RoboND Localization - Where Am I ?

Chris Rowe

Abstract—Two mobile robots are designed using URDF in a simulated Gazebo environment. Each are equipped with a laser range finder and a camera. Given a 2D map, they must perform localization, plan a path and navigate to the goal location. The Adaptive Monte Carlo Localization ROS package is used to process laser and odometry data in order to perform probabilistic localization using particle filters. The AMCL parameters will be experimented with and tuned for each of the robots, so that each produces optimal paths unique to their footprint.

Index Terms—Udacity, Kalman, Particle Filters, Monte Carlo Localization, AMCL.

1 Introduction

If you've been to a mall, you've experienced first-hand, the localization process. When you looked around for some kind of unique landmark, identified the stores currently around you, scanned the mall map to find a matching area. And finally, you found it. You successfully localized yourself as soon as you saw that bright, yellow star screaming "You Are Here".

For robots, localization isn't that easy. Although it has a map of the environment, the features don't have a bright, blinking neon sign above them.

Formally, localization is determining the pose of a robot in a mapped environment. There are type scenarios in which to localize. The simplest is the position tracking problem, where the initial pose is known, and the robot can use features to narrow down its location. Secondly, the global localization problem. The initial pose is unknown and features cannot be used to track itself. The hardest, relocalization, or kidnapped-robot scenario. The robot is suddenly in a novel environment and all prior beliefs and data become inapplicable.

Localization is accomplished by analyzing both sensory inputs, like LaserScan and odometry, and control outputs, like motor control commands. It's a trivial task with perfect data. But real world data is inaccurate, noisy and polluted by external forces, a wheel slip, winds, and even small pebbles.

To combat this compilation of errors, we approach the problem with a probabilistic perspective. By framing the problem as a range of possibilities dependent upon previous states, we can use new data to filter out each impossibility until only a manageable range of possibilities is reached. These methods are solved using filtering methods, such as Kalman Filters and Particle Filters.This project serves to implement the latter method to localize the robot.

2 Background

Provide a sufficient background into the scope of the problem technologically while also identifying some of the current challenges in robot localization and why the problem domain is an important piece of robotics. [?]

Localization is a problem of probability. Real world sensor readings contain an uncertainty which compounds with each sample. All these uncertainties would be improbable to calculate. Since we cannot calculate, we must estimate. The algorithms work through filters, which sieve data of higher uncertainty.

2.1 Kalman Filters

Kalman filters are popular in control systems because they can process noisy data, very quickly and they require much less computational power due to their simple algorithm.

Kalman filters works by iterating between measurement updates and state prediction. An initial guess is used to form a prediction. The prediction is then updated using a new measurement. The weight given to that measurement is relative to its expected uncertainty. Those with little expectancy are given a heavier weight, and vice versa. Using this data, a new prediction is formed. And the measurement update and state prediction cycle continues.

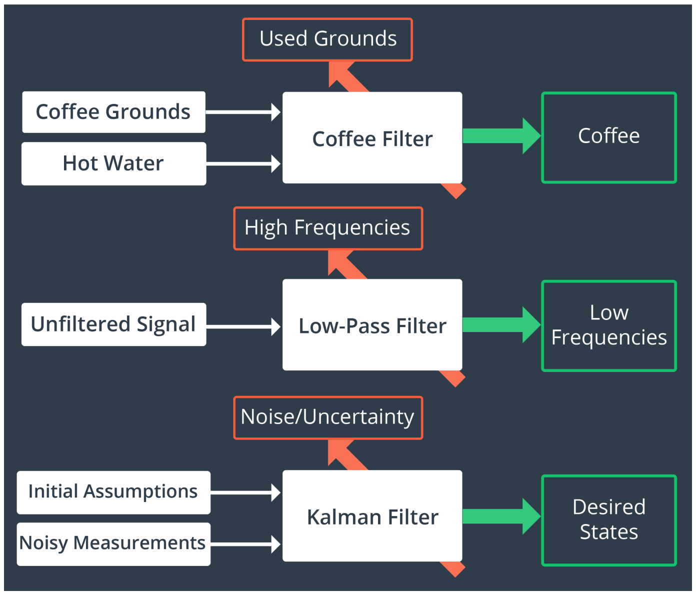


Fig. 1: Kalman Algorithm

Due to its simplicity, it doesn't require as much computational power as compared to other algorithms. The Kalman Filter is what enabled mankind to set foot on the moon. Another great advantage is that it can be applied to fuse data from multiple sensors, extracting and combining the least uncertain from each.

But its simplicity comes with a converse. It will only work under the conditions that the motion and measurement models are linear, and that the state space can be represented by a unimodal distribution. Due to this limitation, the normal Kalman filter will only work on linear problems. Since most real world problems are nonlinear, or multivariate, it must be adjusted.

The Extended Kalman Filter (EKF) is designed to handle nonlinear problems. A nonlinear function is used to update only the mean of a function. Using it to update the variance outputs a non-Gaussian distribution, which cannot be solved in closed form and thus computationally expensive. For the variance, the nonlinear function is linearized over a small section, like a tangent, and then used as the updated state variance.

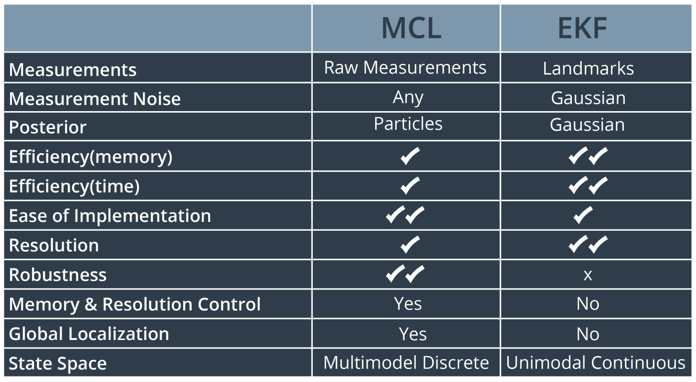
2.2 Particle Filters

Each of the "particles" in the title are a hypothesis of the robot's pose represented by a 3D Pose and a weight. The weight represents the difference between the robot's actual pose and that particle's hypothetical pose. At the start, particles are randomly and uniformly spread around the map. After each cycle, the weights of each particle are calculated and those of lower weight are discarded, while those of larger weight move onto the next resampling process. Almost evolutionary, the cycle continues until all the weak particles are eliminated and only the strong remain, eventually converging on the robot's pose.

