# ROS Navigation Tuning Guide

Kaiyu Zheng

September 2, 2016

## Introduction

The ROS navigation stack is powerful for mobile robots to move from place to place reliably. The job of navigation stack is to produce a safe path for the robot to execute, by processing data from odometry, sensors and environment map. Maximizing the performance of this navigation stack requires some fine tuning of parameters, and this is not as simple as it looks. One who is sophomoric about the concepts and reasoning may try things randomly, and wastes a lot of time.

This article intends to guide the reader through the process of fine tuning navigation parameters. It is the reference when someone need to know the "how" and "why" when setting the value of key parameters. This guide assumes that the reader has already set up the navigation stack and ready to optimize it. This is also a summary of my work with the ROS navigation stack.

## Topics

# 1 Velocity and Acceleration

This section concerns with synchro-drive robots. The dynamics (e.g. velocity and acceleration of the robot) of the robot is essential for local planners including dynamic window approach (DWA) and timed elastic band (TEB). In ROS navigation stack, local planner takes in odometry messages ("odom" topic) and outputs velocity commands ("cmd_vel" topic) that controls the robot's motion.

Max/min velocity and acceleration are two basic parameters for the mobile base. Setting them correctly is very helpful for optimal local planner behavior. In ROS navigation, we need to know translational and rotational velocity and acceleration.

## 1.1 To obtain maximum velocity

Usually you can refer to your mobile base's manual. For example, SCITOS G5 has maximum velocity 1.4 m/s[1]. In ROS, you can also subscribe to the `odom` topic to obtain the current odometry information. If you can control your robot manually (e.g. with a joystick), you can try to run it forward until its speed reaches constant, and then echo the odometry data.

*Translational velocity* $(m/s)$ is the velocity when robot is moving in a straight line. Its max value is the same as the maximum velocity we obtained above. *Rotational velocity* $(rad/s)$ is equivalent as angular velocity; its maximum value is the angular velocity of the robot when it is rotating in place. To obtain maximum rotational velocity, we can control the robot by a joystick and rotate the robot 360 degrees after the robot's speed reaches constant, and time this movement.

For safety, we prefer to set maximum translational and rotational velocities to be lower than their actual maximum values.

## 1.2 To obtain maximum acceleration

There are many ways to measure maximum acceleration of your mobile base, if your manual does not tell you directly.

In ROS, again we can echo odometry data which include time stamps, and them see how long it took the robot to reach constant maximum translational velocity $(t_i)$. Then we use the position and velocity information from odometry (nav_msgs/Odometry message) to compute the acceleration in this process. Do several trails and take the average. Use $t_t, t_r$ to denote the time used to reach translationand and rotational maximum velocity from static, respectively. The maximum translational acceleration $a_{t,max} = \max\, dv/dt \approx v_{max}/t_t$. Likewise, rotational acceleration can be computed by $a_{r,max} = \max\, d\omega/dt \approx \omega_{max}/t_r$.

[1]This information is obtained from MetraLabs's website.

### 1.3 Setting minimum values

Setting minimum velocity is not as formulaic as above. For minimum translational velocity, we want to set it to a large negative value because this enables the robot to back off when it needs to unstuck itself, but it should prefer moving forward in most cases. For minimum rotational velocity, we also want to set it to negative (if the parameter allows) so that the robot can rotate in either directions. Notice that DWA Local Planner takes the absolute value of robot's minimum rotational velocity.

### 1.4 Velocity in x, y direction

$x$ **velocity** means the velocity in the direction parallel to robot's straight movement. It is the same as translational velocity. $y$ **velocity** is the velocity in the direction perpendicular to that straight movement. It is called "strafing velocity" in `teb_local_planner`. $y$ velocity should be set to zero for non-holonomic robot (such as differential wheeled robots).

## 2 Global Planner

### 2.1 Global Planner Selection

To use the `move_base` node in navigation stack, we need to have a global planner and a local planner. There are three global planners that adhere to `nav_core::BaseGlobal Planner` interface: `carrot_planner`, `navfn` and `global_planner`.

#### 2.1.1 carrot_planner

This is the simplest one. It checks if the given goal is an obstacle, and if so it picks an alternative goal close to the original one, by moving back along the vector between the robot and the goal point. Eventually it passes this valid goal as a plan to the local planner or controller (internally). Therefore, this planner does not do any global path planning. It is helpful if you require your robot to move close to the given goal even if the goal is unreachable. In complicated indoor environments, this planner is not very practical.

#### 2.1.2 navfn and global_planner

`navfn` uses Dijkstra's algorithm to find a global path with minimum cost between start point and end point. `global_planner` is built as a more flexible replacement of `navfn` with more options. These options include (1) support for A$*$, (2) toggling quadratic approximation, (3) toggling grid path. Both `navfn` and global planner are based on this paper:

http://cs.stanford.edu/group/manips/publications/pdfs/Brock_1999_ICRA.pdf

## 2.2 Global Planner Parameters

Since `global_planner` is generally the one that we prefer, let us look at some of its key parameters. Note: not all of these parameters are listed on ROS's website, but you can see them if you run the rqt dynamic reconfigure program: with

```
rosrun rqt_reconfigure rqt_reconfigure
```

We can leave `allow_unknown`(true), `use_dijkstra`(true), `use_quadratic`(true), `use_grid_path`(false), `old_navfn_behavior`(false) to their default values. Setting `visualize_potential`(false) to true is helpful when we would like to visualize the potential map in RVIZ.

