Slide

https://kasets.art/IGH2BV

Planning and Navigation

Noppanut Thongton

Path Planning

Path planning

Competences for Navigation: Planning and Reacting

In the artificial intelligence community **planning** and **reacting** are often viewed as contrary approaches or even opposites. In fact, when applied to **physical systems** such as mobile robots, planning and reacting have a strong complementarity, each being critical to the other's success. The navigation challenge for a robot involves **executing a course of action** (or plan) to reach its goal position. During execution, the robot must **react to unforeseen events** (e.g., obstacles) in such a way as to still reach the goal. Without reacting, the planning effort will not pay off because the robot will never physically reach its goal. Without planning, the reacting effort cannot guide the overall robot behavior to reach a distant goal – again, the robot will **never reach its goal**.

Competences for Navigation: Planning and Reacting

An information-theoretic formulation of the navigation problem will make this complementarity clear. Suppose that a robot M at time i has a map M_i and an initial belief state b_i. The robot's goal is to reach a position p while satisfying some **temporal constraints**:

$$loc_g(R) = p \,; (g \ \leq n).$$

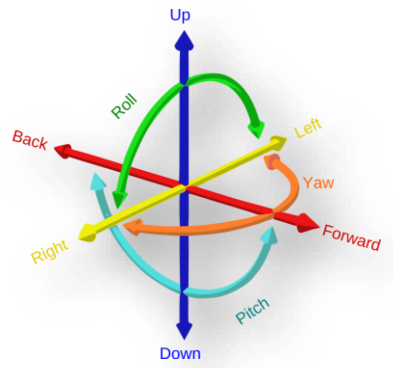
Thus, the robot must be at location p **at or before** timestep n.

Competences for Navigation: Planning and Reacting

Although the goal of the robot is **distinctly physical**, the robot can **only really sense its belief state**, *not its physical location*, and therefore we map the goal of reaching location p to reaching a belief state b_g, corresponding to the belief that $loc_g(R) = p$. With this formulation, a plan q is nothing more than one or more trajectories from b_i to b_g. In other words, plan q will cause the robot's belief state to transition from b_i to b_g if the plan is executed from a world state consistent with both b_i and M_i.

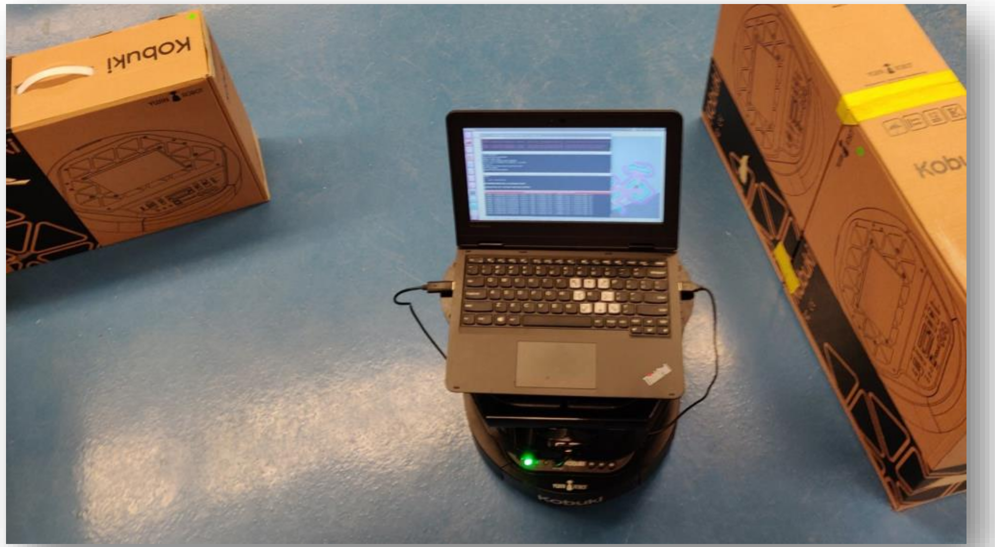
Of course, the problem is that the latter condition may not be met. It is entirely possible that the robot's position is not quite consistent with b_i, and it is even likelier that M_i is either incomplete or incorrect. Furthermore, the real-world environment is dynamic. Even if M_i is correct as a single snapshot in time, the planner's model regarding how M **changes over time** is usually **<u>imperfect</u>**.

Path planning



Even before the advent of affordable mobile robots, the field of path-planning was heavily studied because of its applications in the area of industrial **manipulator robotics**. Interestingly, the path planning problem for a manipulator with, for instance, six degrees of freedom is **far more complex** than that of a differential-drive robot operating in a flat environment.

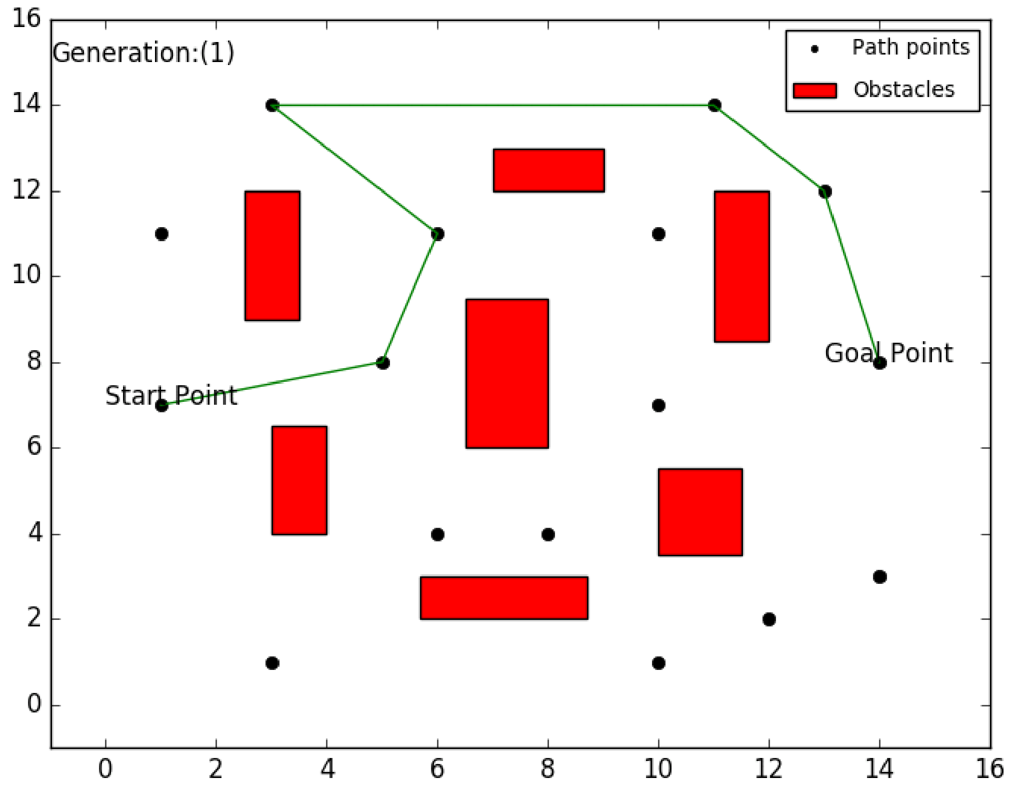
Path planning



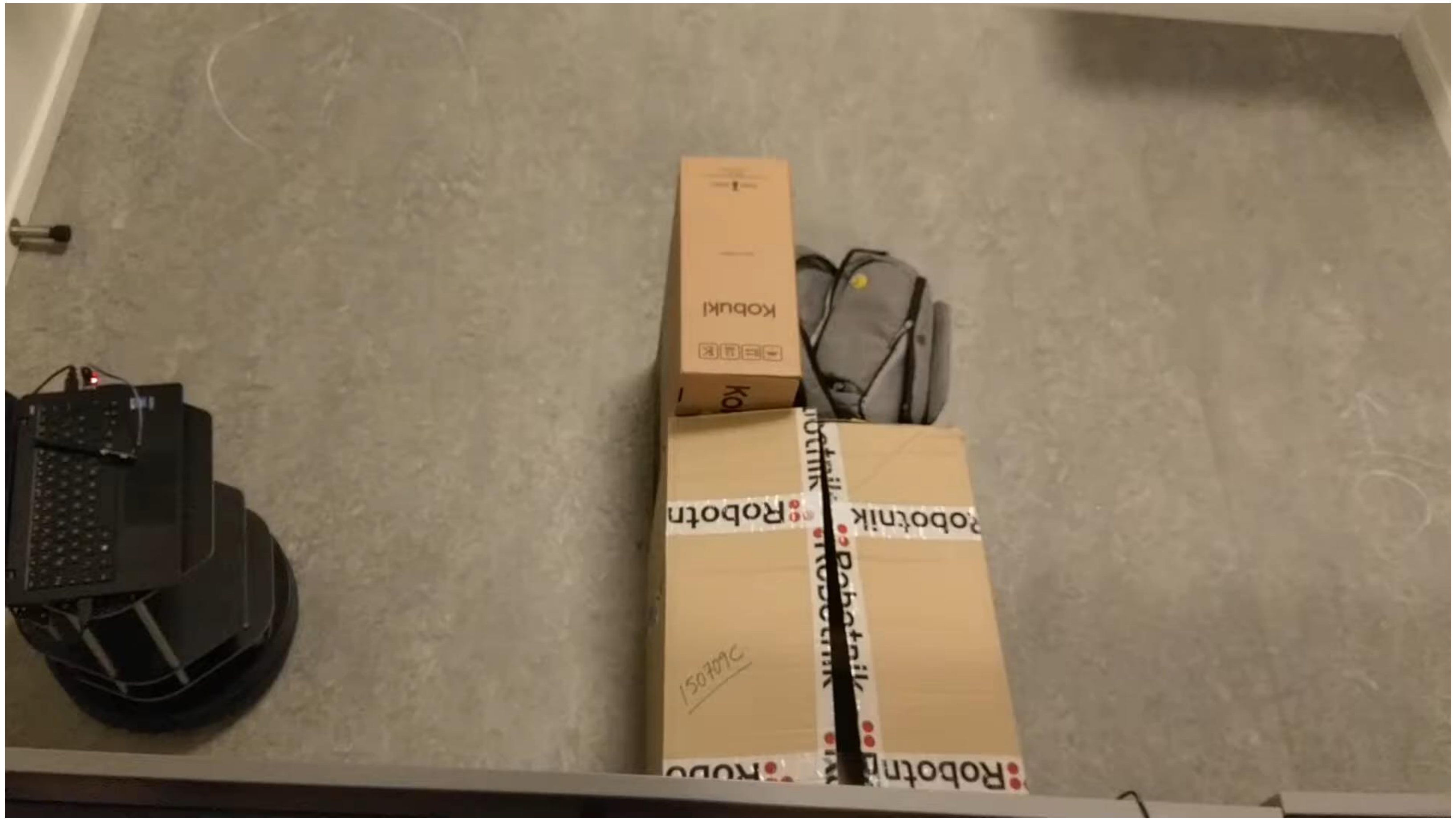
Path planning

Therefore, although we can take inspiration from the techniques invented for manipulation, the path-planning algorithms used by mobile robots tend to be **simpler approximations** owing to the greatly **reduced** degrees of freedom. Furthermore, industrial robots often operate at the fastest possible speed because of the economic impact of high throughput on a factory line. So, the dynamics and not just the kinematics of their motions are **significant**, further complicating path planning and execution. In contrast, several mobile robots operate at such **low speeds** that dynamics are rarely considered during path planning, further simplifying the mobile robot instantiation of the problem.

Path planning



Path planning



Path planning

Path planning

Configuration space

Path planning for manipulator robots and, indeed, even for most mobile robots, is formally done in a representation called ***configuration space***. Suppose that a robot arm (e.g., SCARA robot) has k ***degrees of freedom***. Every state or configuration of the robot can be described with **real values**: ,$q_1, \ldots, q_k$,. The k-values can be regarded as a point p in a k-dimensional space called the **configuration space** C of the robot. This description is convenient because it allows us to describe the **complex 3D shape** of the robot with a single k-dimensional point.

Path planning

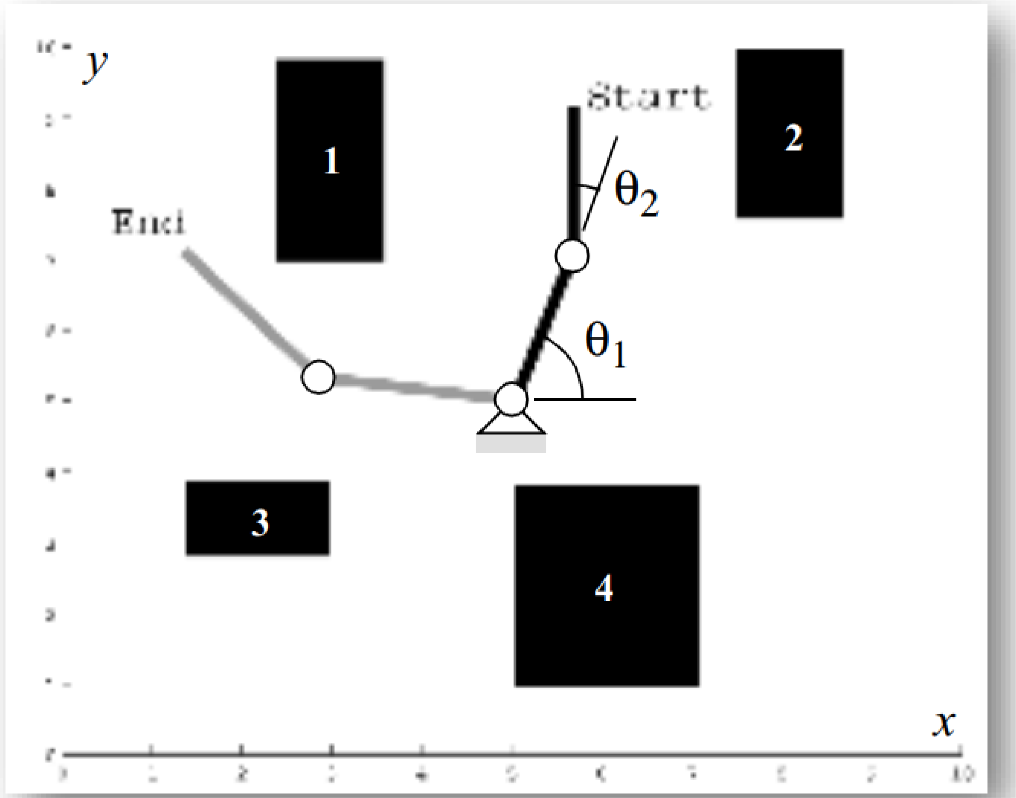
Configuration space

Now consider the robot arm moving in an environment where the **workspace** (i.e., its physical space) contains **known obstacles**. The goal of path planning is to find a path in the physical space from the *initial position* of the arm to the *goal position*, **avoiding** all collisions with the obstacles. This is a difficult problem to visualize and solve in the physical space, particularly as grows large. But in configuration space the problem is straightforward. If we define the configuration space obstacle as the subspace of where the robot arm bumps into something, we can compute the free space in which the robot can move **safely**.