2.3 Comparison / Contrast

The KF excels in accuracy estimation. For situations in which the conditions for KF can be fulfilled, it can be very powerful. In position tracking problems of low uncertainty, the KF has the advantage. A major advantage is the ability to draw data from multiple sensors and fuse them. With its low memory and workload requirement, it is ideal for situations where computational power is scarce. However, for linearizing nonlinear problems, the Extended Kalman Filter will impose a much greater workload.

While the EKF is limited by the linearity of a problem, the MCL are not restricted by a linear-based Gaussian, and can handle problems of higher dimensions. This makes their range of applications extend into the nonlinear domain, where the majority of real world problems lie. And thus MCL can handle the difficult problem global localization. But, with this great power comes greater workload. MCL is much more computationally demanding, but the memory usage and resolution can be altered to the required workload. For this project, the particle filter Monte Carlo Algorithm will be used.



	MCL	EKF
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Efficiency(memory)	✓	✓✓
Efficiency(time)	✓	✓✓
Ease of Implementation	✓✓	✓
Resolution	✓	✓✓
Robustness	✓✓	x
Memory & Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodel Discrete	Unimodal Continuous

Fig. 2: EKF vs MCL

3 Simulation Models

Both robots are defined in their respective XACRO files, `udacity_bot.xacro` and `shelly.xacro`, in their respective packages. The most significant dimensions of each are specified in **Table 1**.

They are differentially actuated by two centered wheels on either side. Two centered casters, one in front and the other in the back allow the robots to be balanced. This enables them to turn in place, but neither are holonomic. Although the shapes of their chassis are different in shape, they both share a similar rectangular XY footprint.

The benchmark model, **udacity_bot**, has a box chassis whose dimension is . The centered of gravity was simplified into that of a homogeneous box.

My model, **shelly**, has a cylindrical chassis of radius 0.1m and height 0.4m. In order to match the orientation of the benchmark's box. It is oriented with a pitch angle of -$\pi/2$ (-1.5707) radians. Its center of gravity was simplified into that of a homogenous cylinder. In simulation, it is defined with twice the maximum speed of the benchmark.

Note that, for my model, the physical characteristics of shelly are that of the cylindrical chassis in Fig: 4. As shown in the superimposition, Fig: 5, nothing physical (mass, collision or inertia, have changed. Visually, in Fig: 6, the cylindrical chassis and components were simply made invisible, and visually replaced with a Green Shell from Mario Kart.

TABLE 1: Comparison of Model Characteristics

Parameter	udacity_bot	shelly
collision	box l x w x h 0.2 x 0.4 x 0.1	cylinder r x h 0.1 x 0.4
orientation	r * p * y 0 * 0 * 0	r * p * y 0 * -1.5707 * 0

inertia	ixx=0.1
	iyy=0.1
	izz=0.1
		ixx=0.2375
		iyy=0.075
		izz=0.2375

hokuyo	X x Y x Z 0.15 x 0 x 0.1	X x Y x Z 0.15 x 0 x 0.15

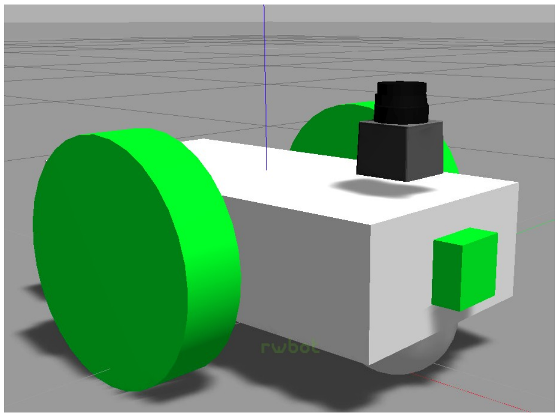


Fig. 3: udacity_bot

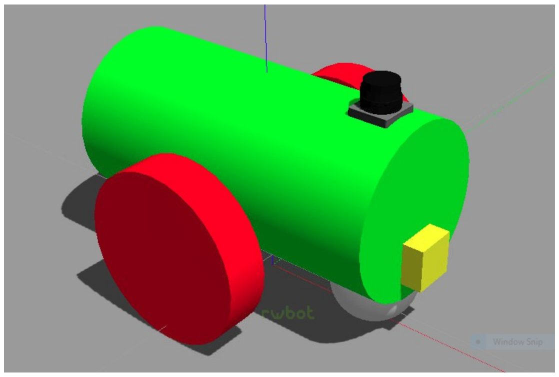


Fig. 4: Shelly's Physical Collision

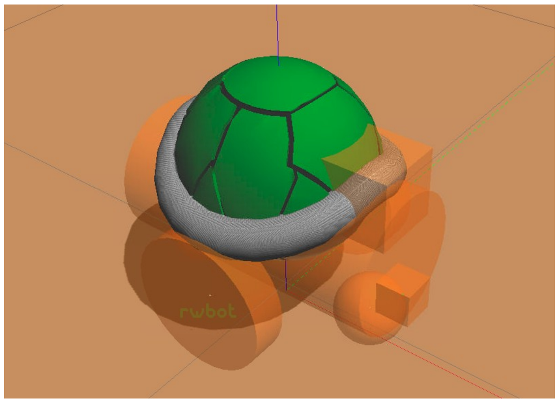


Fig. 5: Shelly's Visual Superimposed Over Collision. The visual representation has changed, but **the collision has not.**

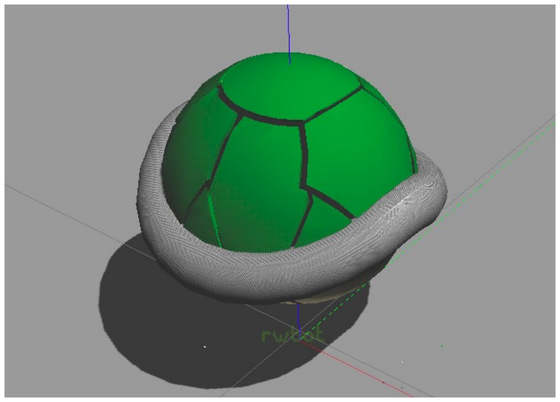


Fig. 6: Shelly - Self Driving Mario Kart Shell [4]

4 ROS Navigation Stack

The ROS Navigation stack performs as a system of ROS packages, each providing a core functionality. In this project, our focus was localization, so the `move_base` and `amcl` packages were implemented.