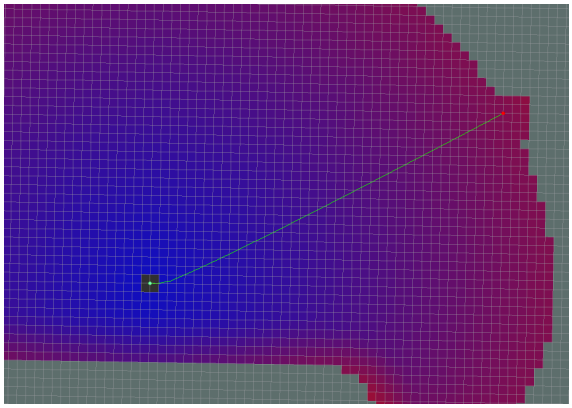

Figure 1: Dijkstra path

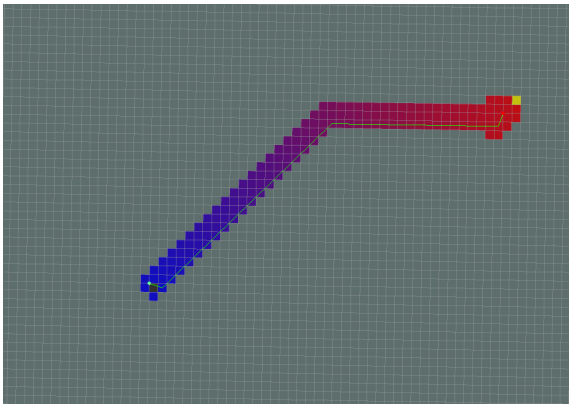

Figure 2: A* path

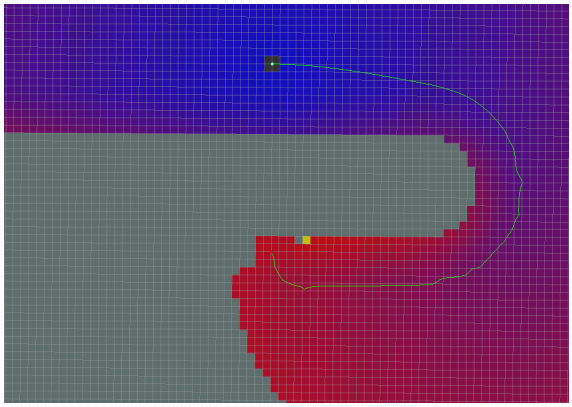

Figure 3: Standard Behavior

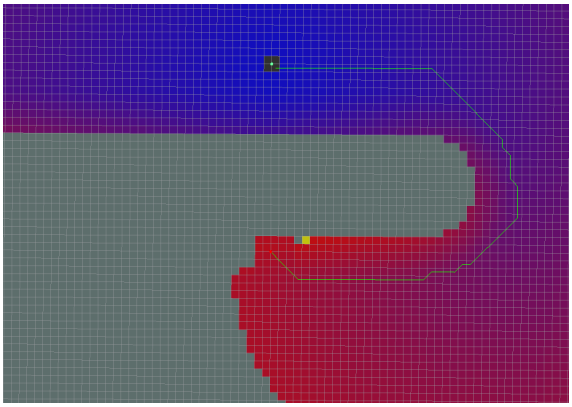

Figure 4: Grid Path

Besides these parameters, there are three other unlisted parameters that actually determine the quality of the planned global path. They are `cost_factor`, `neutral_cost`, `lethal_cost`. Actually, these parameters also present in `navfn`. The source code[2] has one paragraph explaining how `navfn` computes cost values.

[2] https://github.com/ros-planning/navigation/blob/indigo-devel/navfn/include/navfn/navfn.h