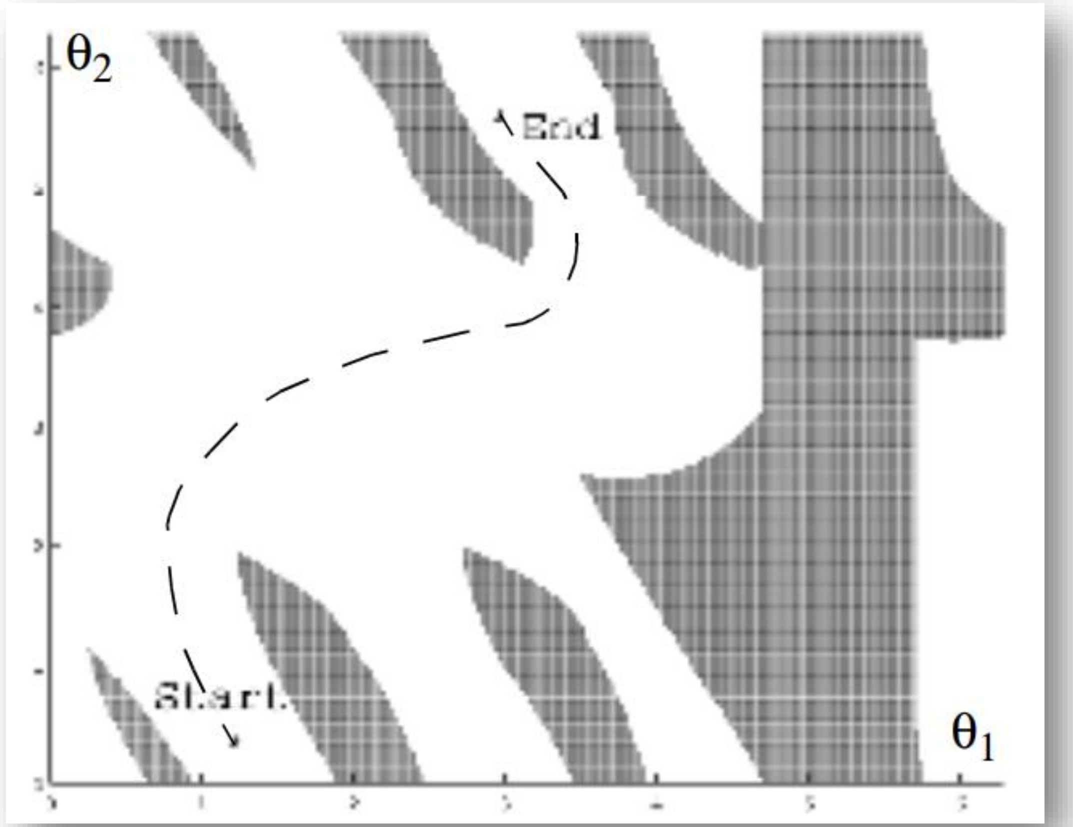
Path planning

Configuration space

Physical space

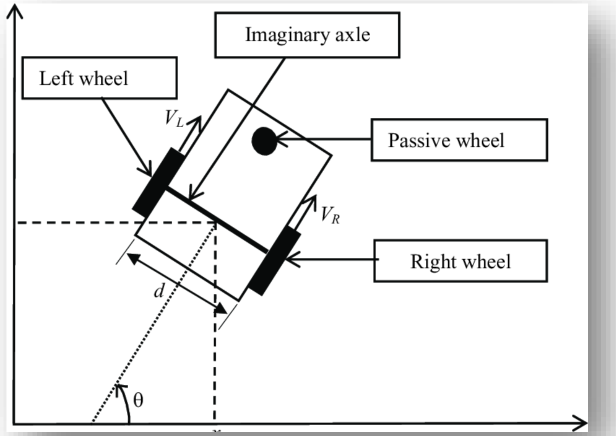


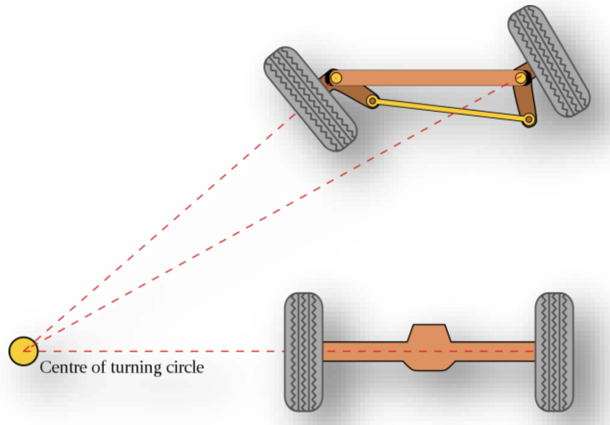
Configuration space



robotcitizens.org

Path planning



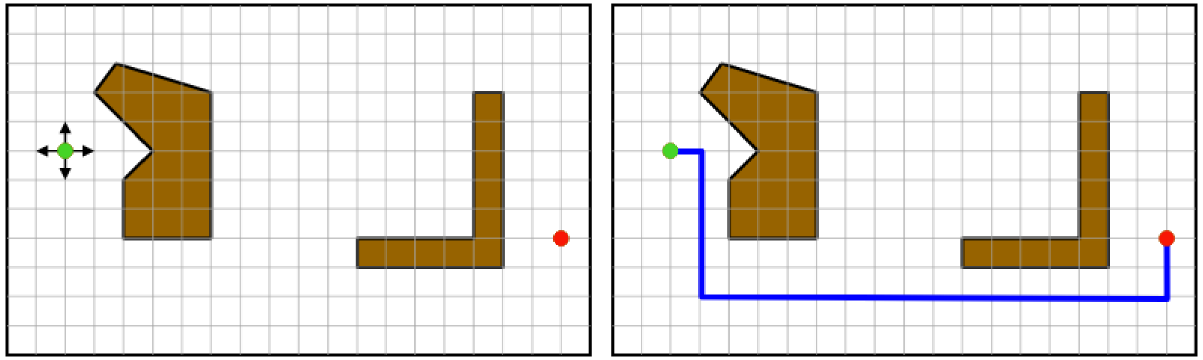


Configuration space

For mobile robots operating on **flat ground**, we generally represent robot position with **three variables** (x, y, θ). But, as we have seen, most robots are **nonholonomic**, using **differential-drive** systems or **Ackerman steered** systems. For such robots, the nonholonomic constraints **limit the robot's velocity** $(\dot{x},\ \dot{y},\ \dot{\theta})$ in each configuration (x, y, θ).

In mobile robotics, the most common approach is to assume for path-planning purposes that the robot is in fact holonomic, simplifying the process tremendously. This is especially common for differential-drive robots because they can rotate in place and so a holonomic path can be easily mimicked if the ***rotational position of the robot is not critical***.

Path planning



Configuration space

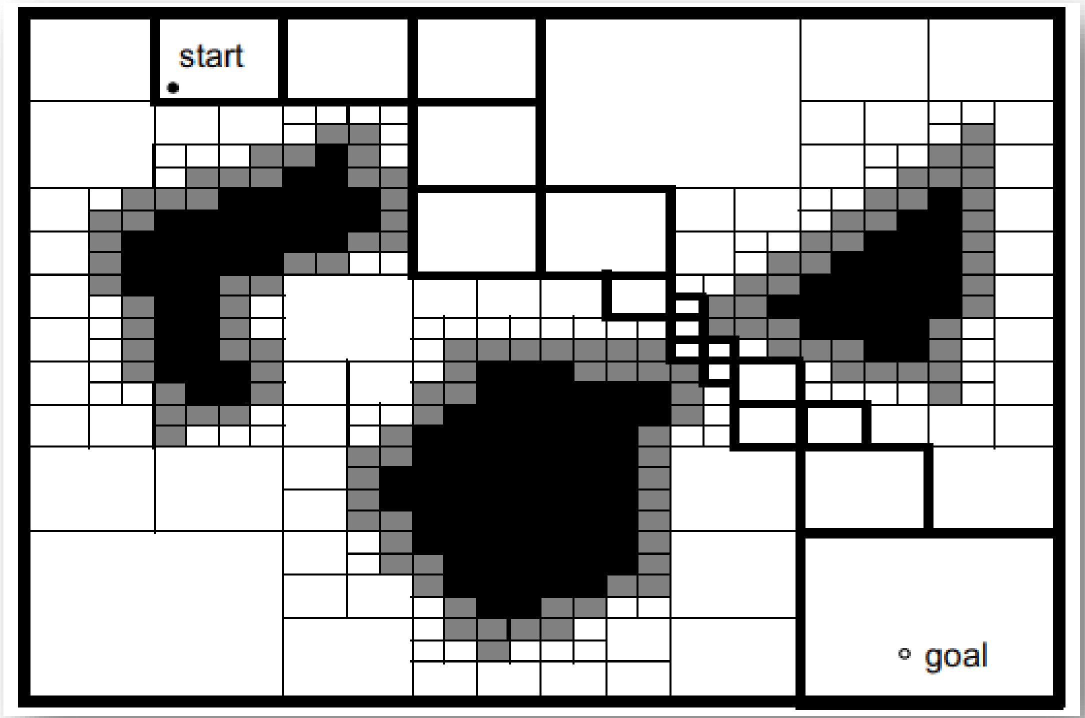
Furthermore, mobile roboticists will often plan under the further assumption that the robot is **simply a point**. Thus, we can further **reduce the configuration space** for mobile robot path planning to a **<u>2D representation</u>** with just ***x* and *y* axes**. The result of all this simplification is that the configuration space looks essentially identical to a 2D (i.e., flat) version of the physical space, with one important difference. Because we have reduced the robot to a **<u>point</u>**, we must inflate each obstacle by the size of the robot's radius to compensate.

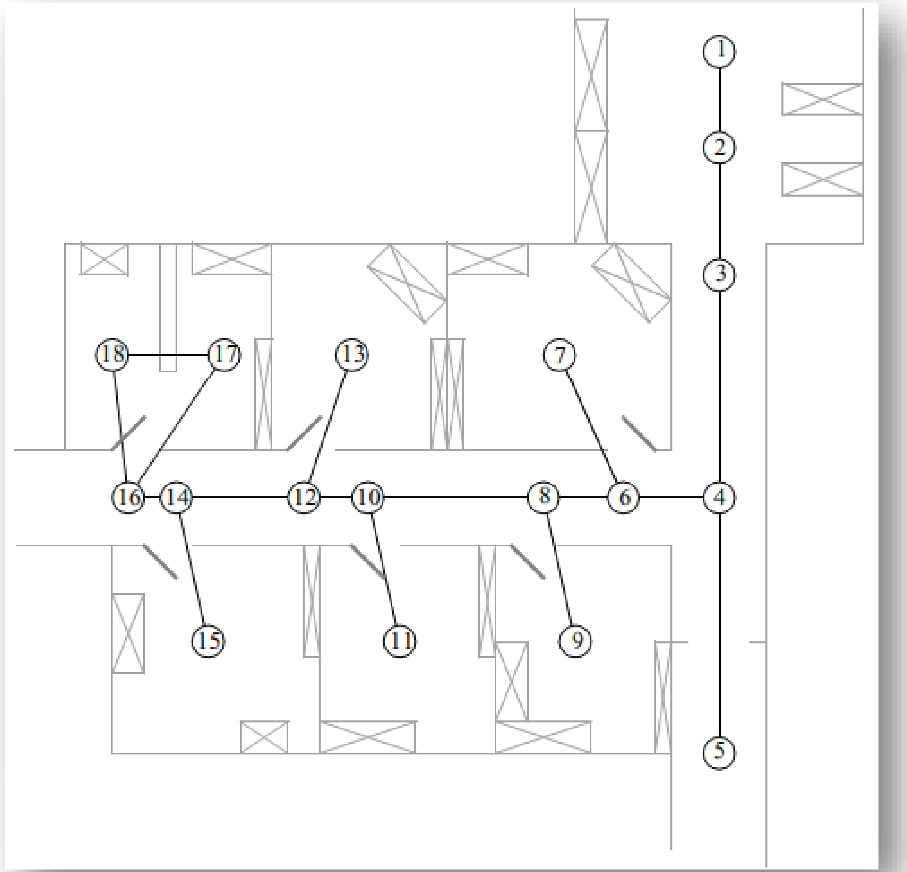
Path planning

Path-planning overview

The robot’s environment representation can range from a continuous geometric description to a **decomposition-based** geometric map or even a **topological map**. The first step of any path-planning system is to transform this possibly continuous environmental model into a **discrete map** suitable for the chosen path-planning algorithm.

Path planning





Path planning

Path-planning overview

Path planners differ as to how they effect this **discrete decomposition**. We can identify three general strategies for decomposition:

1. **Road map**: *identify a set of routes within the free space*.
2. **Cell decomposition**: *discriminate between free and occupied cells*.
3. **Potential field**: *impose a mathematical function over the space*.

Path planning

Path-planning overview

Road map path planning

Road map approaches capture the connectivity of the robot's **free space** in a network of 1D curves or lines, called **road maps**. Once a road map is constructed, it is used as a network of road (path) segments for robot motion planning. Path planning is thus reduced to connecting the initial and goal positions of the robot to the road network, then searching for a series of roads from the initial robot position to its goal position.

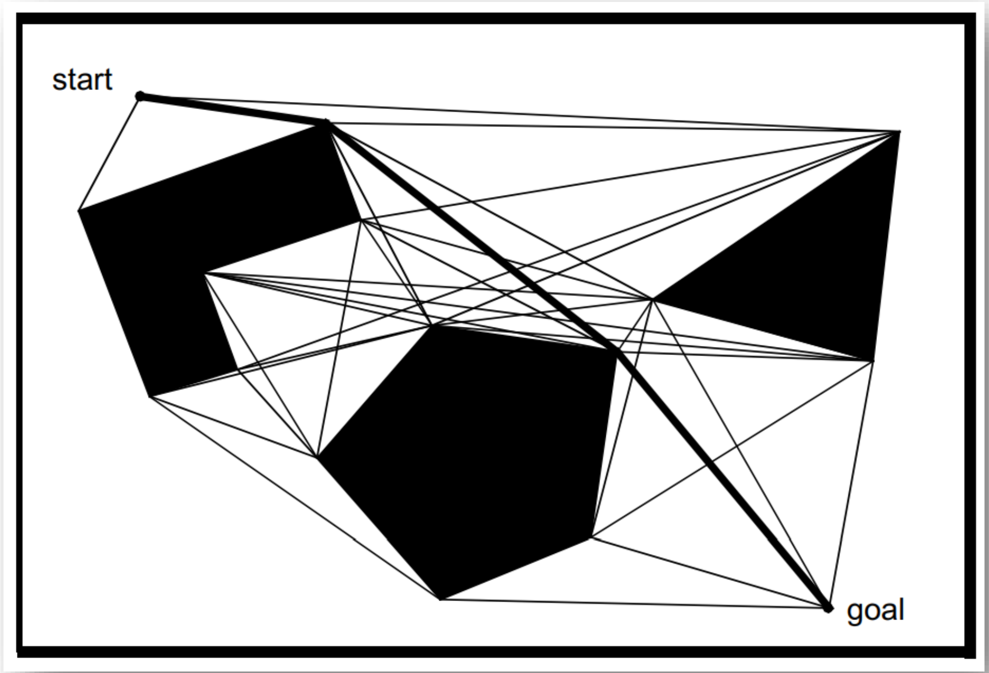
The road map is a **decomposition of the robot's configuration space** based specifically on **obstacle geometry**. The challenge is to construct a set of roads that together enable the robot to go anywhere in its free space, while minimizing the number of total roads. Generally, completeness is preserved in such decompositions as long as the true degrees of freedom of the robot have been captured with appropriate fidelity. We describe two road map approaches below that achieve this result with dramatically different types of roads. In the case of the **visibility graph**, roads come as close as possible to obstacles and resulting paths are minimum-length solutions. In the case of *the Voronoi diagram*, roads stay as far away as possible from obstacles.