The `move_base` package incorporates map, odometry and sensor data to evaluate the best path, and provides the robot with the associated velocity commands. It provides the interface between the path-planning nodes, `base_local_planner` and `base_global_planner` and the costmap nodes, `local_costmap` and `global_costmap`. The `amcl` node performs probabilistic localization by implementing the Adaptive Monte Carlo algorithm, thus keeping track of the robot's pose in relation to its environment.

4.1 Costmap

The package `costmap_2d` creates an occupancy grid map - a 2D representation of the space around the robot. The generated map is composed of 3 superimposed layers: static, obstacle, inflation. The static layer directly interprets the given static map provided to the navigation stack. In the obstacle layer, each detected cell in the grid is categorized as an **unknown**, **occupied**, or **free** space. These values are then passed to the planners, `base_local_planner` and `base_global_planner` which calculates the cost of each cell, and then inflates the value of the cells according to their respective parameters. Costmaps are covered more in the section 5.

4.2 Adaptive Monte Carlo Localization

General Parameters

`odom_model_type` {diff-corrected} Defines the odometry model to be used. Either `diff`, `omni`, `diff-corrected` or `omni-corrected`.

`odom_frame_id` {odom} Defines the odometry transform frame.

`base_frame_id` {robot_footprint} Defines the robot base transform frame.

`global_frame_id` {map} Defines the transform frame published by localization node.

Filter Parameters

`<min,max>_particles` {10:100} Defines the minimum and maximum amount of particles to be used during localization.

`initial_pose_<x,y,a>` {0,0,0} Defines the initial pose of the robot upon spawning.

`update_min_d` {0.01} Translational movement (meters) required before performing a filter update.

`update_min_d` {0.01} Rotational movement (radians) required before performing a filter update.

Laser Parameters

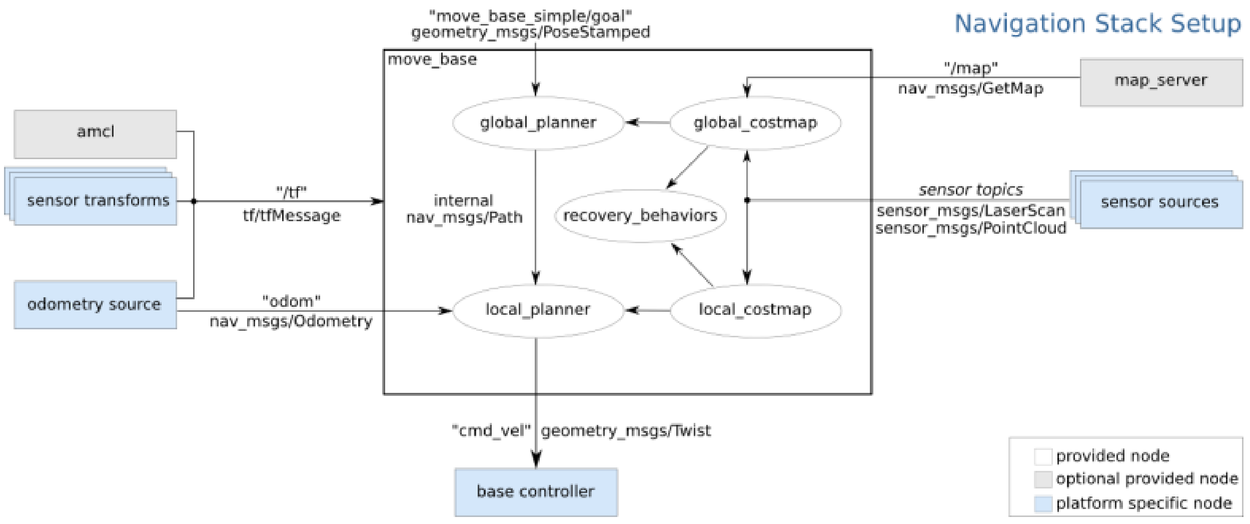


Fig. 7: Navigation Stack Overview [1]

laser_model_type {likelihood_field} Defines which laser beam model is to be used. Either `likelihood_field`, or `likelihood_field_prob` (same as `likelihood_field` but incorporates the beamskip feature.

laser_min_range {0.4} Minimum scan range considered to be valid.

4.3 Base Local Planner

The `base_local_planner` uses a global plan (the complete path from start to finish) along with its local costmap to calculate the optimal trajectory for the next step, aka local plan. The planner subscribes to the odometry and laser data, and so is able to dynamically compute an alternate trajectory as necessary when there is a change in the environment. base_local_planner employs the Trajectory Rollout and Dynamic Window Approach (DWA) algorithms. They both use the same workflow, the only difference being the period of simulation time.

The algorithms begin by sampling possible velocities (dX, dY, dTheta). Each velocity sample is then projected forward in time for anywhere between 0.5 to 10 or more seconds, based on the processor power available. Each projection's trajectory cost is then calculated based upon the trajectory's proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Trajectories that result in a collision with an obstacle are discarded, and the trajectory with the lowest cost is selected. The trajectory's associated velocity is then sent to the robot. The formula used to calculate trajectory costs, C, is:

$$Cost = pdist\_scale * d_P + gdist\_scale * d_G + occdist\_scale * C_0$$

where d_P=(distance to path from trajectory endpoint), d_G=(distance to local goal from trajectory endpoint), C_0=(max obstacle cost (0-254)).

The parameters for the local planner are located in `base_local_planner_params.yaml`.

xy_goal_tolerance {0.2} Defines the tolerance in meters for the controller in the X & Y distance when achieving a goal.

yaw_goal_tolerance {0.1} Defines the tolerance in radians for the controller in yaw/rotation when achieving its goal

pdist_scale {1.0} Defines the weight for how much the planner should stay close to the global path. A high value allows generates local paths that deviate less from the global path.

gdist_scale {0.4} Defines the weight for how much the robot should attempt to reach the local goal. A high value allows flexibility in the local path.

occdist_scale {0.01} Defines the weight for how much the planner should attempt to avoid obstacles.

holonomic_robot {false} Defines whether the robot can travel omni-directionally. Being a differential drive robot, ours is not holonomic.

publish_cost_grid_map {true} Enables the local costmap to be visualized as a heatmap in Rviz.

global_frame_id {odom} Defines the transform associated with `publish_cost_grid_map`.