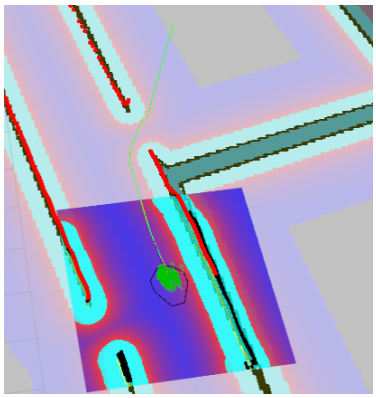

Figure 5: `cost_factor` = 0.01

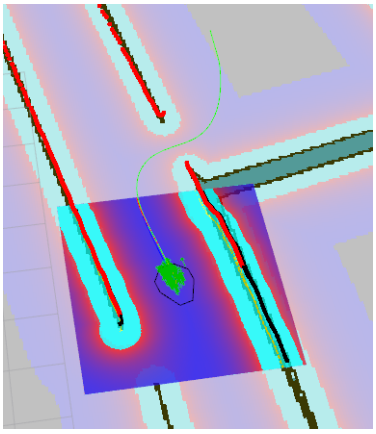

Figure 6: `cost_factor` = 0.55

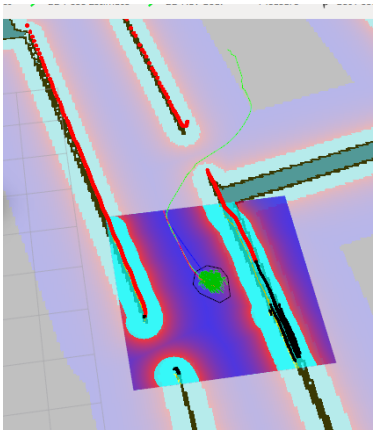

Figure 7: `cost_factor` = 3.55

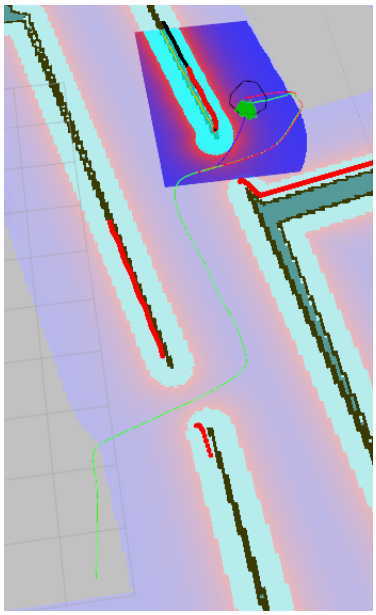

Figure 8: `neutral_cost` = 1

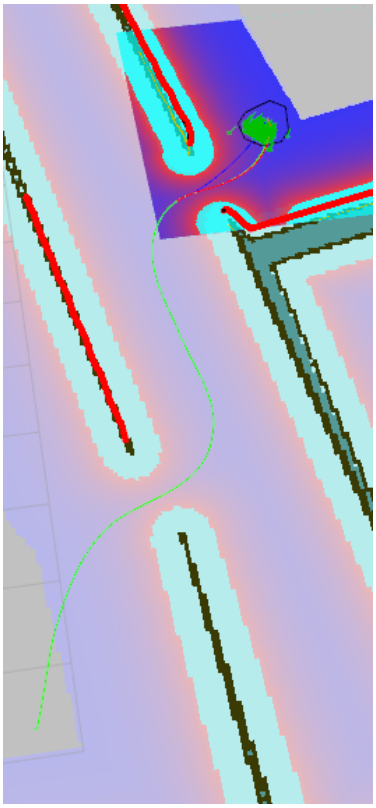

Figure 9: `neutral_cost` = 66

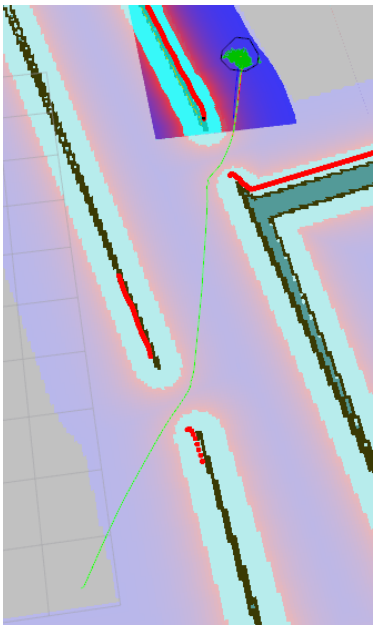

Figure 10: `neutral_cost` = 233

navfn cost values are set to

```
cost = COST_NEUTRAL + COST_FACTOR * costmap_cost_value.
```

Incoming costmap cost values are in the range 0 to 252. The comment also says:

> With COST_NEUTRAL of 50, the COST_FACTOR needs to be about 0.8 to ensure the input values are spread evenly over the output range, 50 to 253. If COST_FACTOR is higher, cost values will have a plateau around obstacles and the planner will then treat (for example) the whole width of a narrow hallway as equally undesirable and thus will not plan paths down the center.

**Experiment observations** Experiments have confirmed this explanation. Setting cost_factor to too low or too high lowers the quality of the paths. These paths do not go through the middle of obstacles on each side and have relatively flat curvature. Extreme neutral_cost values have the same effect. For lethal_cost, setting it to a low value may result in failure to produce any path, even when a feasible path is obvious. Figures $5-10$ show the effect of cost_factor and neutral_cost on global path planning. The green line is the global path produced by global_planner.

After a few experiments we observed that when cost_factor $= 0.55$, neutral_cost = 66, and lethal_cost = 253, the global path is quite desirable.

# 3 Local Planner Selection

Local planners that adhere to nav_core::BaseLocalPlanner interface are dwa_local_planner, eband_local_planner and teb_local_planner. They use different algorithms to generate velocity commands. Usually dwa_local_planner is the go-to choice. We will discuss it in detail. More information on other planners will be provided later.

## 3.1 DWA Local Planner

### 3.1.1 DWA algorithm

See next page.

`dwa_local_planner` uses Dynamic Window Approach (DWA) algorithm. ROS Wiki provides a summary of its implementation of this algorithm:

> 1. Discretely sample in the robot's control space *(dx,dy,dtheta)*
> 2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
> 3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
> 4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
> 5. Rinse and repeat.

DWA is proposed by Dieter Fox et. al. in the paper *The Dynamic Window Approach to Collision Avoidance*. According to this paper, the goal of DWA is to produce a $(v, \omega)$ pair which represents a circular trajectory that is optimal for robot's local condition. DWA reaches this goal by searching the velocity space in the next time interval. The velocities in this space are restricted to be admissible, which means the robot must be able to stop before reaching the closest obstacle on the circular trajectory dictated by these admissible velocities. Also, DWA will only consider velocities within a dynamic window, which is defined to be the set of velocity pairs that is reachable within the next time interval given the current translational and rotational velocities and accelerations. DWA maximizes an objective function that depends on (1) the progress to the target, (2) clearance from obstacles, and (3) forward velocity to produce the optimal velocity pair.

Now, let us look at the algorithm summary on ROS Wiki. The first step is to sample velocity pairs $(v_x, v_y, \omega)$ in the velocity space within the dynamic window. The second step is basically obliterating velocities (i.e. kill off bad trajectories) that are not admissible. The third step is to evaluate the velocity pairs using the objective function, which outputs *trajectory score*. The fourth and fifth steps are easy to understand: take the current best velocity option and recompute.

This DWA planner depends on the local costmap which provides obstacle information. Therefore, tuning the parameters for the local costmap is crucial for optimal behavior of DWA local planner. Next, we will look at parameters in forward simulation, trajectory scoring, costmap, and so on.

### 3.1.2 DWA Local Planner : Forward Simulation

Forward simulation is the second step of the DWA algorithm. In this step, the local planner takes the velocity samples in robot's control space, and examine the circular trajectories represented by those velocity samples, and finally eliminate bad velocities (ones whose trajectory intersects with an obstacle). Each velocity sample is simulated as if it is applied to the robot for a set time interval, controlled by `sim_time`($s$) parameter. We can think of `sim_time` as the time allowed for the robot to move with the sampled velocities.

Through experiments, we observed that the longer the value of `sim_time`, the heavier the computation load becomes. Also, when `sim_time` gets longer, the path produced by the local planner is longer as well, which is reasonable. Here are some suggestions on how to tune this `sim_time` parameter.