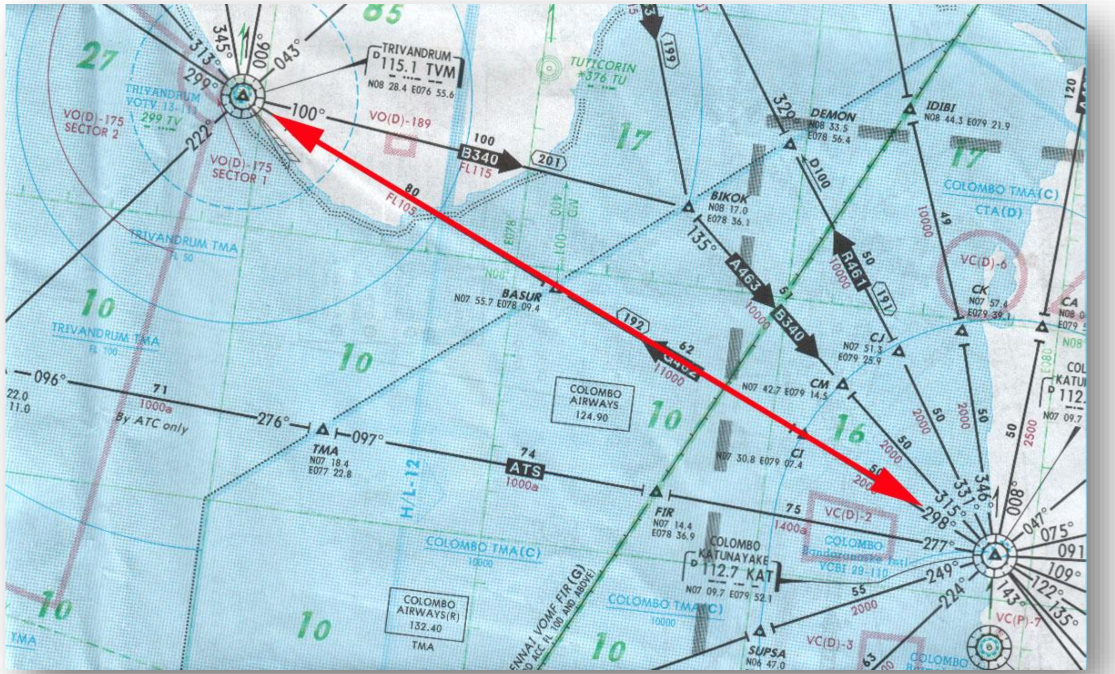
Path planning

Visibility graph

The visibility graph for a **polygonal configuration space** consists of edges joining all pairs of vertices that can see each other (including both the initial and goal positions as vertices as well). The **unobstructed straight lines** (roads) joining those vertices are obviously the shortest distances between them. The task of the path planner is thus to find the shortest path from the initial position to the goal position along the roads defined by the visibility graph.

Path planning

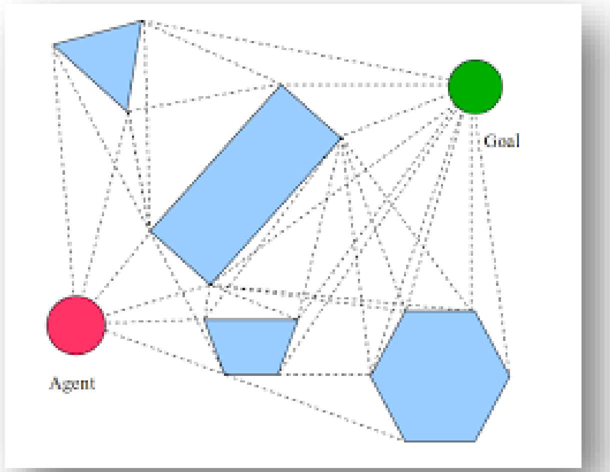




Path planning

Visibility graph

Visibility graph path planning is moderately popular in mobile robotics, partly because implementation is **quite simple**. Particularly when the environmental representation describes objects in the environment as polygons in either continuous or discrete space, the visibility graph search can employ the obstacle polygon descriptions readily.



Path planning

Visibility graph

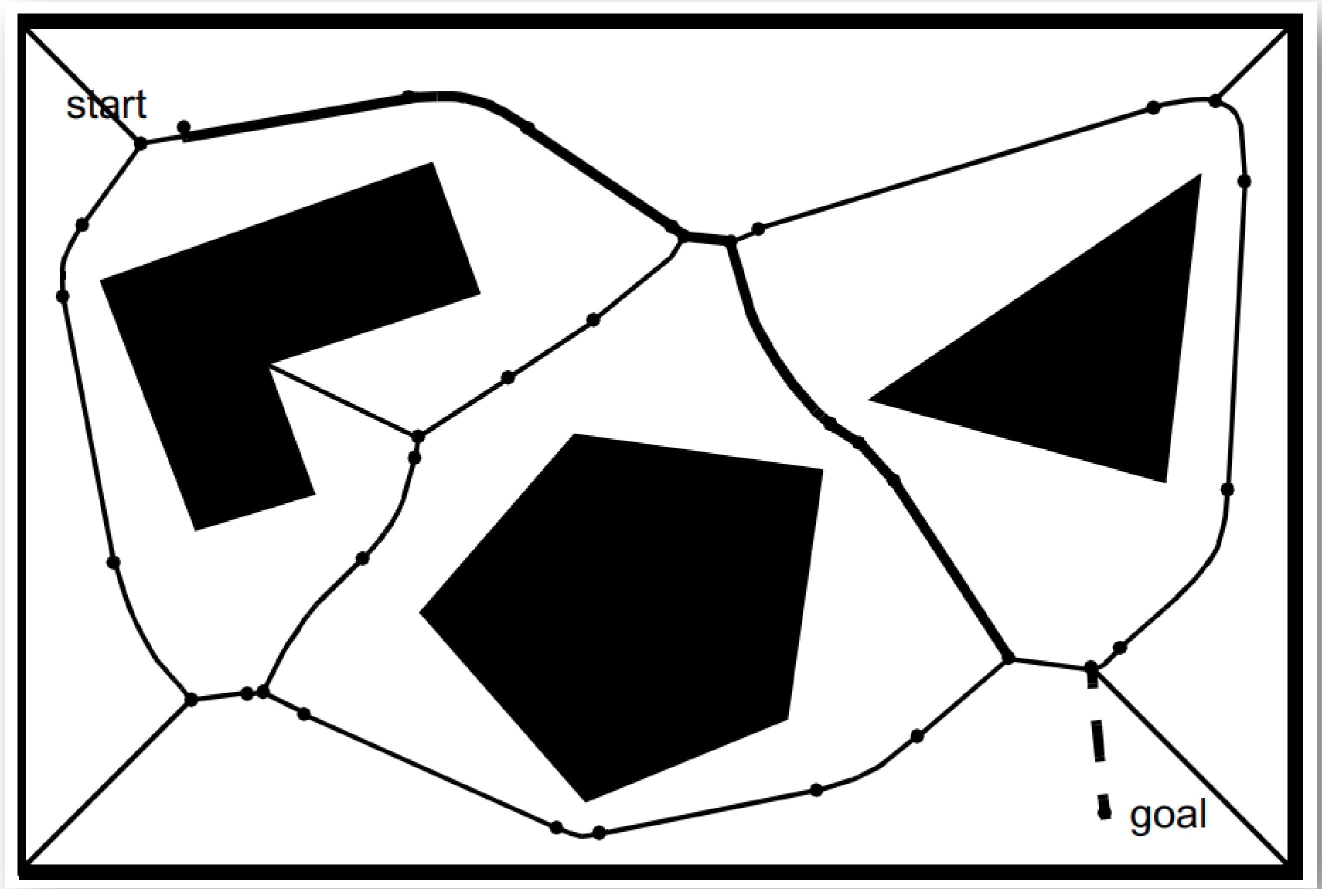
Two important caveats when employing **visibility graph search**. First, **the size of the representation** and **the number of edges and nodes** increase with the number of obstacle polygons. Therefore, the method is extremely fast and efficient in sparse environments but can be slow and inefficient compared to other techniques when used in densely populated environments. The second caveat is a much more serious potential flaw: **the solution paths** found by visibility graph planning tend to take the robot as close as possible to obstacles on the way to the goal. More formally, we can prove that visibility graph planning is optimal in terms of the length of the solution path. This powerful result also means that all sense of safety, in terms of staying a reasonable distance from obstacles, is sacrificed for this optimality. The common solution is to grow obstacles by significantly more than the robot's radius, or, alternatively, to modify the solution path after path planning to distance the path from obstacles when possible. Of course, such actions sacrifice the optimal-length results of visibility graph path planning.

Path planning

Voronoi diagram

Contrasting with the visibility graph approach, a Voronoi diagram is a **complete** road map method that tends to maximize the distance between the robot and obstacles in the map. For each point in the free space, compute its distance to the nearest obstacle. Plot that distance in this *figure* as a height coming out of the page. The height increases as you move away from an obstacle. At points that are equidistant from two or more obstacles, such a distance plot has sharp ridges.

Path planning



Path planning

Voronoi diagram

The Voronoi diagram consists of the edges formed by these sharp ridge points. When the configuration space obstacles are polygons, the Voronoi diagram consists of straight and parabolic segments. Algorithms that find paths on the Voronoi road map are complete just like visibility graph methods, because the existence of a path in the free space implies the existence of one on the Voronoi diagram as well (i.e., both methods guarantee completeness). However, the path in the Voronoi diagram is usually **far from optimal** in the sense of total path length.

Path planning

Path-planning overview

Cell decomposition path planning

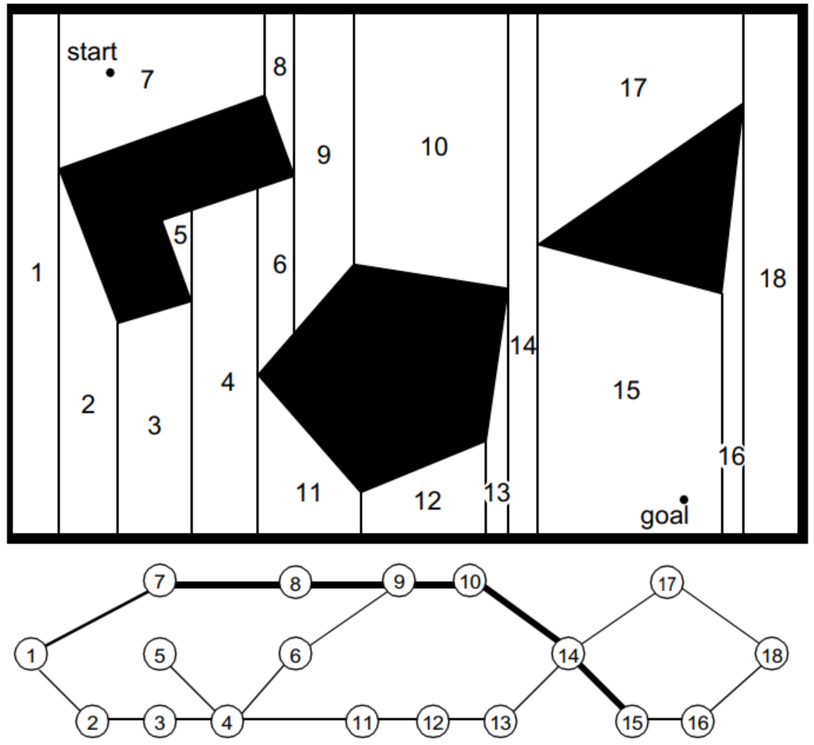
The idea behind cell decomposition is to discriminate between geometric areas, or cells, that are free and areas that are occupied by objects. The basic cell decomposition path-planning algorithm can be summarized as follows. [REF: book]

- Divide into simple, connected regions called “**cells**”.
- Determine which opens cells are adjacent and construct a “**connectivity graph**”.
- Find the cells in which the initial and goal configurations lie and search for a path in the connectivity graph to join the initial and goal cell.
- From the sequence of cells found with an appropriate searching algorithm, compute a path within each cell, for example, passing through the midpoints of the cell boundaries or by a sequence of wall-following motions and movements along straight lines.

Path planning

Path-planning overview

Cell decomposition path planning



Path planning

Path-planning overview

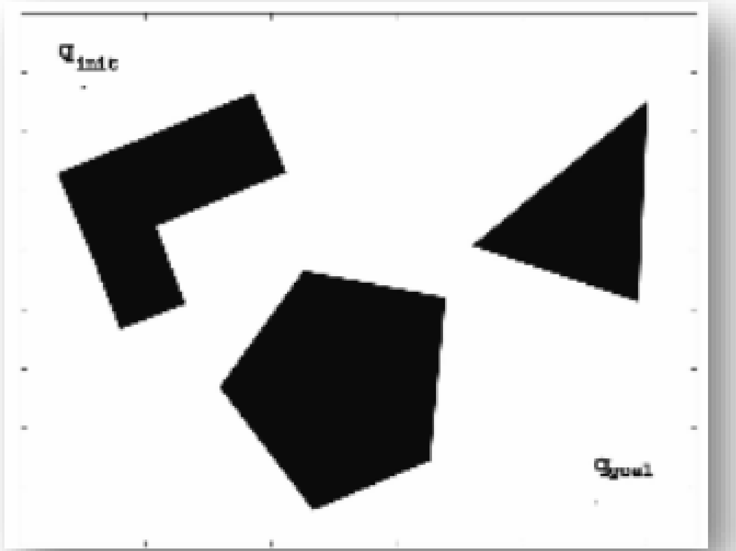
Potential field path planning

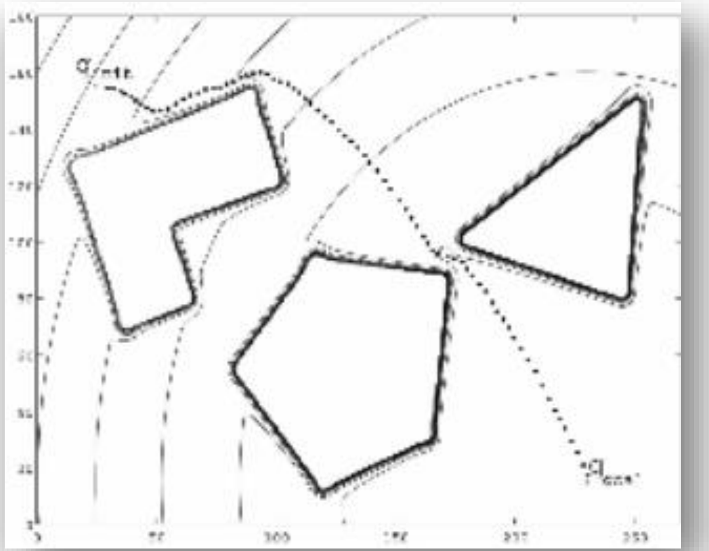
Potential field path planning **creates a field**, or **gradient**, across the robot's map that **directs the robot to the goal position** from multiple prior positions (see this slide). This approach was originally invented for robot manipulator path planning and is used often and under many variants in the mobile robotics community. The potential field method treats the robot as a point under the influence of an artificial potential field . The robot moves by following the field, just as a ball would roll downhill. The goal (a minimum in this space) acts as an **attractive force** on the robot and the obstacles act as **peaks**, or repulsive forces. The superposition of all forces is applied to the robot, which, in most cases, is assumed to be a point in the configuration space. Such an artificial potential field smoothly guides the robot toward the goal while simultaneously avoiding known obstacles.