5 Costmap

The costmaps are an essential component of the navigation stackund them to have the most significant impact on the robot's navigation. It provides a metric of the navigability of the robot's surrounding space. It is crucial that the costmap be as accurate as possible. Otherwise, what should be a clear hallway could be perceived as a wall, and vice versa. The costmap is comprised of 3 layers: the static layer, the obstacle layer and the inflation layer. Both costmaps use the obstacle and inflation layers, but only the `global_costmap` uses the static layer.

Local Costmap

The `local_costmap` is a **dynamic** map with that represents trajectory costs in the robot's immediate space. As such, it is centric to the robot and so moves with the robot. It is a dynamic map, generated by detecting obstacles in real time user a LaserScanner data. It is configured according to the parameters in the `local_costmap_params.yaml` file.

global_frame {odom} Specifies the transform frame to be used as the global frame

length,width {4,4} The dimensions of the local map

static_map {false} Specifies if a static map is being used

rolling_window {true} Specifies whether the costmap will remain centered around the robot as it moves. If the `static_map` parameter is set to true, this parameter must be set to false.

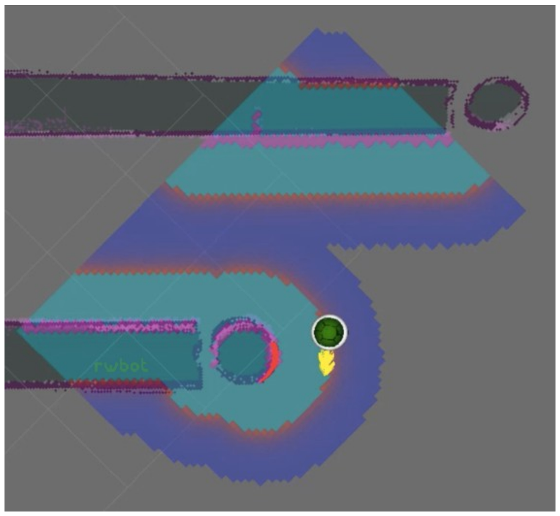


Fig. 8: Local Costmap

Global Costmap

The `global_costmap` is a **static** map, describing all the space in the robot's navigable workspace. It is generated once, before anything else, and does not change even when a new obstacle is detected. This is because the robot needs to maintain the path its traveled so far in order to localize itself. If the `global_costmap` changes, AMCL needs to be restarted. It uses the map passed from The `map_server`, inflating obstacles on the map provided to the navigation stack, according to the parameters in the `global_costmap_params.yaml` config file.

global_frame {map} Specifies the transform frame to be used as the global frame

length,width {10,10} The dimensions of the global map

static_map {true} Specifies if a static map is being used

rolling_window {false} Specifies whether the costmap will remain centered around the robot as it moves. If the `static_map` parameter is set to true, this parameter must be set to false.

Common Costmap

These parameters are shared by both local and global costmaps, and are specified in `costmap_common_params.yaml`.

observation_sources {laser_scan_sensor} Defines a list of sensors that are going to be passing information to the costmap.

laser_scan_sensor Defines the parameters of the sensor as mentioned above in `observation_sources`. `sensor_frame` is {hokuyo}, `data_type` is {LaserScan}, `topic` name is {/udacity_bot/laser/scan}, `marking` and `clearing` are {true}.

footprint Defines the physical contour of the robot's base. [[0.2, 0.2], [-0.2, 0.2], [-0.2, -0.2], [0.2, -0.2]]

robot_base_frame {robot_footprint} Defines the coordinate frame the costmap should reference for the base of the robot.

obstacle_range {4} The maximum range in meters at which to insert obstacles into the costmap using LaserScan data.

raytrace_range {4} The maximum range in meters at which to insert obstacles into the costmap using LaserScan data.

always_send_full_costmap {true} If true the full costmap is published to "/costmap" every update. If false only the part of the costmap that has changed is published on the "/costmap_updates" topic.

resolution {0.05} Resolution of each cell, in meters/cell, of the costmap.

update_frequency {10} Defines the rate, in Hertz, at which the costmap will run its update loop.

publish_frequency {10} Defines the rate, in Hertz, at which the costmap will publish visualization information.

transform_tolerance {0.2} Defines the delay in transform (tf) data that is tolerable in seconds.

cost_scaling_factor {8} A scaling factor to apply to cost values during inflation. The cost function is computed for all cells further than the inscribed radius distance and closer than the inflation radius distance away from an actual obstacle.

inflation_radius {1.5} Defines the radius in meters to which the map inflates obstacle cost values.

5.1 Static Layer

Before spawning the robot, the navigation stack is passed the map `jackal_race.world` by the in `udacity_world.launch` file, which generates a world space as defined. This is the map used to generate the **static_layer** `obstacle_layer`. Only the `global_costmap` includes a static layer. As implied, the static map never changes, even when there is a change in the environment. Since we are trying to localize, this is important because this map will be used by the the robot as a reference to which it will attempt to match the current snapshot of its `local_costmap` - just how we search for the "You Are Here" star on the maps of giant malls to know where we are.

5.2 Obstacle Layer

Conversely to the `static_map`, the `obstacle_map` is generated in real time. It subscribes to the available sensors, which can be assigned to be used for *marking* or *clearing*, or both. In our case, we used on LaserScan sensor and used it for both.