**How to tune `sim_time`** Setting `sim_time` to a very low value ($<= 2.0$) will result in limited performance, especially when the robot needs to pass a narrow doorway, or gap between furnitures, because there is insufficient time to obtain the optimal trajectory that actually goes through the narrow passway. On the other hand, since with DWA Local Planner, all trajectories are simple arcs, setting the `sim_time` to a very high value ($>= 5.0$) will result in long curves that are not very flexible. This problem is not that unavoidable, because the planner actively replans after each time interval (controlled by `controller_frequency`($Hz$)), which leaves room for small adjustments. A value of 4.0 seconds should be enough even for high performance computers.

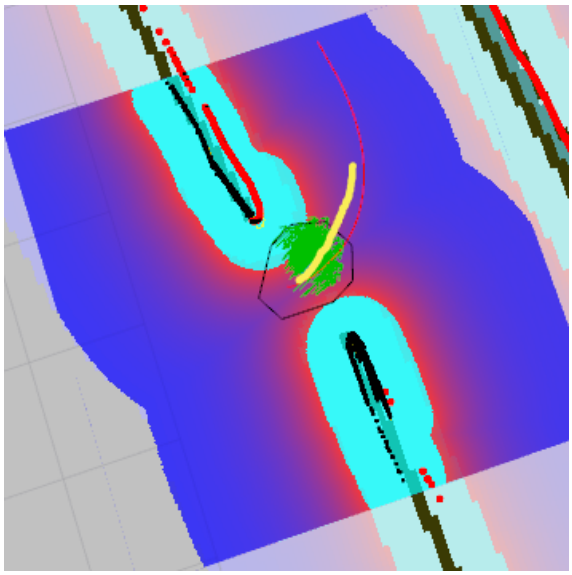

Figure 11: `sim_time` = 1.5

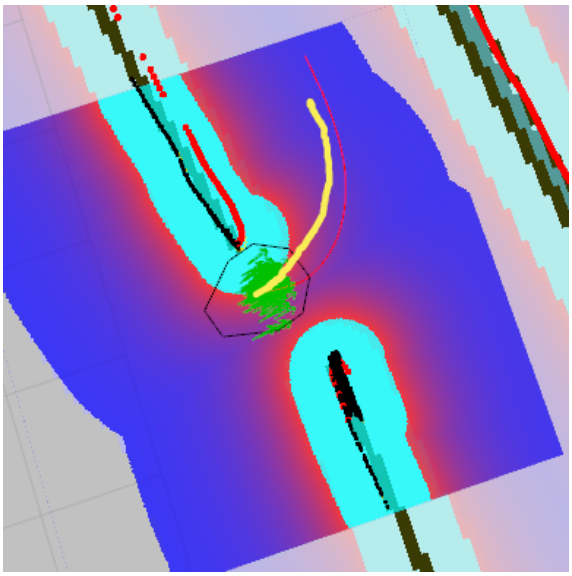

Figure 12: `sim_time` = 4.0

Besides `sim_time`, there are several other parameters that worth our attention.

**Velocity samples** Among other parameters, `vx_sample`, `vy_sample` determine how many translational velocity samples to take in x, y direction. `vth_sample` controls the number of rotational velocities samples. The number of samples you would like to take depends on how much computation power you have. In most cases we prefer to set `vth_samples` to be higher than translational velocity samples, because turning is generally a more complicated condition than moving straight ahead. If you set `max_vel_y` to be zero, there is no need to have velocity samples in y direction since there will be no usable samples. We picked `vx_sample` = 20, and `vth_samples` = 40.

**Simulation granularity** `sim_granularity` is the step size to take between points on a trajectory. It basically means how frequent should the points on this trajectory be examined (test if they intersect with any obstacle or not). A lower value means higher frequency, which requires more computation power. The default value of 0.025 is generally enough for turtlebot-sized mobile base.

### 3.1.3 DWA Local Planner : Trajactory Scoring

As we mentioned above, DWA Local Planner maximizes an objective function to obtain optimal velocity pairs. In its paper, the value of this objective function relies on three components: progress to goal, clearance from obstacles and forward velocity. In ROS's implementation, the cost of the objective function is calculated like this:

$$
\begin{aligned}
cost \;=\; & \texttt{path\_distance\_bias} * (\text{distance}(m) \text{ to path from the endpoint of the trajectory}) \\
+\; & \texttt{goal\_distance\_bias} * (\text{distance}(m) \text{ to local goal from the endpoint of the trajectory}) \\
+\; & \texttt{occdist\_scale} * (\text{maximum obstacle cost along the trajectory in obstacle cost (0-254)})
\end{aligned}
$$

The objective is to get the lowest cost. `path_distance_bias` is the weight for how much the local planner should stay close to the global path [3]. A high value will make the local planner prefer trajectories on global path. `goal_distance_bias` is the weight for how much the robot should attempt to reach the local goal, with whatever path. Experiments show that increasing `goal_distance_bias` enables the robot to be less attached to the global path. `occdist_scale` is the weight for how much the robot should attempt to avoid obstacles. A high value for this parameter results in indecisive robot that stucks in place. Currently for SCITOS G5, we set `path_distance_bias` to 32.0, `goal_distance_bias` to 20.0, `occdist_scale` to 0.02. They work well in simulation.

[3] from Robot Operating System (ROS): The Complete Reference, Volume 1, p.91

#### 3.1.4 DWA Local Planner : Other Parameters

**Goal distance tolerance** These parameters are straightforward to understand. Here we will list their description shown on ROS Wiki:

- `yaw_goal_tolerance` (double, default: 0.05) The tolerance in radians for the controller in yaw/rotation when achieving its goal.
- `xy_goal_tolerance` (double, default: 0.10) The tolerance in meters for the controller in the x & y distance when achieving a goal.
- `latch_xy_goal_tolerance` (bool, default: false) If goal tolerance is latched, if the robot ever reaches the goal xy location it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so.