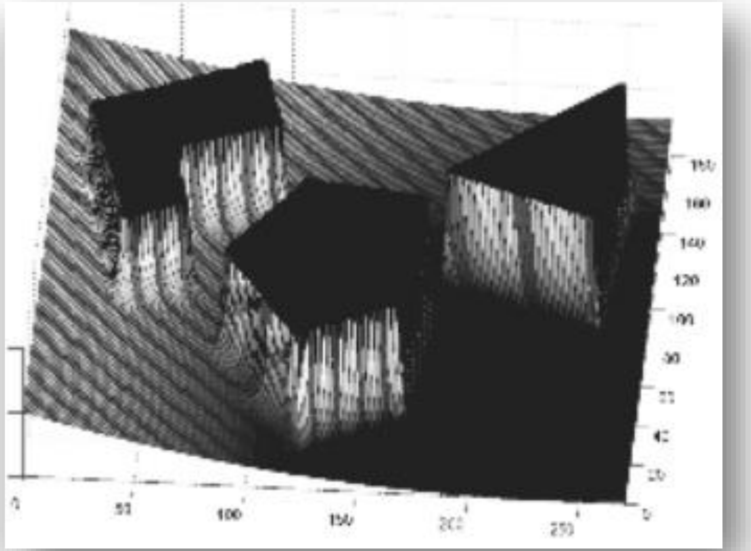
Path planning

Path-planning overview

Potential field path planning







Path planning

Path-planning overview

Potential field path planning

The basic idea behind all potential field approaches is that the robot is attracted toward the goal, while being repulsed by the obstacles that are known in advance. If new obstacles appear during robot motion, one could update the potential field in order to integrate this new information. In the simplest case, we assume that the robot is a point, thus the robot's orientation θ is neglected and the resulting potential field is only 2D (x, y). If we assume a differentiable **potential field function** $U(q)$, we can find the related **artificial force** $F(q)$ acting at the position $q = (x,\ y)$.

$$F(q) = -\nabla U(q)$$

$$\nabla U = \begin{bmatrix} \frac{\partial U}{\partial x} \\ \frac{\partial U}{\partial y} \end{bmatrix}$$

Path planning

Attractive potential: $U_{att}(q) = \frac{1}{2}k_{att} \cdot \rho_{goal}^2(q)$

$$F_{att}(q) = -\nabla U_{att}(q)$$

$$= -k_{att} \cdot \rho_{goal}(q)\nabla\rho_{goal}(q)$$

$$= -k_{att} \cdot (q - q_{goal})$$

Path-planning overview

Potential field path planning

The potential field acting on the robot is then computed as the sum of the attractive field of the goal and the repulsive fields of the obstacles:

$$U(q) = U_{att}(q) + U_{rep}(q)$$

Similarly, the forces can also be separated in an attracting and repulsing part:

$$F(q) = F_{att}(q) - F_{rep}(q)$$

$$= -\nabla U_{att}(q) - \nabla U_{rep}(q)$$

Path planning

Path-planning overview

Potential field path planning

Attractive potential:

$$U_{att}(q) = \frac{1}{2}k_{att} \cdot \rho^2_{goal}(q)$$

$$F_{att}(q) = -\nabla U_{att}(q)$$

$$= -k_{att} \cdot \rho_{goal}(q)\nabla\rho_{goal}(q)$$

$$= -k_{att} \cdot (q - q_{goal})$$

Where k_{att} is a positive scaling factor and $\rho_{goal}(q)$ denotes the ***Euclidean distance*** *$\|q - q_{goal}\|$. This attractive potential is* ***differentiable****, leading to the attractive force F_{att}*

Path planning

Path-planning overview

Potential field path planning

Repulsive potential:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}k_{rep}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)^2 & \text{if } \rho(q) \leq \rho_0 \\ 0 & \text{if } \rho(q) \geq \rho_0 \end{cases}$$

$$F_{rep}(q) = -\nabla U_{rep}(q)$$

$$= \begin{cases} k_{rep}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)\frac{1}{\rho^2(q)}\frac{q - q_{obstacle}}{\rho(q)} & \text{if } \rho(q) \leq \rho_0 \\ 0 & \text{if } \rho(q) \geq \rho_0 \end{cases}$$

Where k_{rep} is again a scaling factor, $\rho(q)$ is the minimal distance from q to the object and the distance of influence of the object. The repulsive potential function U_{rep} is positive or zero and tends to infinity as q gets closer to the object.

robotcitizens.org

Path planning

Obstacle avoidance

Local obstacle avoidance focuses on **changing** the robot’s **trajectory** as informed by its sensors during robot motion. The resulting robot motion is both a function of the robot’s current or recent sensor readings and its goal position and relative location to the goal position. The obstacle avoidance algorithms presented below depend to varying degrees on the existence of a global map and on the robot’s precise knowledge of its location relative to the map. Despite their differences, all of the algorithms after this can be termed obstacle avoidance algorithms because the robot’s local sensor readings play an important role in the robot’s future trajectory.

Path planning

Obstacle avoidance

Algorithm

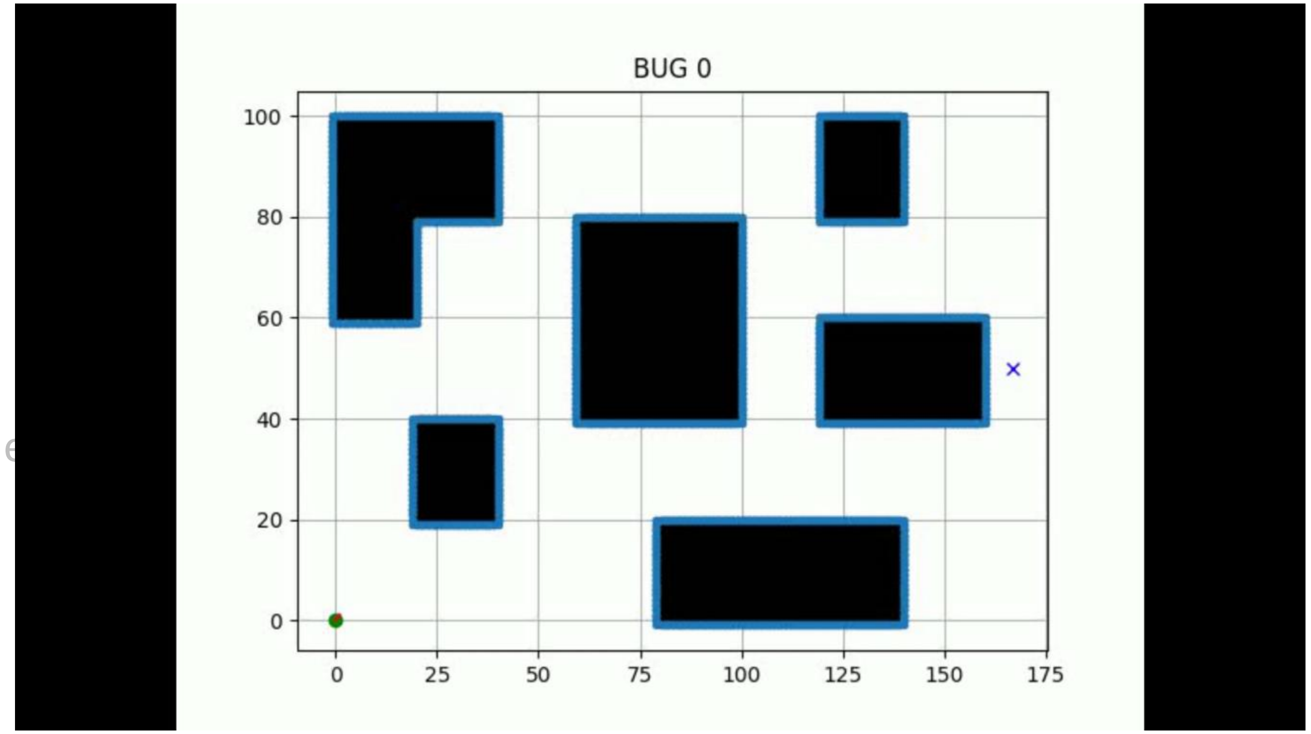
- Bug algorithm
- Vector field histogram
- The bubble band technique
- Curvature velocity techniques
- Dynamic window approaches
- The Schlegel approach to obstacle avoidance
- The ASL approach
- Nearness diagram
- Gradient method
- Adding dynamic constraints

Path planning

Obstacle avoidance

Algorithm

- Bug algorithm
- Vector field histogram
- The bubble band technique
- Curvature velocity techniques
- Dynamic window approaches
- The Schlegel approach to obstacle
- The ASL approach
- Nearness diagram
- Gradient method
- Adding dynamic constraints

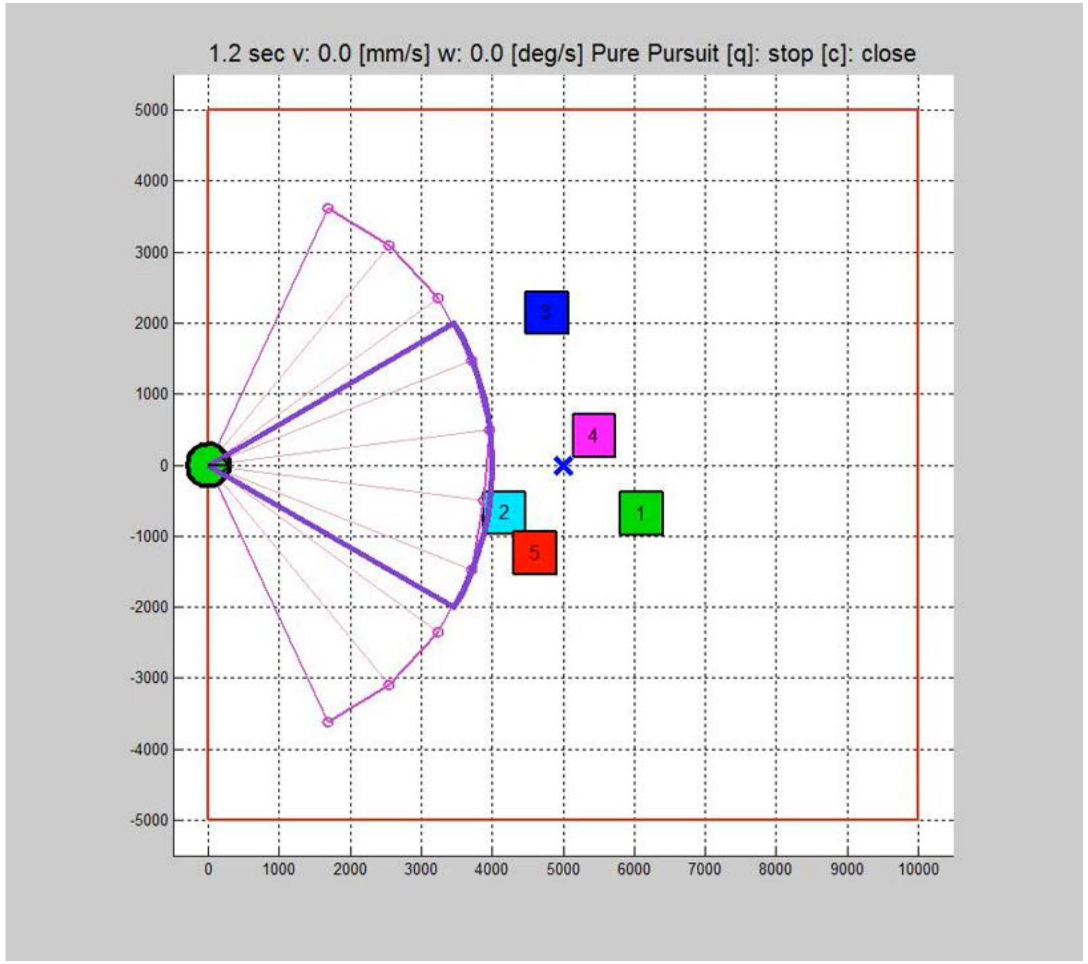


Path planning

Obstacle avoidance

Algorithm

- Bug algorithm
- <u>Vector field histogram</u>
- The bubble band technique
- Curvature velocity techniques
- Dynamic window approaches
- The Schlegel approach to obstacle avoidance
- The ASL approach
- Nearness diagram
- Gradient method
- Adding dynamic constraints



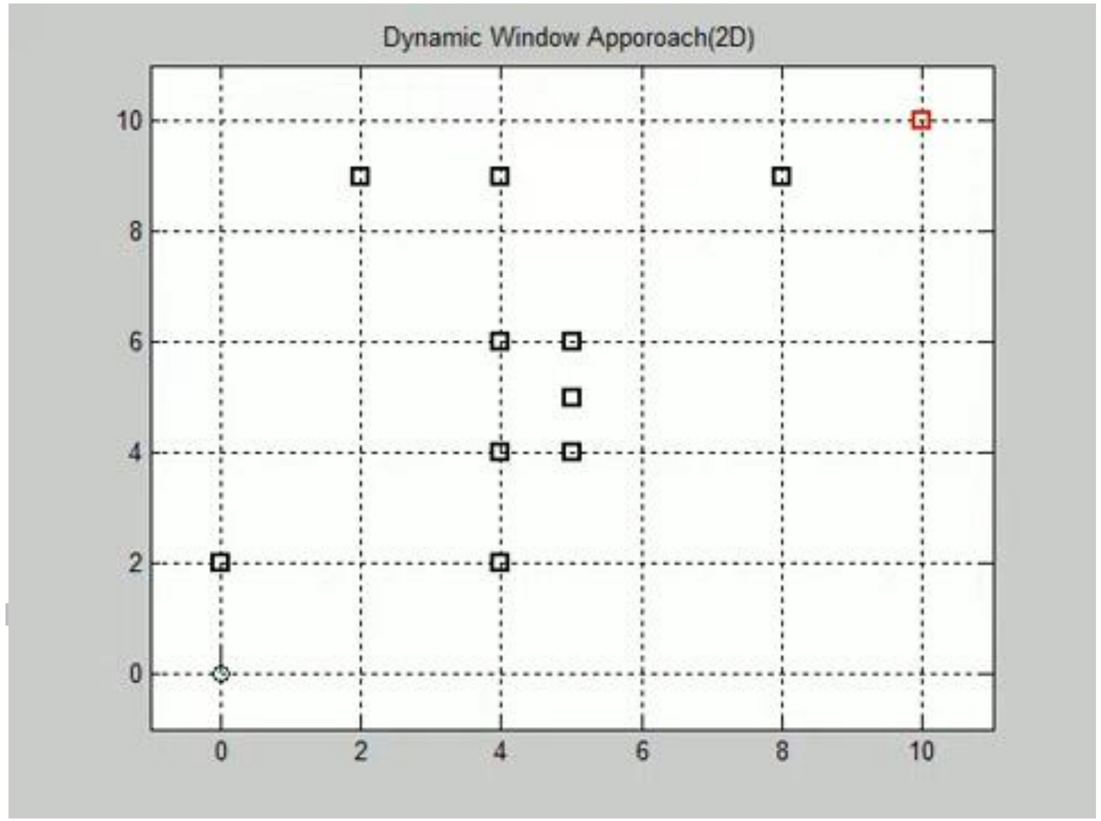
robotcitizens.org

Path planning

Obstacle avoidance

Algorithm

- Bug algorithm
- Vector field histogram
- The bubble band technique
- Curvature velocity techniques
- Dynamic window approaches
- The Schlegel approach to obstacle avoida
- The ASL approach
- Nearness diagram
- Gradient method
- Adding dynamic constraints



robotcitizens.org ROBOTICS FOR ALL AiAT KU KASETSART UNIVERSITY THAI ROBOTICS SOCIETY

Navigation

Navigation system

Navigation Architectures

Control localization

Localization of robot control is an even more critical issue in mobile robot navigation. The basic reason is that a robot architecture includes multiple types of control functionality (e.g., obstacle avoidance, path planning, path execution, etc.). By localizing each functionality to a specific unit in the architecture, we enable individual testing as well as a principled strategy for control composition. For example, consider **collision avoidance**. For stability in the face of changing robot software, as well as for focused verification that the obstacle avoidance system is correctly implemented, it is valuable to localize all software related to the robot's obstacle avoidance process.

Navigation Architectures

Control localization

At the other extreme, high-level planning and task-based decision-making are required for robots to perform useful roles in their environment. It is also valuable to localize **such high-level decision-making software**, enabling it to be tested exhaustively in **simulation** and thus verified even **without a direct connection to the physical robot**. A final advantage of localization is associated with learning. Localization of control can enable a specific learning algorithm to be applied to just one aspect of a mobile robot’s overall control system. Such targeted learning is likely to be the first strategy that yields successful integration of learning and traditional mobile robotics.

The advantages of localization and modularity prove a compelling case for the use of principled navigation architectures.