The `obstacle_layer` is where the occupancy grid map is updated. Each cell represents an XY coordinate of the space, as seen from above. Similar to a "Color by Number" coloring book, each cell in the `obstacle_layer`

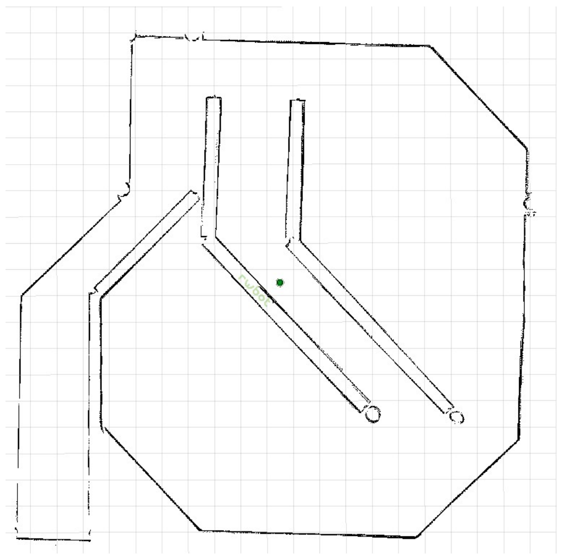


Fig. 9:
The static_layer & obstacle_layer of global_costmap.

is assigned a number to identify the status of that space as either free, occupied or unknown. **Marking** is the insertion of a detected obstacle. **Clearing** is the removal of a space previously detected as an obstacle, such as when an obstacle has moved. Dynamically updating the `obstacle_layer` ensures that the `local_costmap` has the most accurate representation of that space at that time. In the case of the `global_costmap`, the `obstacle_layer` is processed only once, immediately after, and using, the `static_layer`.

5.3 Inflation Layer

The `inflation_layer` is the final layer, and step, in the creation of the costmaps. It expands upon the representation of cell states in the occupancy grid map made by the `obstacle_layer`. It converts the trinary representation of free, occupied or unknown, into a spectrum of 255 possible cost values.

It uses the robot's footprint to propagate the cost values out from occupied cells that decrease with distance, as depicted by the decay curve in ***Fig: 13***.

A circumscribed circle is drawn around the footprint, and an inscribed circle is drawn inside the footprint, with the centerpoints representing the center of the robot. Using these radii, the cost values are organized into 5 categories.

- **Freespace (0)**: Distances where absolutely no obstacles are present. Full speed ahead!
- **Buffer (1-127)**: Distances between the circumscribed circle and the `inflation_radius` representing a detected obstacle is present, but not close enough to cause a collision. This acts as a buffer which the `local_planner` will take into account when calculating paths. It is crucial to tune the cost gradient (1-127) such that it is as gradual as possible. More detail in ***Fig: 13*.**
- **Possible Collision (128-252)**: Distances between the ***inscribed radius (purple)*** and the ***circumscribed radius (red)***, representing the possibility of a collision. For a spherical footprint, this wouldn't be necessary. But the majority of robot footprints are non-spherical, so a collision is dependent on the orientation of the robot.
- **Inscribed (253)**: Distances equivalent to the inscribed radius are in the range, representing the point at which collision is confirmed.
- **Lethal (254)**: Distances less than the inscribed radius, meaning there is an actual obstacle in that cell and the robot will definitely collide with it.

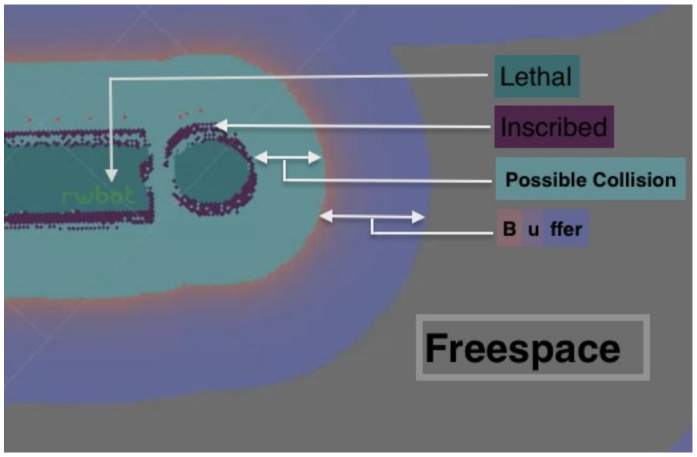


Fig. 10: A color-coded example of the ranges in the Inflation Layer of a global_costmap.

On the diagram in Fig: 10, the **Freespace** range, all distances greater than the `inflation_radius`, range is colored grey.

The **Buffer**, all distances between `inflation_radius` and the circumscribed circle, is colored as a gradient beginning from dark blue and ending at red. The cost is lowest at the dark blue-grey boundary. Increasing towards red, the highest cost is where the it is most red. Where the red boundary meets the circumscribed circle at the light blue-red boundary, is the inscribed circle.

The **Possible Collision** range, distances between the circumscribed and inscribed radii, is represented as light-blue.

The **Inscribed** boundary, distances equal to the radius of the inscribed circled, is purple.

Finally, the **Lethal** range, any distance greater than the inscribed circle, is represented as teal.