**Oscilation reset** In situations such as passing a doorway, the robot may oscilate back and forth because its local planner is producing paths leading to two opposite directions. If the robot keeps oscilating, the navigation stack will let the robot try its recovery behaviors.

- `oscillation_reset_dist` (double, default: 0.05) How far the robot must travel in meters before oscillation flags are reset.

# 4 Costmap Parameters

As mentioned above, *costmap* parameters tuning is essential for the success of local planners (not only for DWA). In ROS, costmap is composed of static map layer, obstacle map layer and inflation layer. Static map layer directly interprets the given static SLAM map provided to the navigation stack. Obstacle map layer includes 2D obstacles and 3D obstacles (voxel layer). Inflation layer is where obstacles are inflated to calculate cost for each 2D costmap cell.

Besides, there is a *global costmap*, as well as a *local costmap*. Global costmap is generated by inflating the obstacles on the map provided to the navigation stack. Local costmap is generated by inflating obstacles detected by the robot's sensors in real time.

There are a number of important parameters that should be set as good as possible.

## 4.1 footprint

Footprint is the contour of the mobile base. In ROS, it is represented by a two dimensional array of the form $[x_0, y_0], [x_1, y_1], [x_2, y_2], ...]$, no need to repeat the first coordinate. This footprint will be used to compute the radius of inscribed circle and

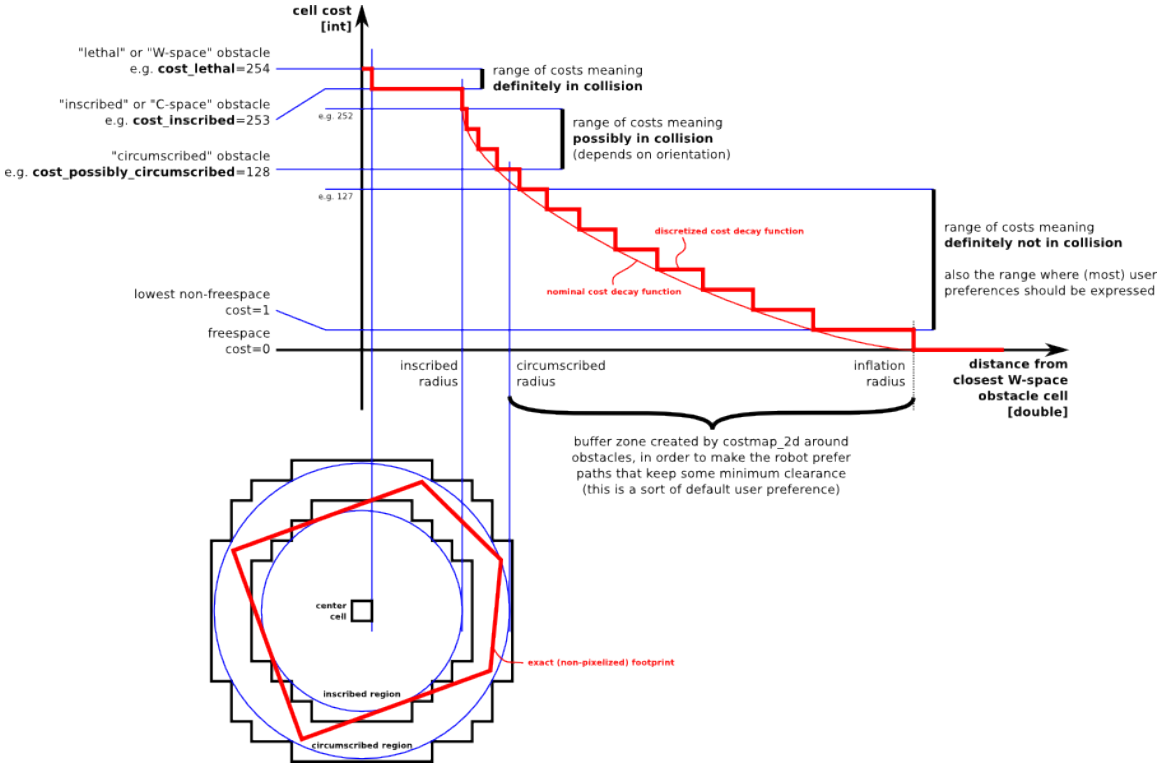


Figure 13: inflation decay

circumscribed circle, which are used to inflate obstacles in a way that fits this robot. Usually for safety, we want to have the footprint to be slightly larger than the robot's real contour.

To determine the footprint of a robot, the most straightforward way is to refer to the drawings of your robot. Besides, you can manually take a picture of the top view of its base. Then use CAD software (such as Solidworks) to scale the image appropriately and move your mouse around the contour of the base and read its coordinate. The origin of the coordinates should be the center of the robot. Or, you can move your robot on a piece of large paper, then draw the contour of the base. Then pick some vertices and use rulers to figure out their coordinates.

## 4.2 inflation

Inflation layer is consisted of cells with cost ranging from 0 to 255. Each cell is either occupied, free of obstacles, or unknown. Figure 13 shows a diagram [4] illustrating how inflation decay curve is computed.

`inflation_radius` and `cost_scaling_factor` are the parameters that determine the inflation. `inflation_radius` controls how far away the zero cost point is from

[4]Diagram is from http://wiki.ros.org/costmap_2d

the obstacle. `cost_scaling_factor` is inversely proportional to the cost of a cell. Setting it higher will make the decay curve more steep.