Navigation Architectures

Techniques for decomposition

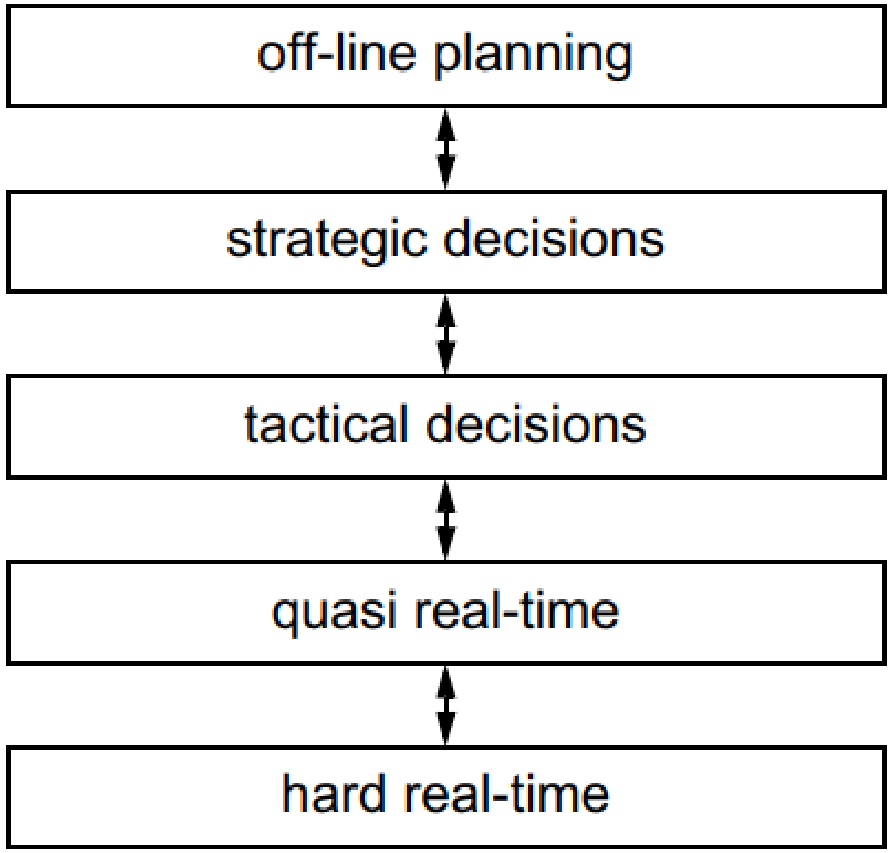
Decompositions identify axes along which we can justify discrimination of robot software into distinct modules. Decompositions also serve as a way to classify various mobile robots into a more quantitative taxonomy**.** ***Temporal decomposition*** distinguishes between real-time and non-real-time demands on mobile robot operation. ***Control decomposition*** identifies the way in which various control outputs within the mobile robot architecture combine to yield the mobile robot’s physical actions. Below we describe each type of decomposition in greater detail.

Navigation Architectures

Temporal decomposition

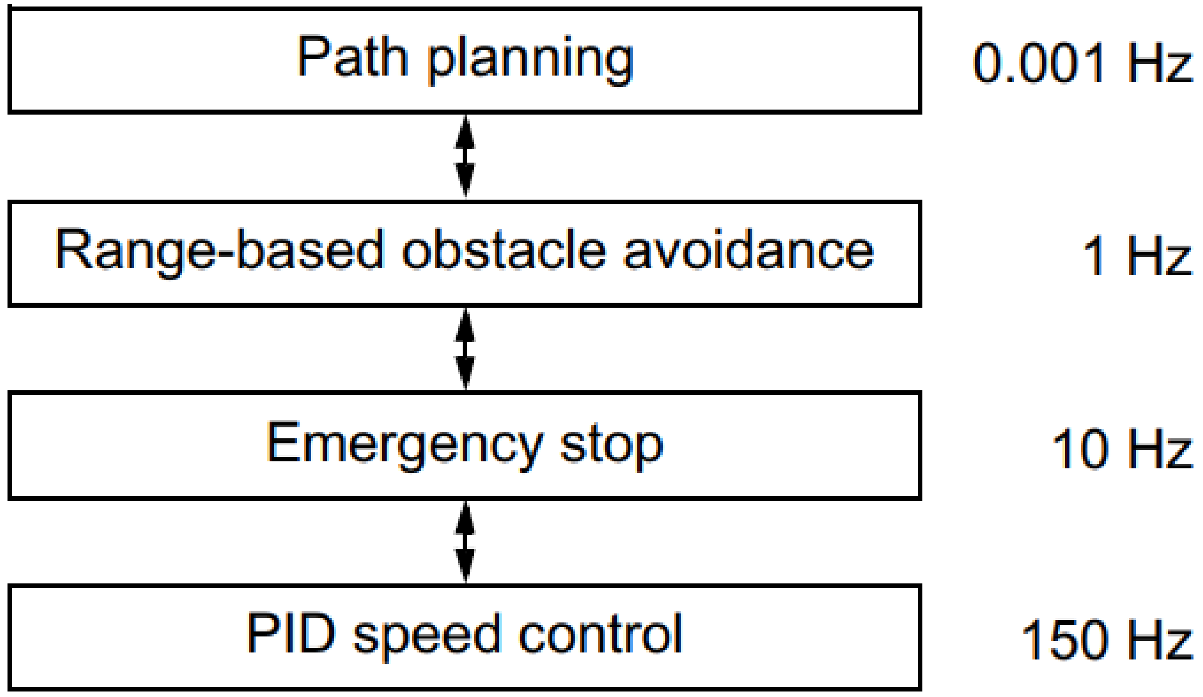
A temporal decomposition of robot software distinguishes between processes that have varying real-time and non-real-time demands. This figure depicts a generic temporal decomposition for navigation. In this figure, the most real-time processes are shown at the bottom of the stack, with the highest category being occupied by processes with no real-time demands.

Navigation Architectures



Navigation Architectures

Temporal decomposition



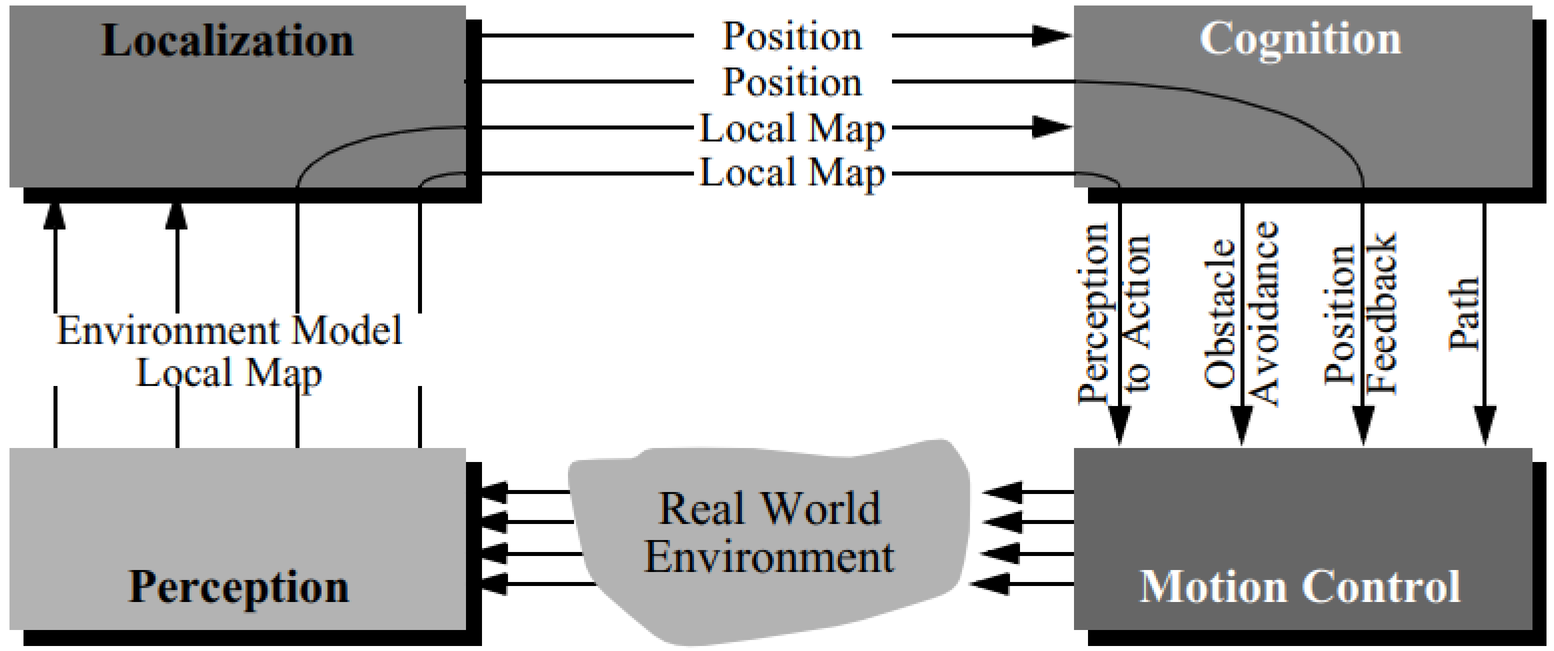
Navigation Architectures

Control decomposition

Whereas temporal decomposition discriminates based on the **time behavior** of software modules, control decomposition identifies the way in which each module's output **contributes** to the overall robot control outputs. Presentation of control decomposition requires the evaluator to understand the basic principles of discrete systems representation and analysis.