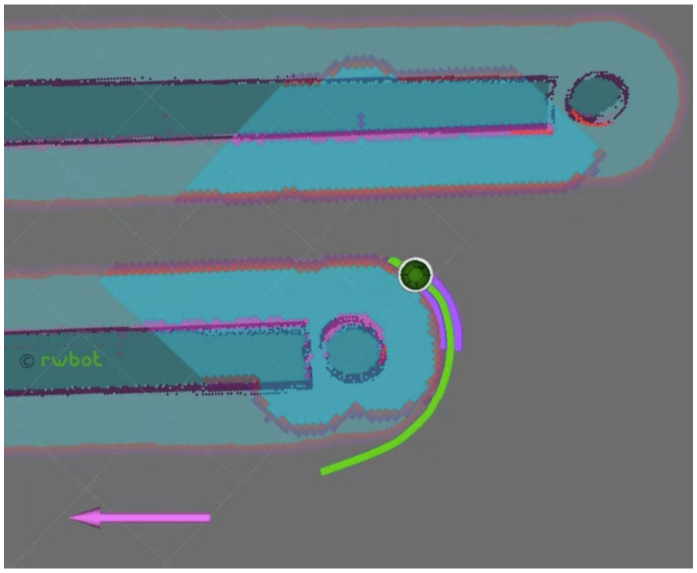
5.4 Tuning Costmap Parameters

Inflation Layer

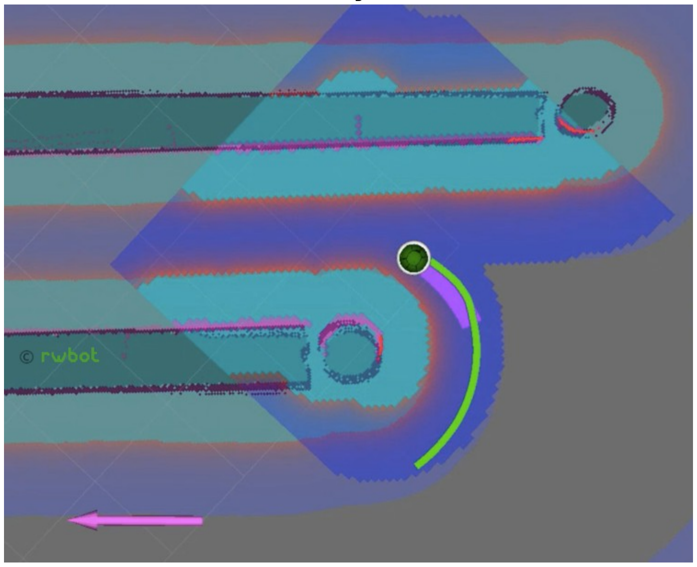
Since the trajectory calculations done by `base_local_planner` are dependent upon the costmap, tuning the inflation layer is crucial. To explain, I will be using the accompanying the screenshots in Fig: 11.

The untuned, default values are used in Fig: 11a, while Fig: 11b represents my tuned values. The blue around all obstacles is the `global_costmap`. The `local_costmap` is the square around the robot, as shown in Fig: 8. The blue inside the square is brighter than the surrounding blue because it is where where the two costmaps overlap.

The difference between the two is that the costmap in Fig: 11a doesn't seem to have a **Buffer** range at all? *Why?*



(a)
`inflation_radius`: 0.5
`cost_scaling_factor`: 10



(b)
`inflation_radius`: 1.5
`cost_scaling_factor`: 8

Fig. 11: Comparison between untuned and optimal configurations of the inflation layer

Fig: 11a does indeed have a buffer range. But its gradient is so *steep* that the boundaries are essentially the same line.

This creates a problem. As mentioned before, the `local_planner` depends on the accuracy of the ranges in the costmaps to create a calculate a path with the least amount of obstacles and distance (cost). When planning paths *around* an obstacle, like the U-turn, the smaller the turning radius, the smaller the distance and the smaller the cost. And the shortest path is found as far inside the buffer zone as possible.

So here the problem arises. With a cost gradient so steep, as to cause the buffer range's boundaries indistinguishable, crossing into the **Possible Collision** range is inevitable. So more times than not, there will be a collision. And as a safety feature, the `base_local_planner` is designed to immediately stop upon colliding with an obstacle. It then enters recovery mode, which employs various attempts to exit into a safer range. However, the attempts rarely work, and at that point the `base_local_planner` will declare that an irrecoverable error has occurred and shuts down movement.

With the robot now paralyzed as in 11a, the run was a failure. The only way to reach the goal point was by manually splitting the `global_costmap` into multiple poses, which would be counter-intuitive in our attempt to employ autonomous navigation.

The solution is simple. As described by the cost curve, ***Fig:*** *13*, the gradient must be as gradual as possible. This is done by balancing the only two parameters of the `inflation_layer` - **`inflation_radius`** and **`cost_scaling_factor`**.

They are tuned by first setting both parameters to zero. The optimal **Buffer** range has a radius at least wider than the robot, and the gradient, represented as a red-blue gradient, should be mostly blue, but also having an area of red at least half the width of the robot.

`inflation_radius` is found first, by increasing it until it is at least wider than the robot. It should be noted that **`cost_scaling_factor`** is inversely proportional to **`inflation_radius`**. If **`cost_scaling_factor`** is too small, there will be too much red. If too large, it will start decreasing the **`inflation_radius`**.

When corrected, obstacles are no longer a death sentence for the robot. Now, it takes corners beautifully as shown in my optimal configuration in Fig: 11b.

6 Parameter Tuning

Tuning Local Planner

TABLE 2: Important Local Planner Parameters

Parameter	udacity_bot	shelly
yaw_tolerance	0.1	0.15
max_vel_x	0.5	1.0
sim_time	1.7	1.7
vx_samples	20	20
vtheta_samples	20	40
pdist_scale	1.0	2.0
gdist_scale	0.4	0.4
occdist_scale	0.001	0.001

- **`yaw_goal_tolerance`**: Increased from default (0.05) because the robots would repeatedly overshoot when turning in place. `shelly` required more tolerance due to her increased speed.
- **`max_vel_x`**: Doubled `shelly`'s speed.
- **`sim_time`**: The higher the `sim_time`, the longer the trajectory. Since the trajectories created are inherently "arcs", the radii of non-straight trajectories, especially around corners, become too curved, and the robots

would tend to chase the endpoint of the local trajectory more than they do the global path. Around corners, this caused them to swerve into the obstacles.

- **vtheta_samples**: Taking more rotational velocity samples increased the resulting trajectory's consistency. The paths created had much less minute oscillations, increasing the smoothness of the paths.
- **occdist_scale**: This controls the weight of obstacle avoidance. Increasing this would caused the robots to occasionally stop completely if a path was too close to an obstacle. So it was left alone.
- **pdist_scale** & **gdist_scale**: The weight of the `global_path` is defined by `pdist_scale`, while the weight of the `local_path` is defined by `gdist_scale`.

  Since the trajectories of `global_path` generally chose smarter paths, and were much less mercurial, the weight of `global_path` needed to be emphasized. Since these two parameters work in conjunction, they must be tuned together.

  For both robots, the weight of **gdist_scale** was cut in half from 0.8 to 0.4.

  For `udacity_bot`, the weight of **pdist_scale** was increased from its default, 0.6 to 1.0.

  Since `shelly` was twice as fast, the weight of **pdist_scale** needed to account for that, and thus doubled to 2.0. The path must be smooth or else minute variations would cause the base to veer significantly from side to side.