Dr. Pronobis sugggests the optimal costmap decay curve is one that has relatively low slope, so that the best path is as far as possible from the obstacles on each side. The advantage is that the robot would prefer to move in the middle of obstacles. As shown in Figure 8 and 9, with the same starting point and goal, when costmap curve is steep, the robot tends to be close to obstacles. In Figure 14, `inflation_radius` = 0.55, `cost_scaling_factor` = 5.0; In Figure 15, `inflation_radius` = 1.75, `cost_scaling_factor` = 2.58

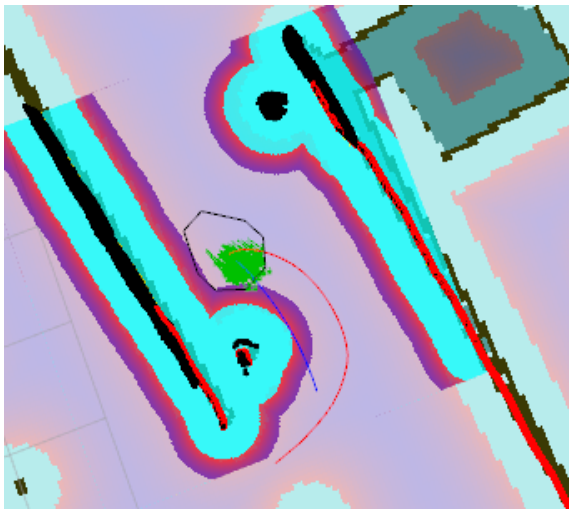

Figure 14: steep inflation curve

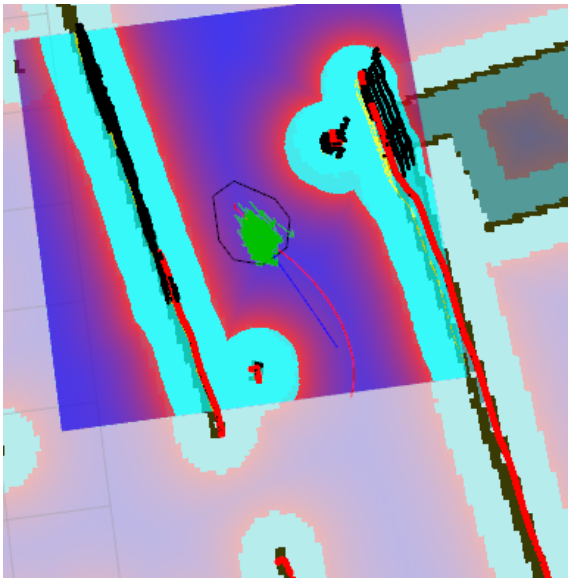

Figure 15: gentle inflation curve

Based on the decay curve diagram, we want to set these two parameters such that the inflation radius almost covers the corriders, and the decay of cost value is moderate, which means decrease the value of `cost_scaling_factor` .

### 4.3 costmap resolution

This parameter can be set separately for local costmap and global costmap. They affect computation load and path planning. With low resolution (>= 0.05), in narrow passways, the obstacle region may overlap and thus the local planner will not be able to find a path through.

For global costmap resolution, it is enough to keep it the same as the resolution of the map provided to navigation stack. If you have more than enough computation power, you should take a look at the resolution of your laser scanner, because when creating the map using gmapping, if the laser scanner has lower resolution than your desired map resolution, there will be a lot of small "unknown dots" because the laser scanner cannot cover that area, as in Figure 10.

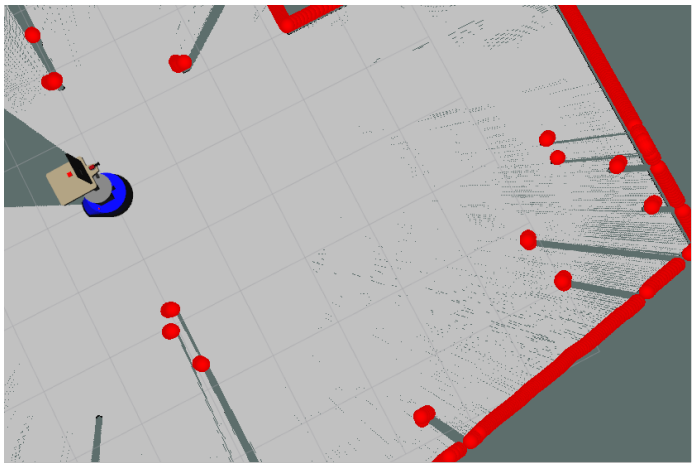

Figure 16: gmapping resolution = 0.01. Notice the unknown dots on the right side of the image

For example, Hokuyo URG-04LX-UG01 laser scanner has metric resolution of 0.01mm[5]. Therefore, scanning a map with resolution <= 0.01 will require the robot to rotate several times in order to clear unknown dots. We found 0.02 to be a sufficient resolution to use.

### 4.4 obstacle layer and voxel layer

These two layers are responsible for marking obstacles on the costmap. They can be called altogether as *obstacle layer*. According to ROS wiki, the obstacle layer tracks in two dimensions, whereas the voxel layer tracks in three. Obstacles are marked (detected) or cleared (removed) based on data from robot's sensors, which has topics for costmap to subscribe to.

In ROS implementation, the voxel layer inherits from obstacle layer, and they both obtain obstacles information by interpreting laser scans or data sent with `PointCloud` or `PointCloud2` type messages. Besides, the voxel layer requires depth sensors such as Microsoft Kinect or ASUS Xtion. 3D obstacles are eventually projected down to the 2D costmap for inflation.

**How voxel layer works** Voxels are 3D volumetric cubes (think 3D pixel) which has certain relative position in space. It can be used to be associated with data or properties of the volume near it, e.g. whether its location is an obstacle. There has been quite a few research around online 3D reconstruction with the depth cameras via voxels. Here are some of them.