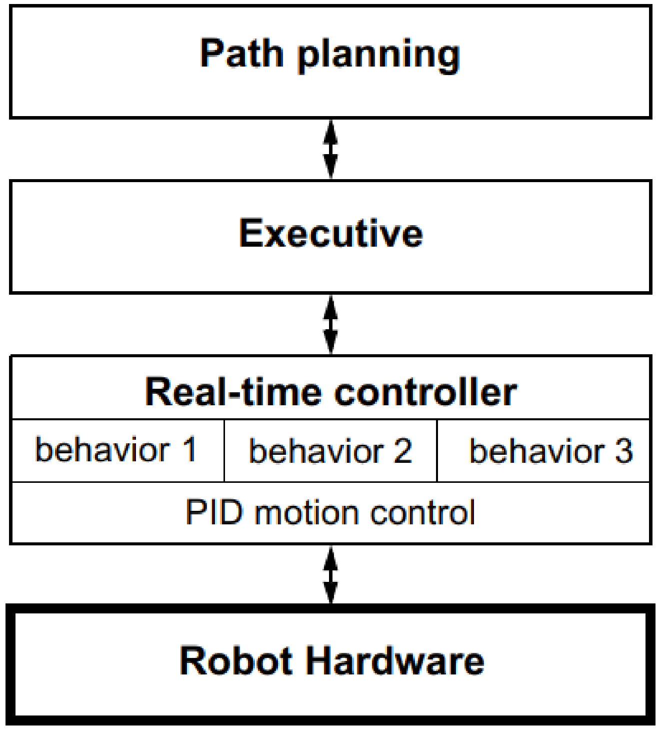
Navigation Architectures

Control decomposition



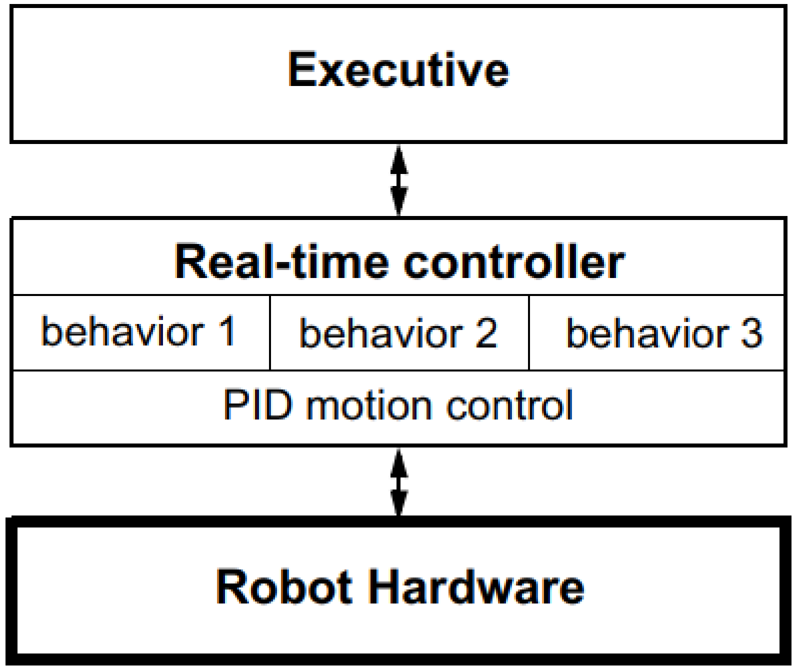
Navigation Architectures

Case study: A general tiered mobile robot



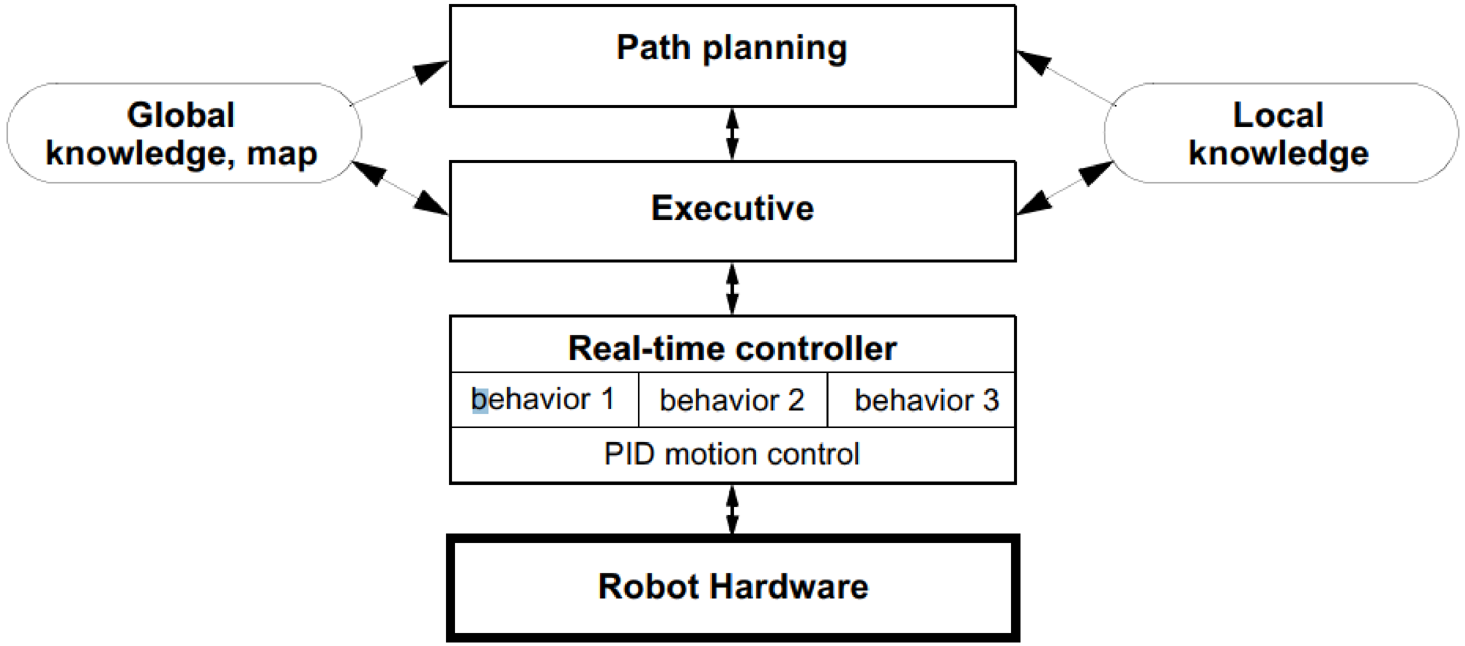
Navigation Architectures

Case study: A two-tiered architecture for off-line planning

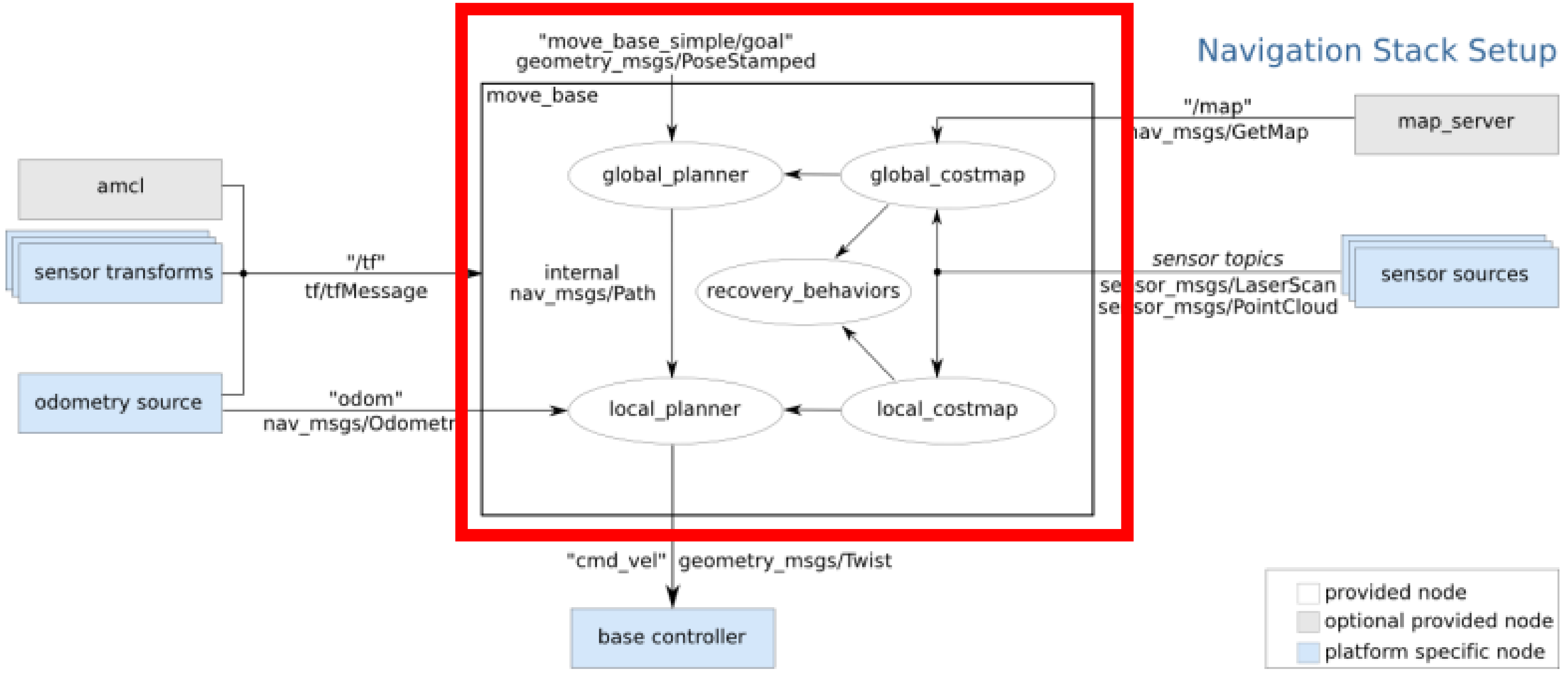


Navigation Architectures

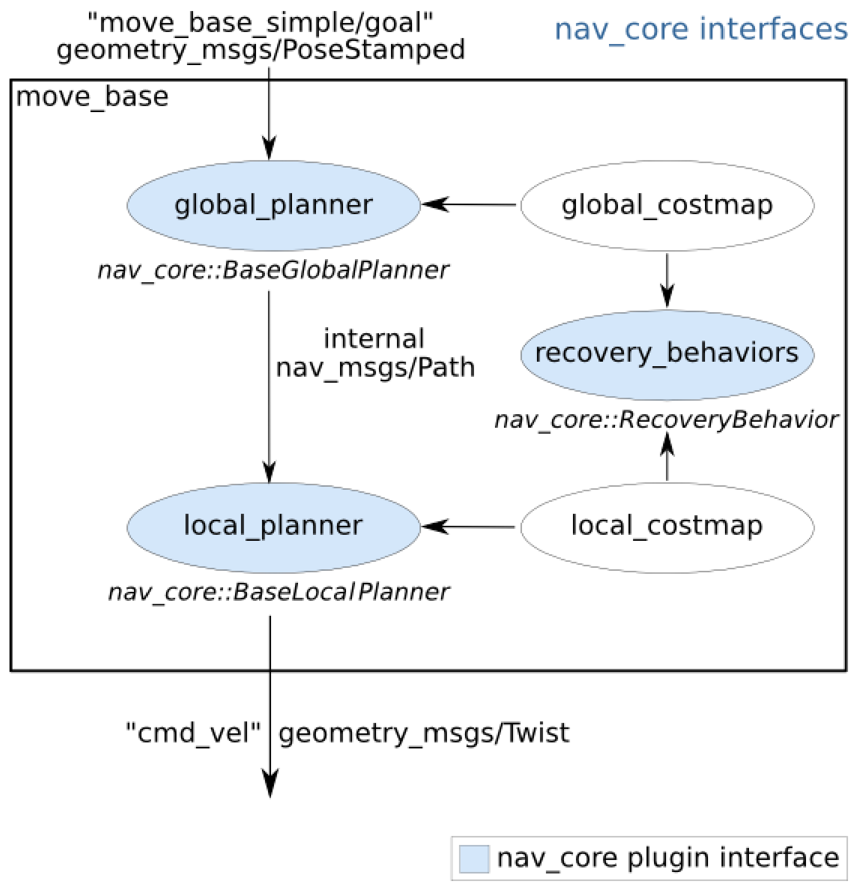
Case study: A three-tiered episodic planning architecture



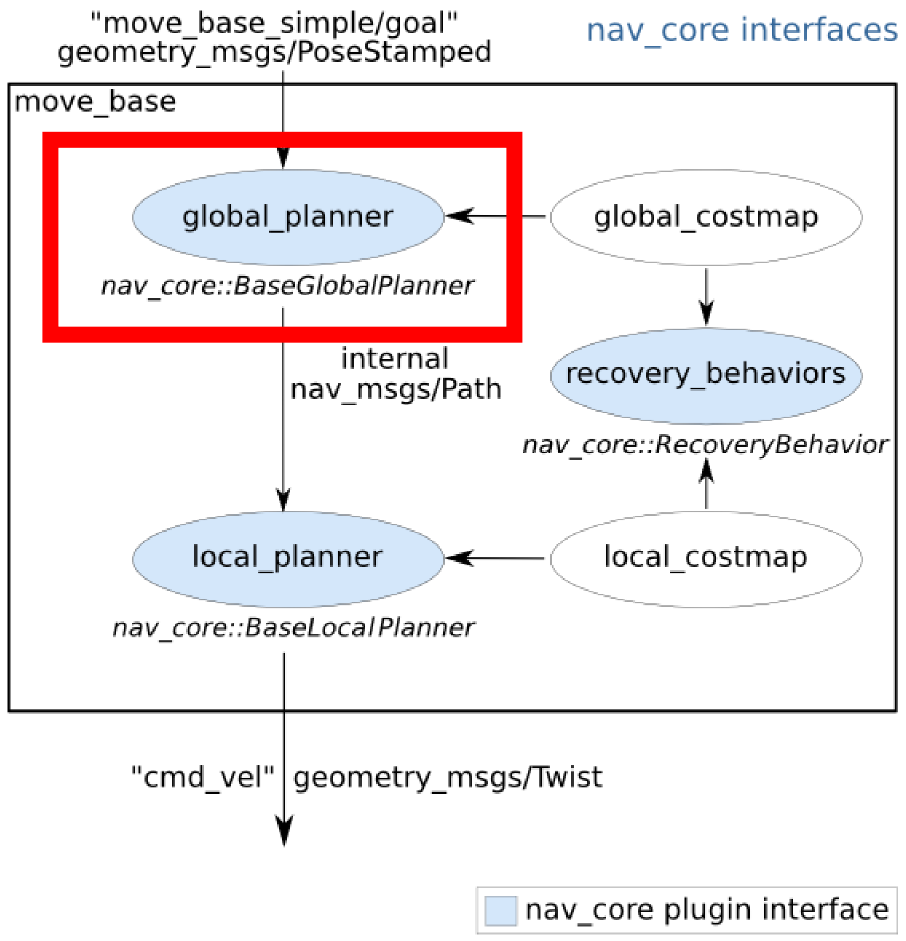
ROS navigation



ROS navigation



ROS navigation


"move_base_simple/goal"
geometry_msgs/PoseStamped
nav_core interfaces
move_base
global_planner
nav_core::BaseGlobalPlanner
global_costmap
internal
nav_msgs/Path
recovery_behaviors
nav_core::RecoveryBehavior
local_planner
local_costmap
nav_core::BaseLocalPlanner
"cmd_vel" geometry_msgs/Twist
nav_core plugin interface

ROS navigation

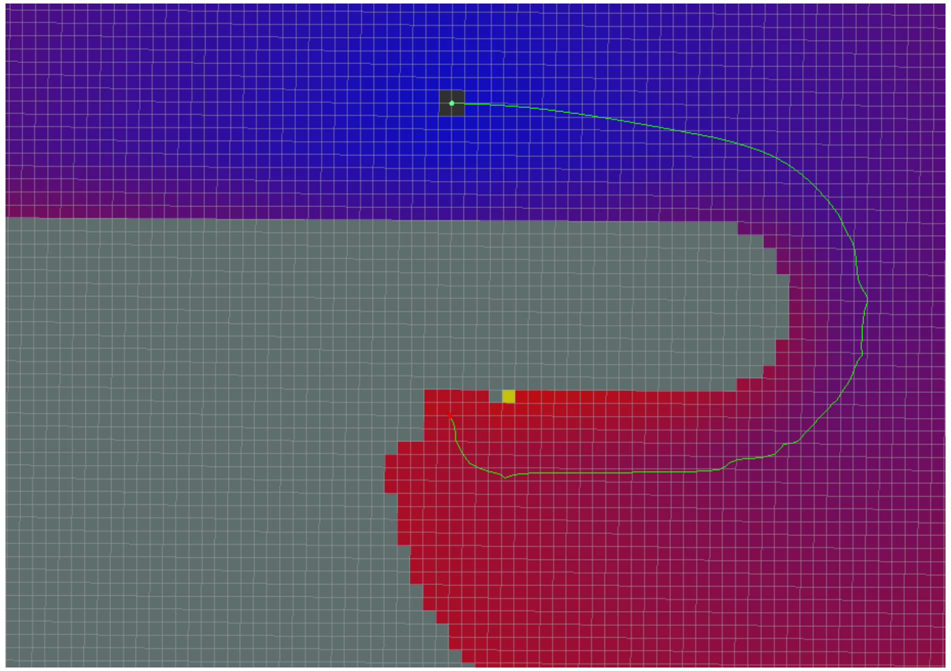
Global planner [global_planner]

This package provides an implementation of a fast, interpolated global planner for navigation. This class adheres to the *nav_core::BaseGlobalPlanner* interface specified in the **nav_core** package. It was built as a more flexible replacement to navfn, which in turn is based on this paper.

ROS navigation

Global planner [global_planner]

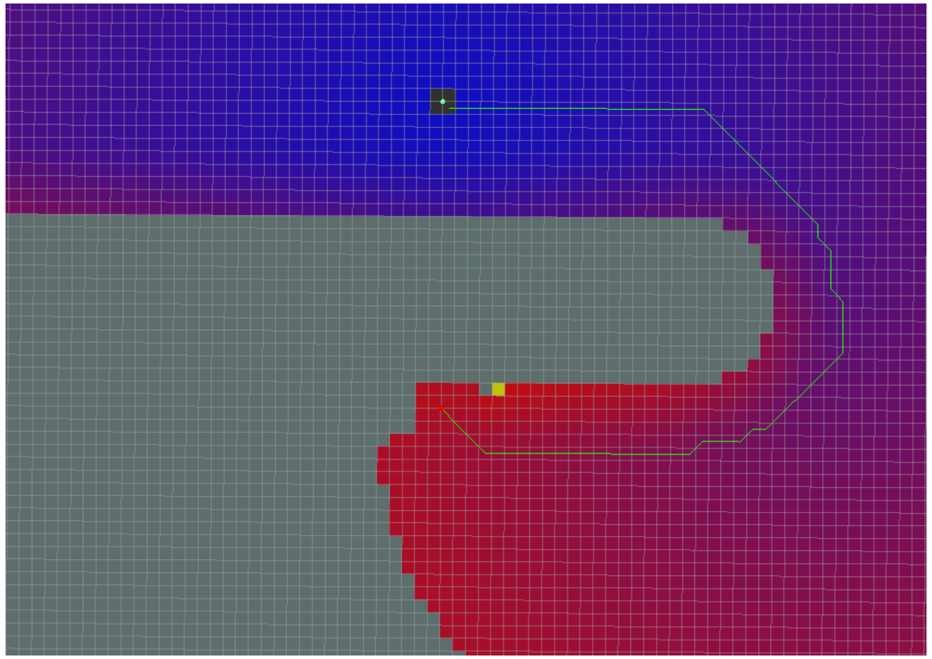
Behavior: Standard Behavior



ROS navigation

Global planner [global_planner]

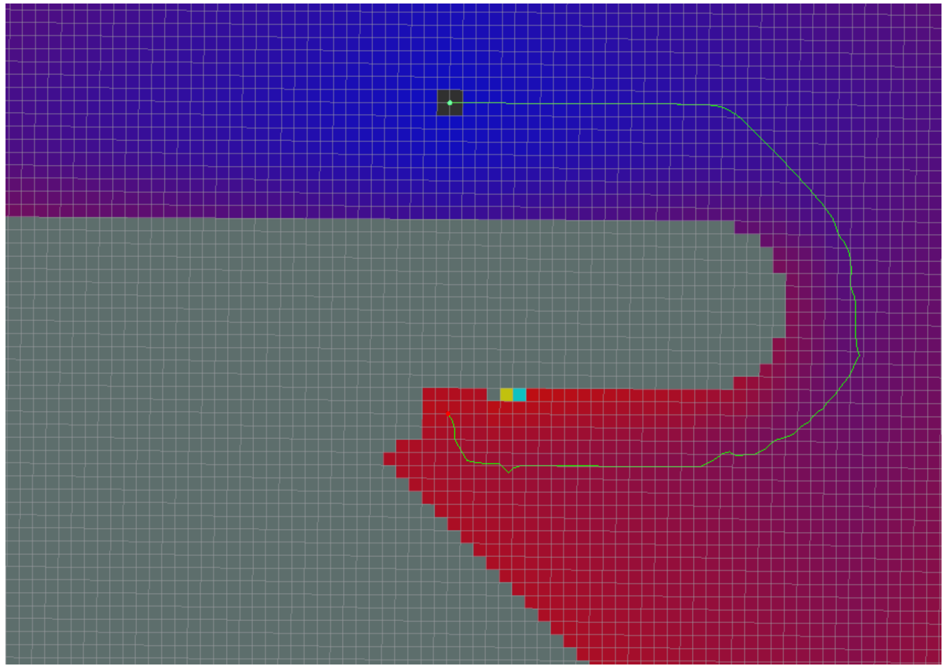
Behavior: Grid Path



ROS navigation

Global planner [global_planner]

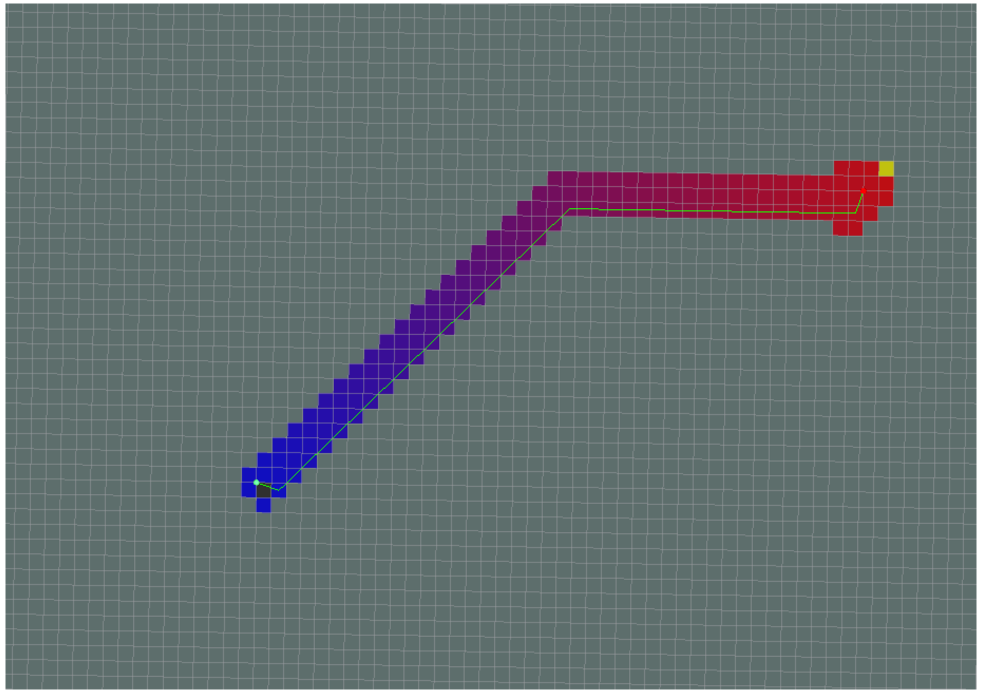
Behavior: Simple Potential Calculation



ROS navigation

Global planner [global_planner]

