Map Making

- Where do maps come from?
- Some are hand crafted by people. Very accurate, but work intensive. Robot limited to navigating in areas already mapped.

How do we represent the map?

- Divide world into a grid, just like graph paper.
- Mark some cells as occupied, and some as unoccupied.
- This is the map.
- Called occupancy grid. Figure 1 shows an example.
- Cells are called voxels
- If we need to we can idealize blobs into polygons for those path planning algorithms. More on that later.

Question: If a sensor tells us that some particular cell is occupied (or not occupied), what should we do?

- Mark the cell as occupied (or unoccupied)?
- What if the sensor is wrong?
- How do we incorporate this uncertainty into our model?
- Probability

The way we do this is just good old fashioned Bayes’ Rule:

\[ P(m_i|o) = \alpha P(o|m_i)P(m_i) \]

The problem with this is that our sensor readings are relative to the robot, whereas our map voxels are relative to some global reference frame. We can easily convert between the two, but only if the know where we are, but that requires localization, like particle filters. But the localization, as we’ve seen so far, requires we know where objects are, that we have a map. Is this a chicken and egg problem? No, we can do them at the same time. This is called simultaneous localization and mapping, or SLAM.
Localization with Occupancy Maps

Let us assume the existence of multiple laser rangefinders. For example, Figure 2 shows a robot in a room using four laser rangefinders to measure distance.

The basic idea is that we update our map in tandem with our localization. Each particle $k$ will store a pose $x^k$ as well as its own version of the map. The process is to look at each particle, move it as before, compute the weight based on how well the sensors match the map, and update the map on that particle. Once done, Our algorithm is detailed in Algorithm 1.

- Moving the particle: The particles pose is moved just like in particle filter localization. Given some desired motion $u$, and a motion model, each particle is given a different change in pose drawn from that motion model.

- Predict the observations: This is extremely similar to what is done in partile filter localization to provide our $o_p$, except that rather than calculating based on the particles pose and a global map, it calculates based on the particles pose and its own individual map.
Algorithm 1 Particle SLAM with occupancy grid maps

weightsum = 0
for all p ∈ particles do
    Move the particle, as in landmark-based Particle Filters
    p.weight ← 1
    for all o ∈ observations do
        ▶ Weight particles
        o_p ← theoretical perfect measurement from p.location.
        p.weight ← p.weight · pdf(o − c · o_p, 0, σ)
        for all m_i ∈ map cells currently observable do
            ▶ Update map with Bayes Rule
            m_i ← αp(o|m_i)p(m_i)
        end for
    end for
    weightsum ← weightsum + p.weight
end for

Resample just as before.
particles ← newParticles ▶ New, resampled list of particles

- Update the map: This is the trickiest part. Here, we have to update every voxel on the particles map that the rangefinder goes through. The update is easy though - its a Bayes filter! After all, your voxel is either occupied, or not, which is discrete. Have another look at the update for m_i in the above algorithm. Weighting the particle: Again, this is very similar to particle filter localization. Given the predicted observations, the actual observations and the sensor model, what is the likelihood this pose and map is correct? This becomes the particles weight.

- Resampling: This is done exactly like in particle filter localization.

The cool thing is that this works despite it appearing to just ignore the chicken-egg problem. A lot of the magic comes from the particles. If we had tried this with Kalman Filters, the math would get hairy fast, and it wouldn’t work as well.

Challenge

Occupancy maps tend to be BIG, a problem which is exacerbated if you’re doing particle filter-based mapping, where every particle has it’s own map. Suddenly, merely storing enough particles is a challenging problem.

One way to (partially) mitigate this problem is by storing your map using a space-efficient data structure like a Quadtree (or, if in three dimensions, an Octtree). The intuition behind Quadtrees is simple: many maps have wide swaths of open space. Why not compress many small unoccupied voxels into a single big unoccupied voxel?

An example of a quadtree is in Figure 3. The root node corresponds to the entire map (in this case, an 8x8 square. It is gray, because some voxels within are filled, and some are not. Its four children (four, because it’s a quadtree) are arranged in clockwise order: the first corresponds to the 4x4 square in the upper left, the second to the 4x4 square in the
upper right, then lower right, then lower left. The first and third are white because nothing within is occupied. So why store with any more specificity than that?

The second and fourth children of the root note are gray, because some voxels within are occupied and some are not. Because these nodes are gray, they need to be specified more fully with children. If you consider the second child, its four children correspond to the four 2x2 blocks within its 4x4 block. The first is gray, because some voxels are occupied, and some are not. The second is black, because all voxels within are occupied. The third and fourth are white because no voxels within are occupied. The one gray child is the only one which needs to have children stored.

You should be able to go back and forth between occupancy grids and quadtrees.

![Figure 3: Occupancy grid and corresponding quadtree](image)

This approach is great for bitmaps, where voxels are either occupied or unoccupied, but this breaks down when we have probabilities. In quadtrees, groups of cells are merged to one big cell when all the leaf descendants of a node have the same value. Since probabilities on continuous values, this rarely happens and few cells would be merged. The solution is to create $k$-class quadtrees. $k$-class quadtrees are just like regular binary quadtrees, except that instead of 2 classes, there are $k$ classes. The rules are still the same, you merge when all leaf descendants are the same. Our classes will simply be a range of probabilities. For example, class 1 might be anywhere between 0 and 0.1. Storing the map this way is less accurate because we lose precision in our probabilities, but it can compress real world maps up to 20 times.