[5]data from https://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG-04LX_UG01_spec_en.pdf

- KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera
- Real-time 3D Reconstruction at Scale using Voxel Hashing

`voxel_grid` is a ROS package which provides an implementation of efficient 3D voxel grid data structure that stores voxels with three states: marked, free, unknown. The *voxel grid* occupies the volume within the costmap region. During each update of the voxel layer's boundary, the voxel layer will mark or remove some of the voxels in the voxel grid based on observations from sensors. It also performs ray tracing, which is discussed next. Note that the voxel grid is not recreated when updating, but only updated unless the size of local costmap is changed.

**Why ray tracing** in obstacle layer and voxel layer? Ray tracing is best known for rendering realistic 3D graphics, so it might be confusing why it is used in dealing with obstacles. One big reason is that obstacles of different type can be detected by robot's sensors. Take a look at figure 17. In theory, we are also able to know if an obstacle is rigid or soft (e.g. grass)[6].

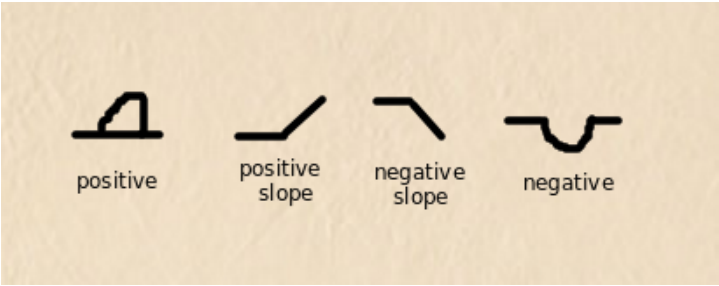


Figure 17: With ray tracing, laser scanners is able to recognize different types of obstacles.

A good blog on voxel ray tracing versus polygong ray tracing: `http://raytracey.blogspot.com/2008/08/voxel-ray-tracing-vs-polygon-ray.html`

With the above understanding, let us look into the parameters for the obstacle layer[7]. These parameters are global filtering parameters that apply to all sensors.

- `max_obstacle_height`: The maximum height of any obstacle to be inserted into the costmap in meters. This parameter should be set to be slightly higher than the height of your robot. For voxel layer, this is basically the height of the voxel grid.

[6] mentioned in *Using Robots in Hazardous Environments* by Boudoin, Habib, pp.370
[7] Some explanations are directly copied from costmap2d ROS Wiki

- `obstacle_range`: The default maximum distance from the robot at which an obstacle will be inserted into the cost map in meters. This can be over-ridden on a per-sensor basis.

- `raytrace_range`: The default range in meters at which to raytrace out obstacles from the map using sensor data. This can be over-ridden on a per-sensor basis.

These parameters are only used for the voxel layer (VoxelCostmapPlugin).

- `origin_z`: The z origin of the map in meters.

- `z_resolution`: The z resolution of the map in meters/cell.

- `z_voxels`: The number of voxels to in each vertical column, the height of the grid is z_resolution * z_voxels.

- `unknown_threshold`: The number of unknown cells allowed in a column considered to be "known"

- `mark_threshold`: The maximum number of marked cells allowed in a column considered to be "free".

**Experiment observations** Experiments further clarify the effects of the voxel layer's parameters. We use ASUS Xtion Pro as our depth sensor. We found that position of Xtion matters in that it determines the range of "blind field", which is the region that the depth sensor cannot see anything.

In addition, voxels representing obstacles only update (marked or cleared) when obstacles appear within Xtion range. Otherwise, some voxel information will remain, and their influence on costmap inflation remains.

Besides, `z_resolution` controls how dense the voxels is on the $z$-axis. If it is higher, the voxel layers are denser. If the value is too low (e.g. 0.01), all the voxels will be put together and thus you won't get useful costmap information. If you set z_resolution to a higher value, your intention should be to obtain obstacles better, therefore you need to increase `z_voxels` parameter which controls how many voxels in each vertical column. It is also useless if you have too many voxels in a column but not enough resolution, because each vertical column has a limit in height. Figure 18-20 shows comparison between different voxel layer parameters setting.

Figure 18: Scene: Plant in front of the robot

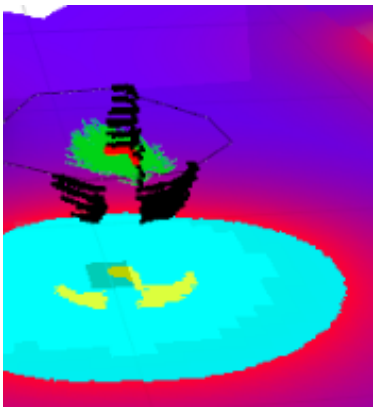

Figure 19: high `z_resolution`

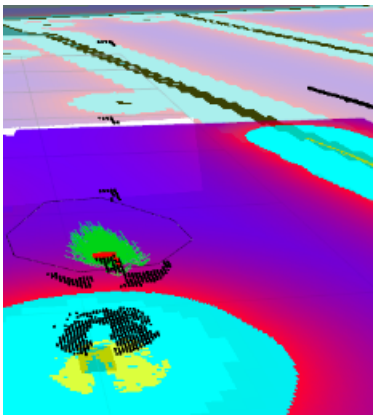

Figure 20: low `z_resolution`

# 5 AMCL

`amcl` is a ROS package that deals with robot localization. It is the abbreviation of Adaptive Monte Carlo Localization, also known as partical filter localization. This localization technique works like this: Each sample stores a position and orientation data representing the robot's pose. Particles are all sampled randomly initially. When the robot moves, particles are resampled based on their current state as well as robot's action using recursive Bayesian estimation.

More discussion on AMCL parameter tuning will be provided later. Please refer to `http://wiki.ros.org/amcl` for more information. For the details of the original algorithm Monte Carlo Localization, read Chapter 8 of *Probabilistic Robotics*, by Thrun, Burgard, and Fox.

# 6 Recovery Behaviors

An annoying thing about robot navigation is that the robot may get stuck. Fortunately, the navigation stack has recovery behaviors built-in. Even so, sometimes the robot will exhaust all available recovery behaviors and stay still. Therefore, we may need to figure out a more robust solution.