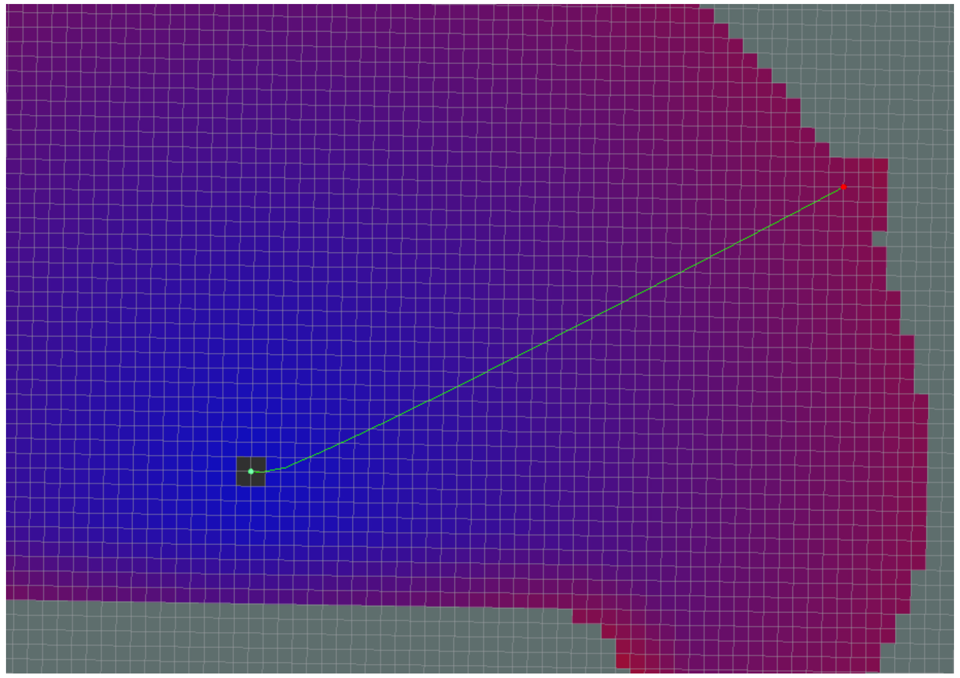
Behavior: A* Path



ROS navigation

Global planner [global_planner]

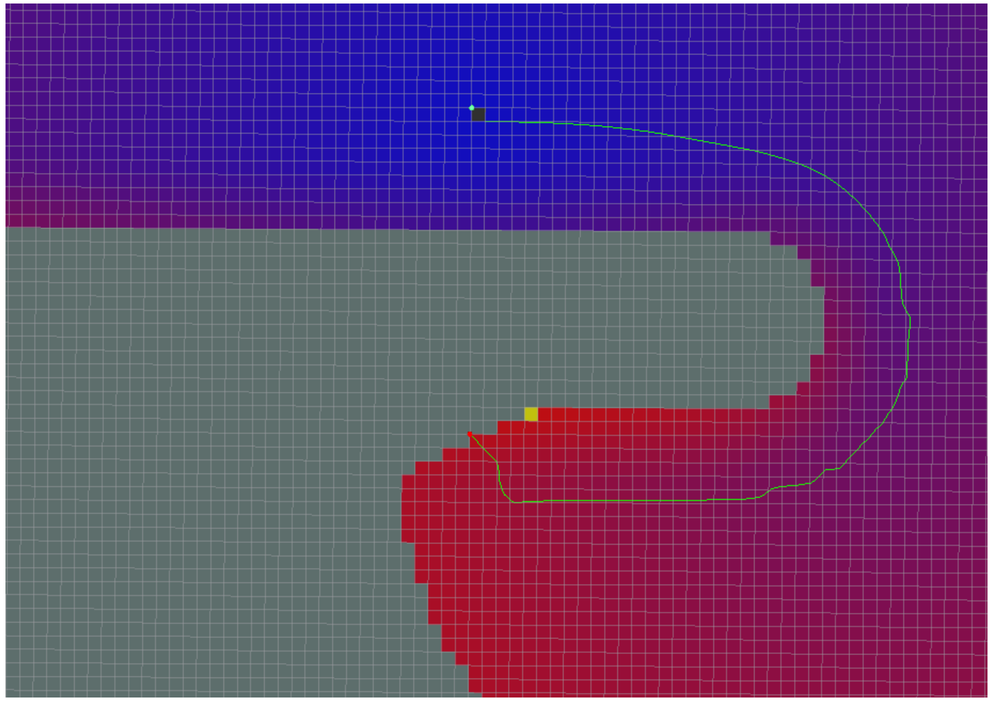
Behavior: Dijkstra's Path



ROS navigation

Global planner [global_planner]

Behavior: Old Navfn Behavior

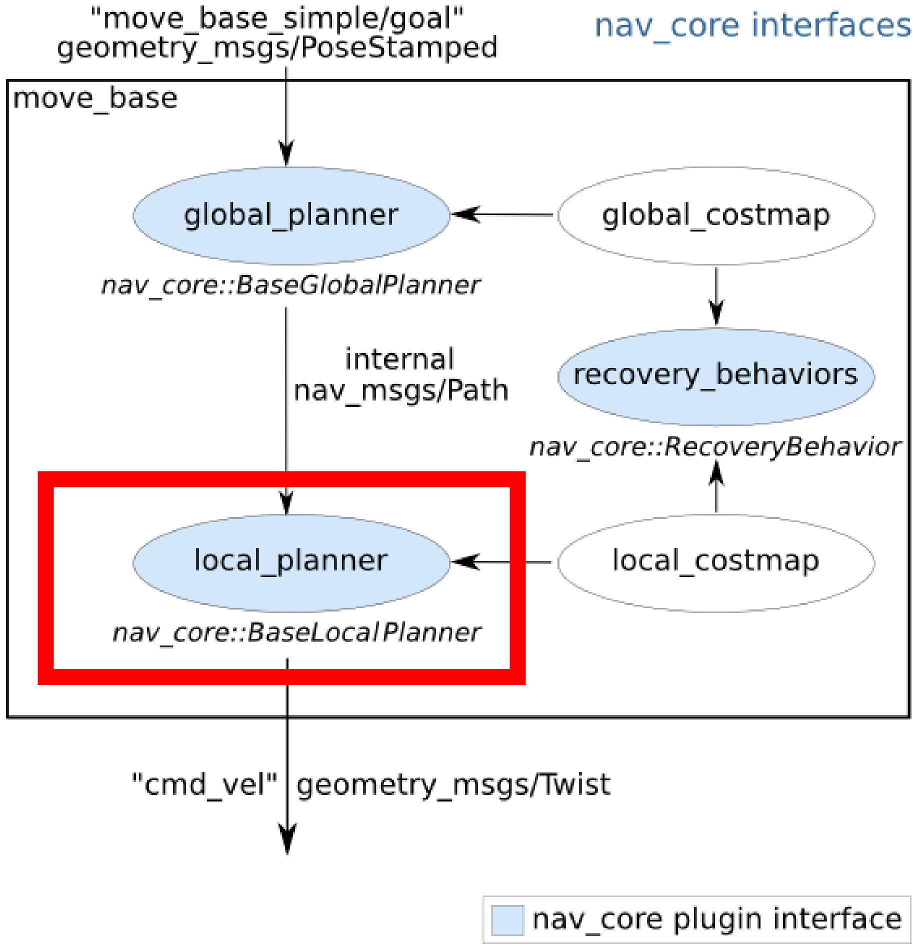


ROS navigation

Global planner [global_planner]

Ref : http://wiki.ros.org/global_planner

ROS navigation



ROS navigation

Local planner [dwa_local_planner]

This package provides an implementation of the **Dynamic Window Approach** to local robot navigation on a plane. Given a global plan to follow and a costmap, the local planner produces velocity commands to send to a mobile base. This package supports any robot whose footprint can be represented as a convex polygon or circle and exposes its configuration as ROS parameters that can be set in a launch file. The parameters for this planner are also dynamically reconfigurable. This package's ROS wrapper adheres to the *BaseLocalPlanner* interface specified in the *nav_core* package.

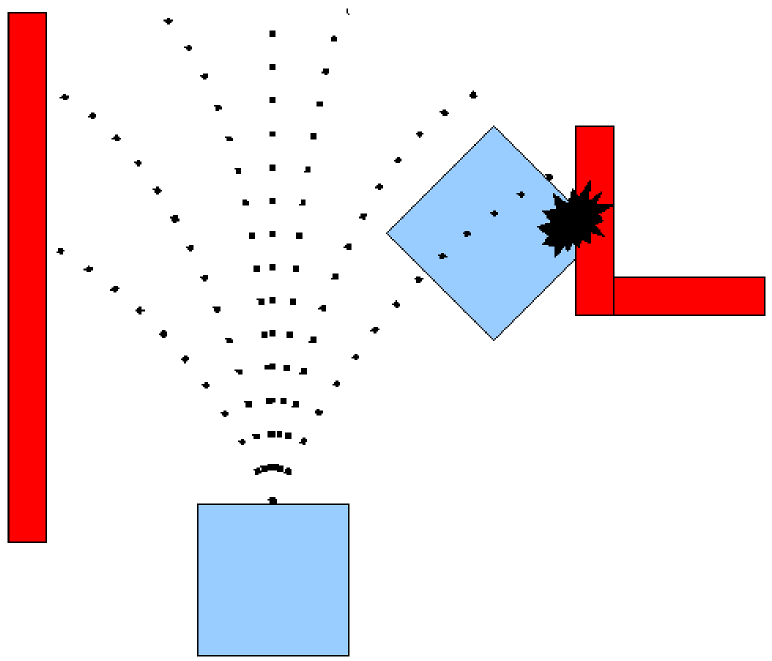
ROS navigation

Local planner [dwa_local_planner]

The *dwa_local_planner* package provides a **controller** that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a **grid map**. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx, dy, dtheta velocities to send to the robot.

ROS navigation

Local planner [dwa_local_planner]



ROS navigation

Local planner [dwa_local_planner]

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.

ROS navigation

Local planner [dwa_local_planner]

Useful references:

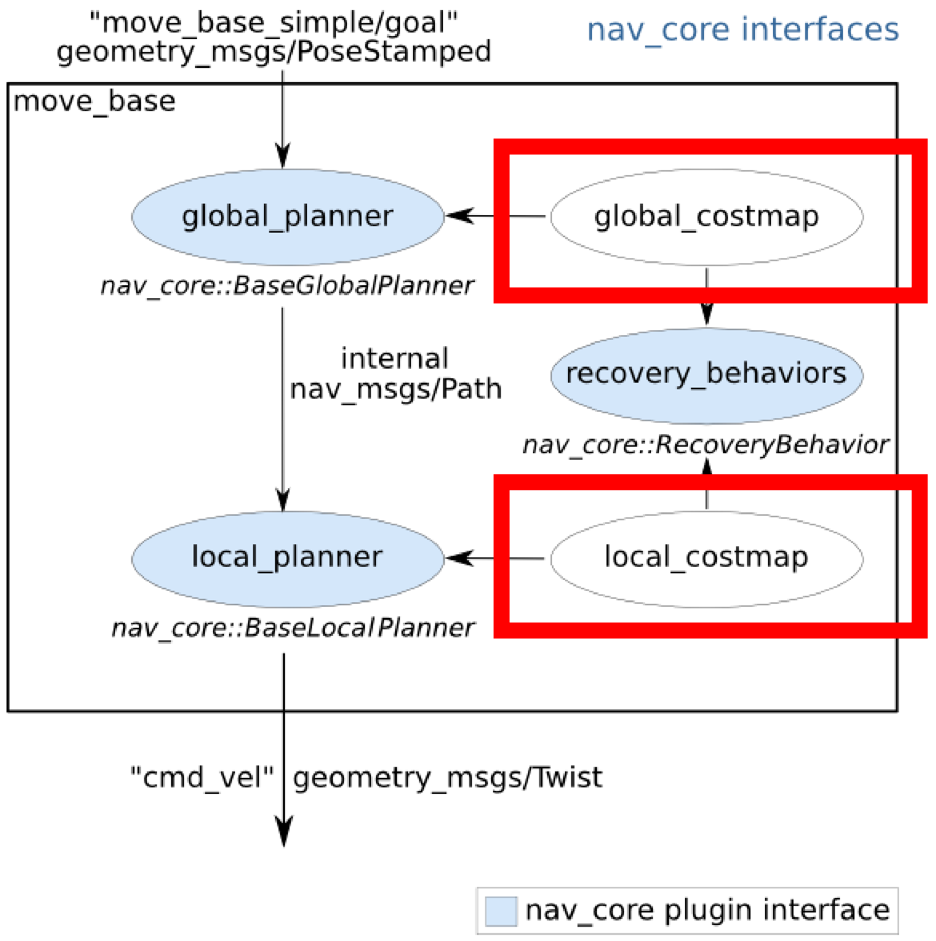
- D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". The Dynamic Window Approach to local control.
- Alonzo Kelly. "An Intelligent Predictive Controller for Autonomous Vehicles". A previous system that takes a similar approach to control.
- Brian P. Gerkey and Kurt Konolige. "Planning and Control in Unstructured Terrain ". Discussion of the Trajectory Rollout algorithm in use on the LAGR robot.

ROS navigation

Local planner [dwa_local_planner]

REF: http://wiki.ros.org/dwa_local_planner?distro=noetic

ROS navigation



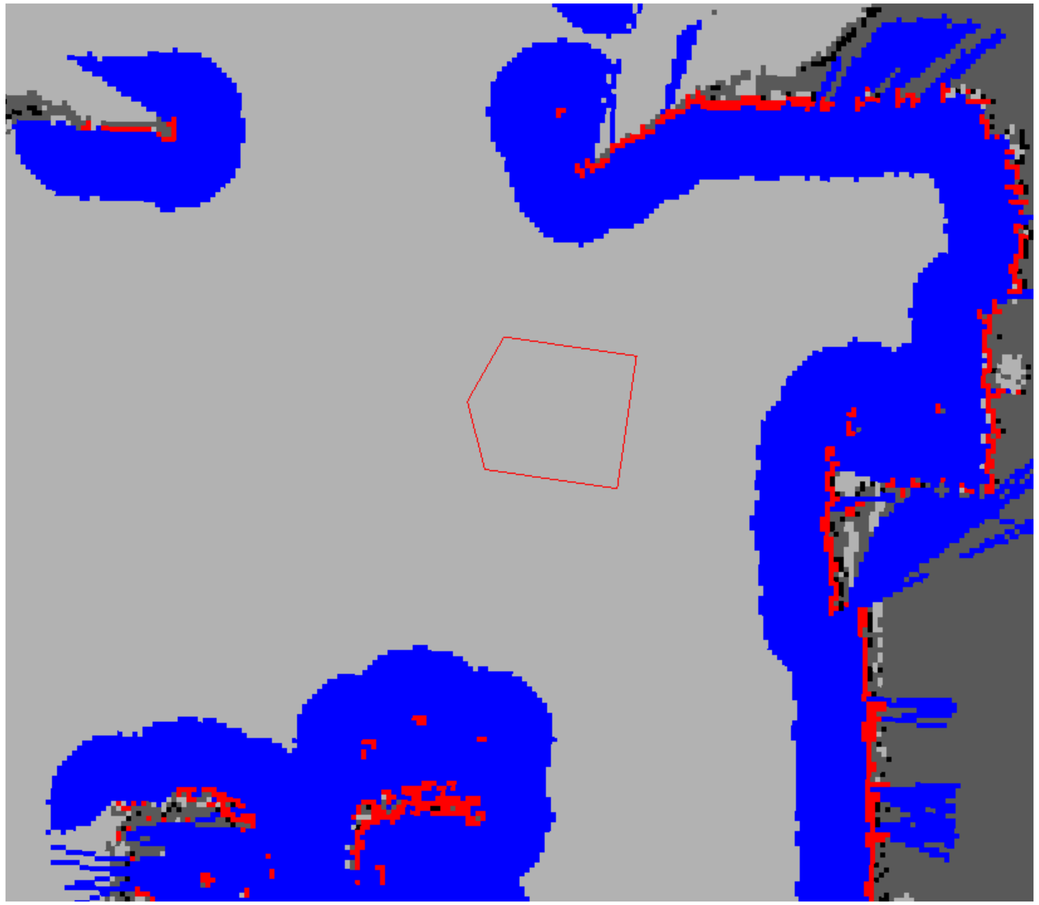
ROS navigation

Cost map [costmap_2d]

This package provides an implementation of a 2D costmap that takes in sensor data from the world, builds a 2D or 3D occupancy grid of the data (depending on whether a voxel-based implementation is used), and inflates costs in a 2D costmap based on the **occupancy grid** and a user specified inflation radius. This package also provides support for map_server based initialization of a costmap, rolling window based costmaps, and parameter-based subscription to and configuration of sensor topics.