Tuning AMCL

TABLE 3: AMCL Parameters

Parameter	udacity_bot	shelly
laser_min_range	0.4	0.4
update_min_d	0.01	0.01
update_min_a	0.01	0.01
min_particles	10	10
max_particles	100	40

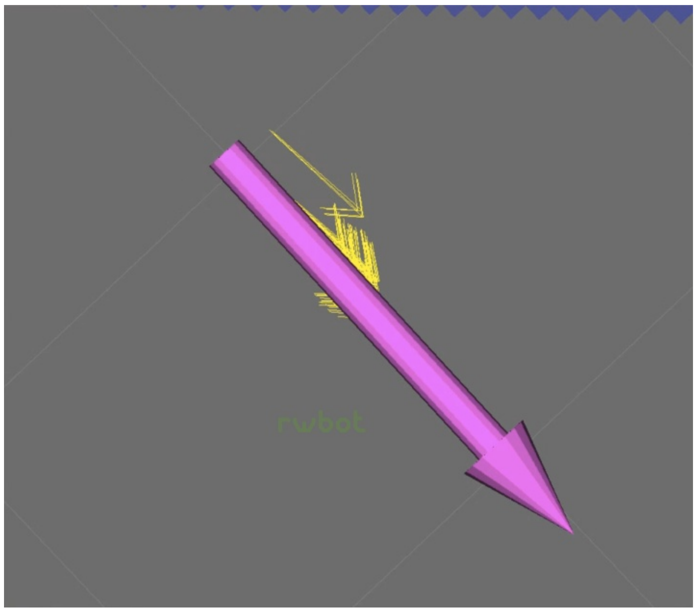
- **laser_min_range**: At short distances, distances close to the robot would fluctuate, so a minimum range of 0.4 meters was defined.
- **update_min_d** & **update_min_a**: The default minimum translational (d), and rotational (a) movement required before performing a filter update was 0.2(m) and $\pi/6$(rad).

  For robots with a 0.2(m) length moving at $\approx$ 0.5(m/s) and $\approx$ 1.0(m/s), filters were only updated at 2.5(Hz) and 5(Hz) respectively. Increasing the frequency to led to more accurate localization, and reduced the minimum and maximum number of particles necessary.
- **min_particles** & **max_particles**: Due to the increased update frequency as mentioned above, less particles were necessary. Both had a minimum of 10, decreased from the default 100.

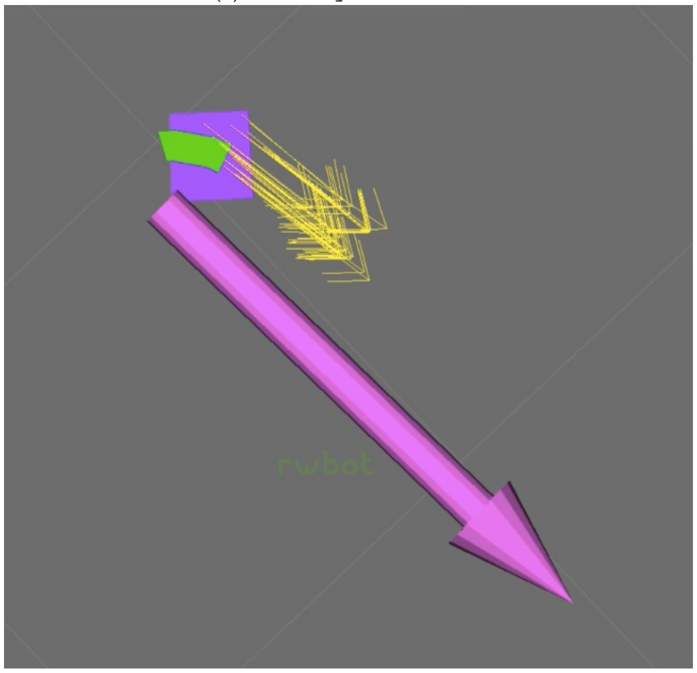
  For `udacity_bot`, the necessary `min_particles` needed was reduced to 100!

  Since `shelly` was twice as fast, and with the same update frequency, the necessary `max_particles` needed dropped from the default 5000 to just 40!

7 Results and Discussion



(a) `udacity_bot` results



(b) `shelly results`

Fig. 12: Localization Results

Both models generated very smooth global paths. Due to **shelly**'s doubled speed, there were small, but noticeable oscillations in the local path. To compensate, I doubled the weight of the global path as well. This fixed the problem, as the global plan was much more consistent. **udacity_bot** generated smooth and consistent for local and global plans.

Both models were able to successfully reach the goal position within a minute. **udacity_bot** cut it close with a mean of 58s, with little variance. **shelly** was much faster, reaching in at an average of 45s. Since **shelly** was given twice the maximum speed, I expected her to clock in around 35s. When the goal was within her local map, she started to decelerate rapidly. She usually gets to the goal around 38s and spends the next 7s slowly inching and rotating into the goal pose.

Conversely, what **shelly** gave up for speed, was traded for accuracy. While all her arrows were in the correct orientation, they ended up alongside the Pose arrow. This is expected since her goal tolerances were slightly higher. This was a compensation due to her lessened ability to turn in place. This is most likely due to an error in the alignment of her wheels. Thus, in accuracy, **udacity_bot** is the clear winner. The majority, less a couple arrows, were all parallel and superimposed.

The ROS implementation of AMCL has attempted to address the kidnapped robot problem with an optional recovery method. The parameters, disabled by default, `recovery_alpha_slow` and `recovery_alpha_fast`, enable recovery functionality using the addition of random poses. However, I have never been able to get it to recover. This is attributed to the fact that, similar to Kalman Filters, MCL is fundamentally a Bayesian filtering algorithm - all of which rely heavily on an initial belief.

AMCL has proved to be a reliable method of localization, given that an adequate map and initial pose is provided. The most successful implementations would be in environments with a static or predictable environmental layout. This requirement is satisfied best by autonomous environments, with little human intervention to disrupt the environment. The most obvious applications would be any manufacturing plant, facility or assembly line using robots of a moderate level of autonomy: Amazon's warehouses, underground parking lots, drone-mail, autonomous farms etc.

8 Conclusion / Future work

Two robots, **udacity_bot** and **shelly**, were modeled from XACRO files and placed in the maze environment to assess their AMCL parameter space. Although many of the parameters were similar, each model needed to be tuned to accommodate a problem unique to each.

shelly can be improved in several areas. As recommended [5], find the actual velocity limits of the wheels would remove the need to find workarounds for random quirks of the simulation software. A LIDAR unit could be used to generate LaserScan data with higher precision and resolution. Most importantly, the issue to address is her difficulty rotating in space. This will allow much more accurate orientation alignment at the goal.