**Types of recovery behaviors** ROS navigation has two recovery behaviors. They are `clear_costmap_recovery` and `rotate_recovery`. Clear costmap recovery is basically reverting the local costmap to have the same state as the global costmap. Rotate recovery is to recover by rotating 360 degrees in place.

**Unstuck the robot** Sometimes rotate recovery will fail to execute due to rotation failure. At this point, the robot may just give up because it has tried all of its

recovery behaviors - clear costmap and rotation. In most experiments we observed that when the robot gives up, there are actually many ways to unstuck the robot. To avoid giving up, we used SMACH to continuously try different recovery behaviors, with additional ones such as setting a temporary goal that is very close to the robot, and returning to some previously visited pose (i.e. backing off). These methods turn out to improve the robot's durability substantially, and unstuck it from previously hopeless tight spaces from our experiment observations[8].

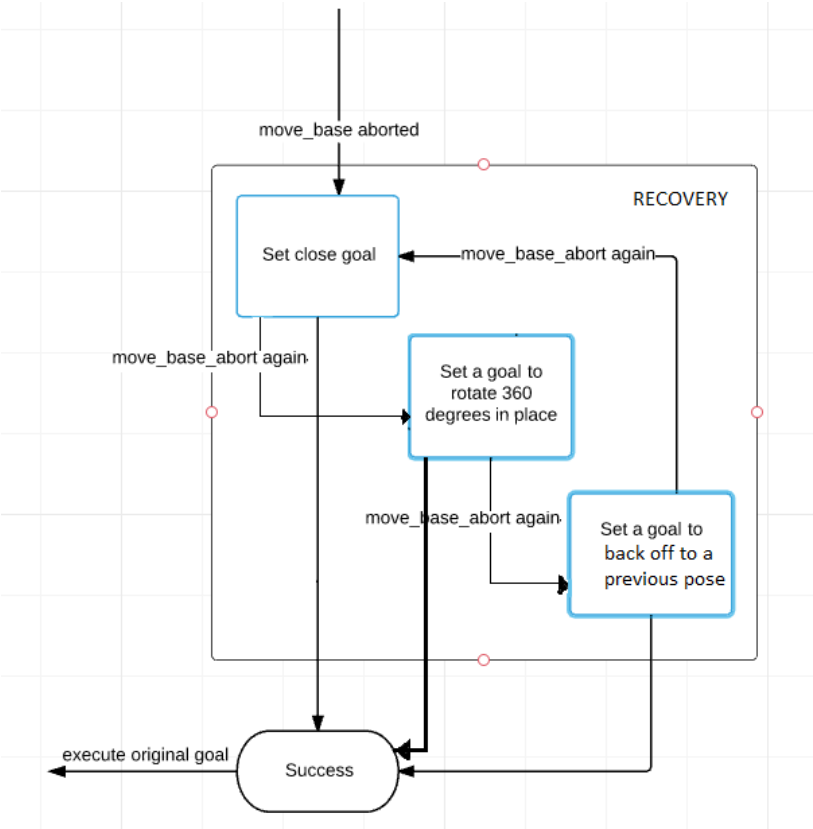


Figure 21: Simple recovery state in SMACH

**Parameters** The parameters for ROS's recovery behavior can be left as default in general. For clear costmap recovery, if you have a relatively high `sim_time`, which means the trajectory is long, you may want to consider increasing `reset_distance` parameter, so that bigger area on local costmap is removed, and there is a better chance for the local planner to find a path.

[8]Here is a video demo of my work on mobile robot navigation: https://youtu.be/1-7GNtR6gVk

# 7 Dynamic Reconfigure

One of the most flexible aspect about ROS navigation is dynamic reconfiguration, since different parameter setup might be more helpful for certain situations, e.g. when robot is close to the goal. Yet, it is not necessary to do a lot of dynamic reconfiguration.

**Example** One situation that we observed in our experiments is that the robot tends to fall off the global path, even when it is not necessary or bad to do so. Therefore we increased `path_distance_bias`. Since a high `path_distance_bias` will make the robot stick to the global path, which does not actually lead to the final goal due to tolerance, we need a way to let the robot reach the goal with no hesitation. We chose to dynamically decrease the `path_distance_bias` so that `goal_distance_bias` is emphasized when the robot is close to the goal. After all, doing more experiments is the ultimate way to find problems and figure out solutions.

# 8 Problems

1. Getting stuck

   This is a problem that we face a lot when using ROS navigation. In both simulation and reality, the robot gets stuck and gives up the goal.

2. Different speed in different directions

   We observed some weird behavior of the navigation stack. When the goal is set in the -x direction with respect to TF origin, dwa local planner plans less stably (the local planned trajectory jumps around) and the robot moves really slowly. But when the goal is set in the +x direction, dwa local planner is much more stable, and the robot can move faster.

   I reported this issue on Github here: https://github.com/ros-planning/navigation/issues/503. Nobody attempted to resolve it yet.

3. Reality VS. simulation

   There is a difference between reality and simulation. In reality, there are more obstacles with various shapes. For exmaple, in the lab there is a vertical stick that is used to hold to door open. Because it is too thin, the robot sometimes fails to detect it and hit on it. There are also more complicated human activity in reality.

4. Inconsistency

   Robots using ROS navigation stack can exhibit inconsistent behaviors, for example when entering a door, the local costmap is generated again and again

with slight difference each time, and this may affect path planning, especially when resolution is low. Also, there is no memory for the robot. It does not remember how it entered the room from the door the last time. So it needs to start out fresh again every time it tries to enter a door. Thus, if it enters the door in a different angle than before, it may just get stuck and give up.

# Thanks

Hope this guide is helpful. There will be more information added for local planners besides DWA Local planner, as well as for AMCL parameter tuning. Please feel free to add more.