

Continuous motion planning

High level path planning is basically the same, use A* from robot's nearest cell to goal.

But, don't choose a discrete action.

Compute the relative heading *β* to a point on the path a short distance ahead of the robot. Based on this angle, we compute an angular and linear velocity to command to the robot.



Basic idea of the pure pursuit algorithm, which looks ahead on the path to smoothly rejoin it.

For our case, the path always begins at the robot, so this is simplified and just provides a framework for smooth speed control and helps avoid overshooting turns in the path.

Continuous CMN discussion

It works fine in our simulator that skips the perception stage, but struggles on the physical robot. The continuous pipeline really shines when the robot can keep moving and follow trajectories spanning multiple timesteps.

When the robot has to stop every iteration to very slowly pivot for the next measurement, there is no momentum, and motions are just not high enough magnitude to cause big changes between frames.

So, this has been a good proof-of-concept, but is not practical with this measurement source.

Discussion and future work

We were able to demonstrate that CMN can work in a discrete context, even on a physical robot in the real world.

However, running it is very slow due to all the panorama gathering, which makes continuous CMN infeasible, and overall makes the algorithm annoying to use.

A future project could extend CMN to be far more realistic as a navigation pipeline by exploring a few areas:

- Use only front-facing data for localization
- Retrain the original CMN’s perception model so RGB data can be used
- Try more topological discrete actions like “move until wall”
- Implement on more robotic platforms and test in more environments

Acknowledgements

Thanks to my advisor Lawson Wong for his guidance in this project.

Thanks to Chengguang Xu for his help understanding and implementing the work from his original CMN paper.

And thanks to Northeastern University for the robot and lab space I used.

The End

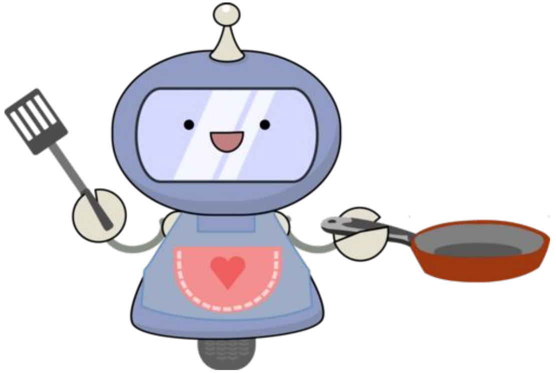


Image from Nohat: link.

The code is open-source, and there are some CMN videos (including the one from this presentation) online:

- github.com/kevin-robb/coarse-map-nav-integration
- www.youtube.com/@KevinRobbDesigns/playlists