Hardware Deployment

For deployment, first the actual physical variables must found: motor power, weight, inertia, friction etc. A very simplified simulation model could be created to stress test and gain an idea of where future problems may possibly arise. The laser sensor parameters must be updated to reflect the model/type to be used. Unlike the simulation, the odometry data must be will be riddled with noise. It must be tested in order to establish a baseline and to calibrate the odometry alpha parameters.

The processing capabilities of the onboard system must be assessed. The parameters known to increase workload (particle number, transform tolerances, map resolution, forward simulation time) should be tuned to find a balance between latency and accuracy. And it must be kept in mind that a reliable global map must be provided.

References

[1] *ROS Wiki*, Setup and Configuration of the Navigation Stack
[2] *ROS Wiki*, Inflation Cost Curve
[3] *ROS Wiki*, xacro, base_local_planner, amcl, costmap_2d, move_base
[4] *Maze-Man*, Mario Power Ups, 3D Warehouse Sketchup
[5] *Kaiyu Zheng*, ROS Navigation Tuning Guide
[6] Sebastian Thrun, Wolfram Burgard and Dieter Fox, *Probabilistic Robotics*, The MIT Press, 2005

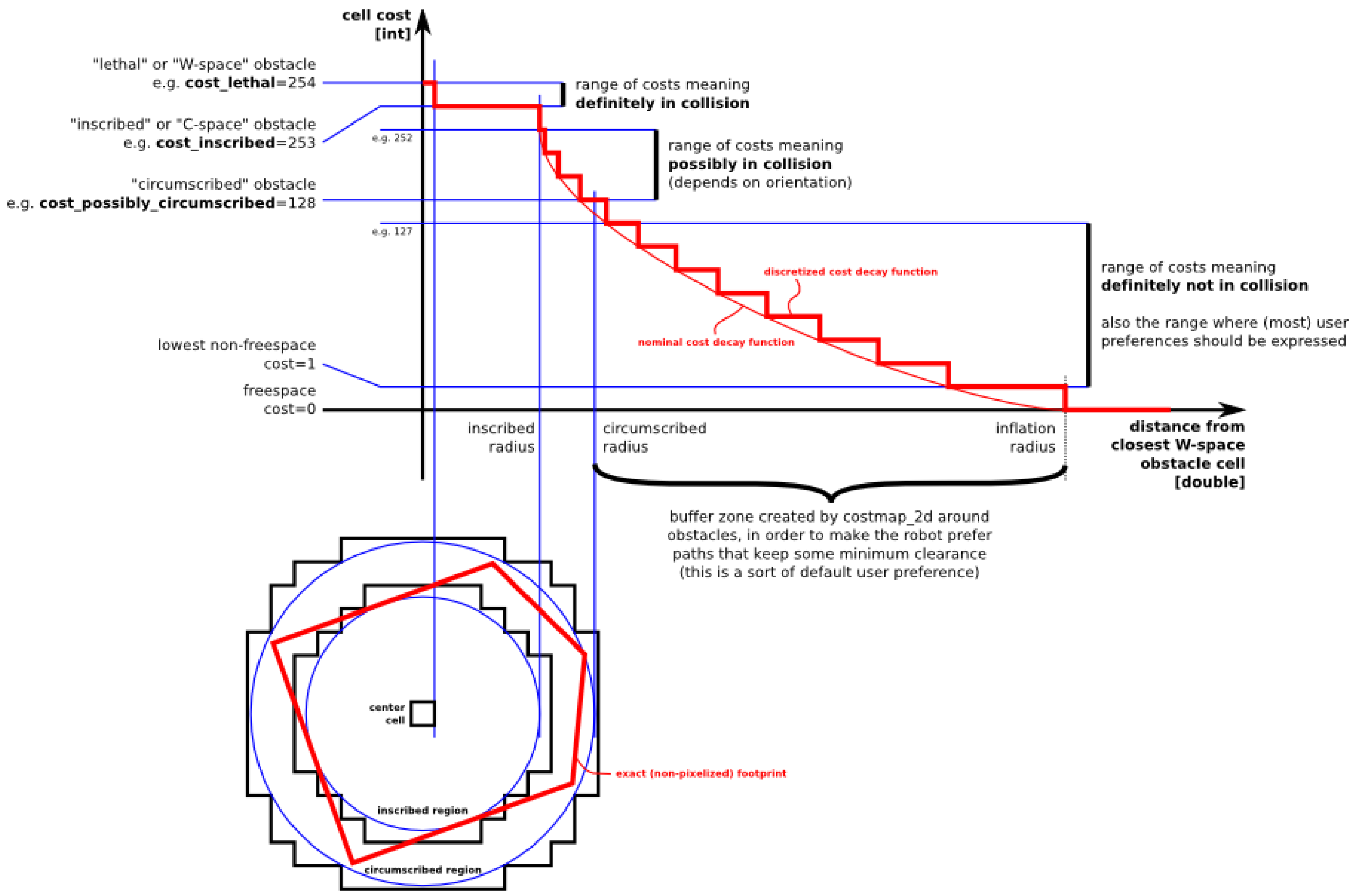

cell cost [int]
"lethal" or "W-space" obstacle e.g. cost_lethal=254
"inscribed" or "C-space" obstacle e.g. cost_inscribed=253
"circumscribed" obstacle e.g. cost_possibly_circumscribed=128
lowest non-freespace cost=1
freespace cost=0
e.g. 252
e.g. 127
range of costs meaning definitely in collision
range of costs meaning possibly in collision (depends on orientation)
range of costs meaning definitely not in collision
also the range where (most) user preferences should be expressed
discretized cost decay function
nominal cost decay function
inscribed radius
circumscribed radius
inflation radius
distance from closest W-space obstacle cell [double]
buffer zone created by costmap_2d around obstacles, in order to make the robot prefer paths that keep some minimum clearance (this is a sort of default user preference)
center cell
inscribed region
circumscribed region
exact (non-pixelized) footprint

Fig. 13:
Inflation Cost Curve [2]: How the `inflation_layer` assigns cell costs using the robot's footprint (red).