ROS navigation

Cost map [costmap_2d]



ROS navigation

Cost map [costmap_2d]

The costmap_2d package provides a configurable structure that maintains information about where the robot should navigate in the form of an **occupancy grid**. The costmap uses sensor data and information from the static map to store and update information about obstacles in the world through the costmap_2d::Costmap2DROS object. The costmap_2d::Costmap2DROS object provides a purely two-dimensional interface to its users, meaning that queries about obstacles can only be made in columns. For example, a table and a shoe in the same position in the XY plane, but with different Z positions would result in the corresponding cell in the costmap_2d::Costmap2DROS object's costmap having an identical cost value. This is designed to help planning in planar spaces.

ROS navigation

Cost map [costmap_2d] : Marking and Clearing

The costmap automatically subscribes to **sensors topics** over ROS and updates itself accordingly. Each sensor is used to either **mark** (insert obstacle information into the costmap), **clear** (remove obstacle information from the costmap), or both. A marking operation is just an index into an array to change the **cost of a cell**. A clearing operation, however, consists of raytracing through a grid from the origin of the sensor outwards for each observation reported. If a three-dimensional structure is used to store obstacle information, obstacle information from each column is projected down into two dimensions when put into the costmap.

ROS navigation

Cost map [costmap_2d] : Occupied, Free, and Unknown Space

While each cell in the costmap can have one of **255 different cost** values, the underlying structure that it uses is capable of representing only three. Specifically, each cell in this structure can be either **free**, **occupied**, or **unknown**. Each status has a special cost value assigned to it upon projection into the costmap. Columns that have a certain number of **occupied cells** are assigned a *costmap_2d::LETHAL_OBSTACLE* cost, columns that have a certain number of **unknown cells** are assigned a *costmap_2d::NO_INFORMATION* cost, and **other columns** are assigned a *costmap_2d::FREE_SPACE* cost.

ROS navigation

Cost map [costmap_2d] : Map Updates

The costmap performs map update cycles at the rate specified by the **update_frequency** parameter. Each cycle, sensor data comes in, marking and clearing operations are perfumed in the underlying occupancy structure of the costmap, and this structure is projected into the costmap where the appropriate cost values are assigned as described above. After this, each **obstacle inflation** is performed on each cell with a *costmap_2d::LETHAL_OBSTACLE* cost. This consists of propagating cost values outwards from each occupied cell out to a user-specified inflation radius. The details of this inflation process are outlined below.

ROS navigation

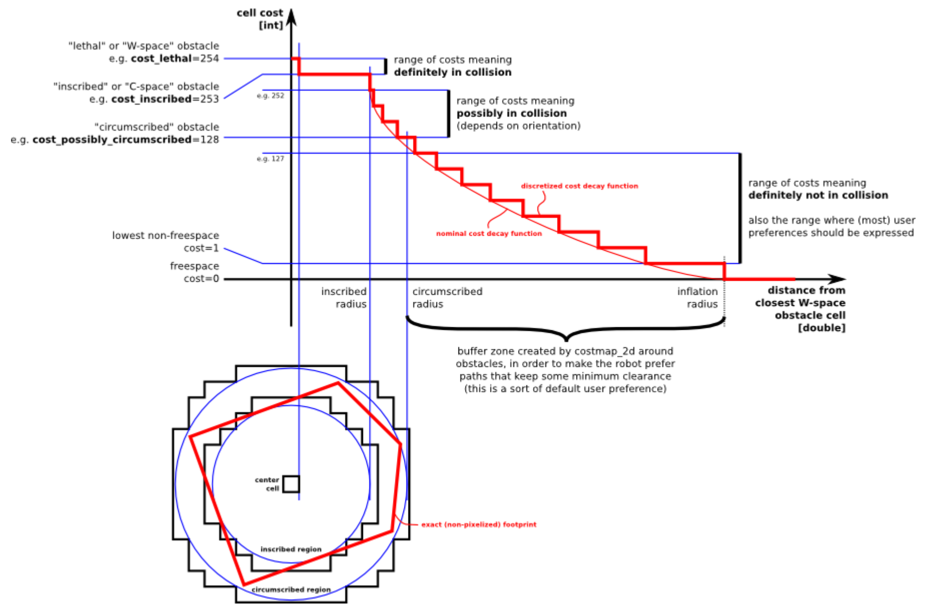
Cost map [costmap_2d] : tf

In order to insert data from sensor sources into the costmap, the *costmap_2d::Costmap2DROS* object makes extensive use of tf. Specifically, it assumes that all transforms between the coordinate frames specified by the **global_frame** parameter, the **robot_base_frame** parameter, and sensor sources are connected and up-to-date. The transform_tolerance parameter sets the maximum amount of latency allowed between these transforms. If the tf tree is not updated at this expected rate, the navigation stack stops the robot.

ROS navigation

Cost map [costmap_2d] : Inflation

Inflation is the process of propagating cost values out from occupied cells that decrease with distance. For this purpose, we define 5 specific symbols for costmap values as they relate to a robot.



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
exact (non-pixelized) footprint
inscribed region
circumscribed region

ROS navigation

Cost map [costmap_2d] : Inflation

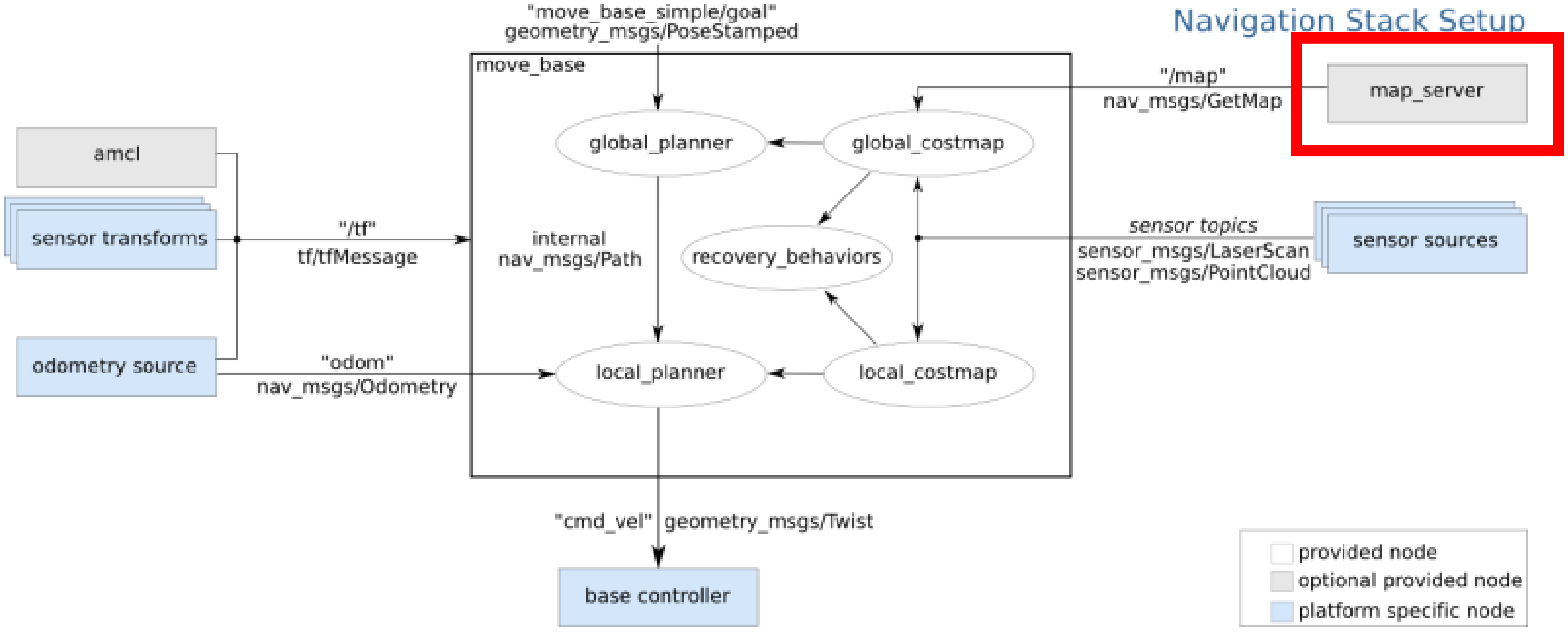
- "**Lethal**" cost means that there is an **actual** (workspace) obstacle in a cell. So, if the robot's center were in that cell, the robot would obviously be in collision.
- "**Inscribed**" cost means that a cell is **less than the robot's inscribed radius** away from an actual obstacle. So, the robot is certainly in collision with some obstacle if the robot center is in a cell that is at or above the inscribed cost.
- "**Possibly circumscribed**" cost is similar to "inscribed", but using the robot's circumscribed radius as **cutoff distance**. Thus, if the robot center lies in a cell at or above this value, then it depends on the orientation of the robot whether it collides with an obstacle or not. We use the term "possibly" because it might be that it is not really an obstacle cell, but some user-preference, that put that particular cost value into the map. For example, if a user wants to express that a robot should attempt to avoid a particular area of a building, they may inset their own costs into the costmap for that region independent of any obstacles. Note, that although the value is 128 is used as an example in the diagram above, the true value is influenced by both the inscribed_radius and inflation_radius parameters as defined in the code.
- "**Freespace**" cost is assumed to be **zero**, and it means that there is nothing that should keep the robot from going there.
- "**Unknown**" cost means there is **no information** about a given cell. The user of the costmap can interpret this as they see fit.
- All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on their distance from a "Lethal" cell and the decay function provided by the user.

ROS navigation

Cost map [costmap_2d] : Inflation

REF: http://wiki.ros.org/costmap_2d?distro=noetic

ROS navigation

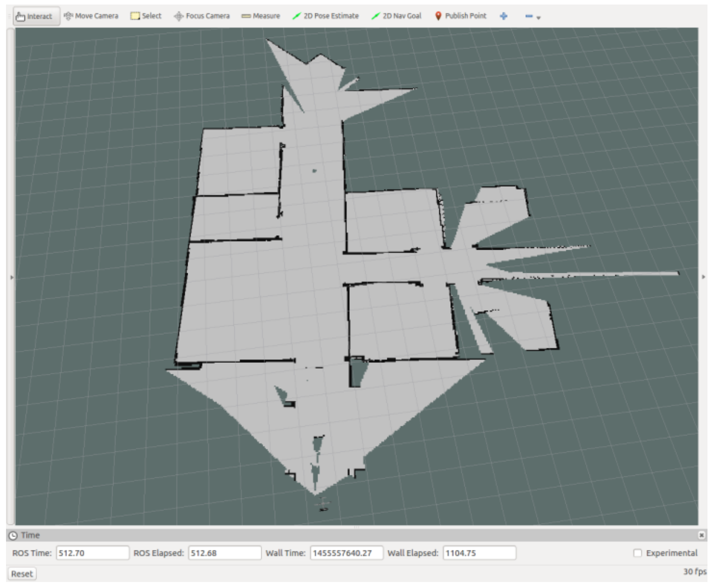


robotcitizens.org

ROS navigation

Map server [map_server]

Map-server provides the map_server ROS Node, which offers map data as a ROS Service. It also provides the **map_saver** command-line utility, which allows dynamically generated maps to be saved to file.



ROS Time: 512.70 ROS Elapsed: 512.68 Wall Time: 1455557640.27 Wall Elapsed: 1104.75

ROS navigation

Map server [map_server]

Map format

1. Image format
2. YAML format

ROS navigation

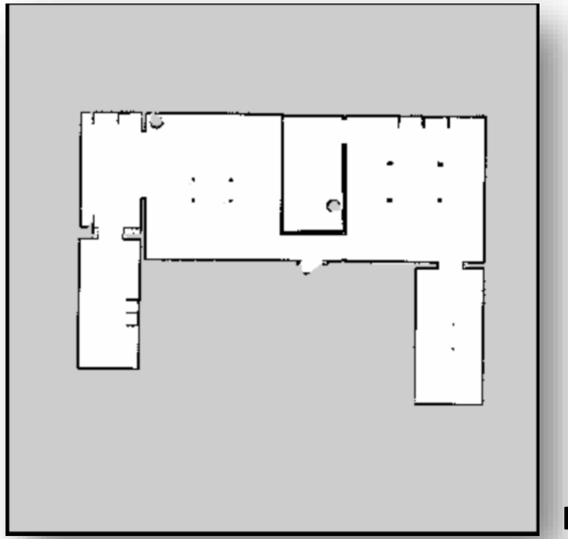
Map server [map_server]

Map format

1. **Image format**

   The image describes the **occupancy** state of each cell of the world in the color of the corresponding pixel. In the standard configuration, **whiter pixels are free**, **blacker pixels are occupied**, and **pixels in between are unknown**. Color images are accepted, but the color values are averaged to a gray value.

2. **YAML format**


map.pgm

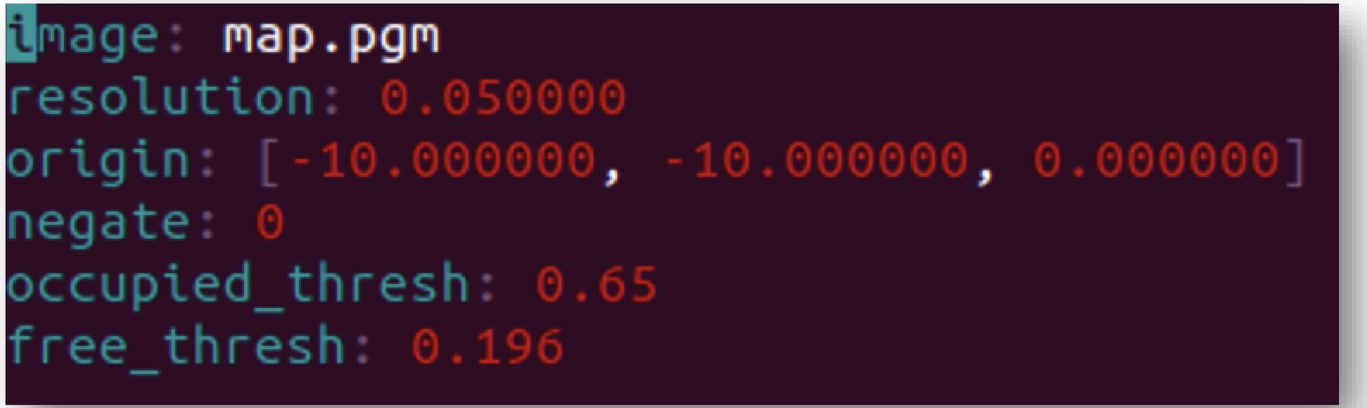
ROS navigation

Map server [map_server]

Map format

1. Image format
2. YAML format

map.yaml



```
image: map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

- **image** : **Path to the image file** containing the occupancy data; can be absolute, or relative to the location of the YAML file
- **resolution** : Resolution of the map, **meters / pixel**
- **origin** : The 2-D pose of the lower-left pixel in the map, as (x, y, yaw), with yaw as counterclockwise rotation (yaw=0 means no rotation). Many parts of the system currently ignore yaw.
- **occupied_thresh** : Pixels with occupancy probability greater than this threshold are considered completely occupied.
- **free_thresh** : Pixels with occupancy probability less than this threshold are considered completely free.
- **negate** : Whether the white/black free/occupied semantics should be reversed (interpretation of thresholds is unaffected)

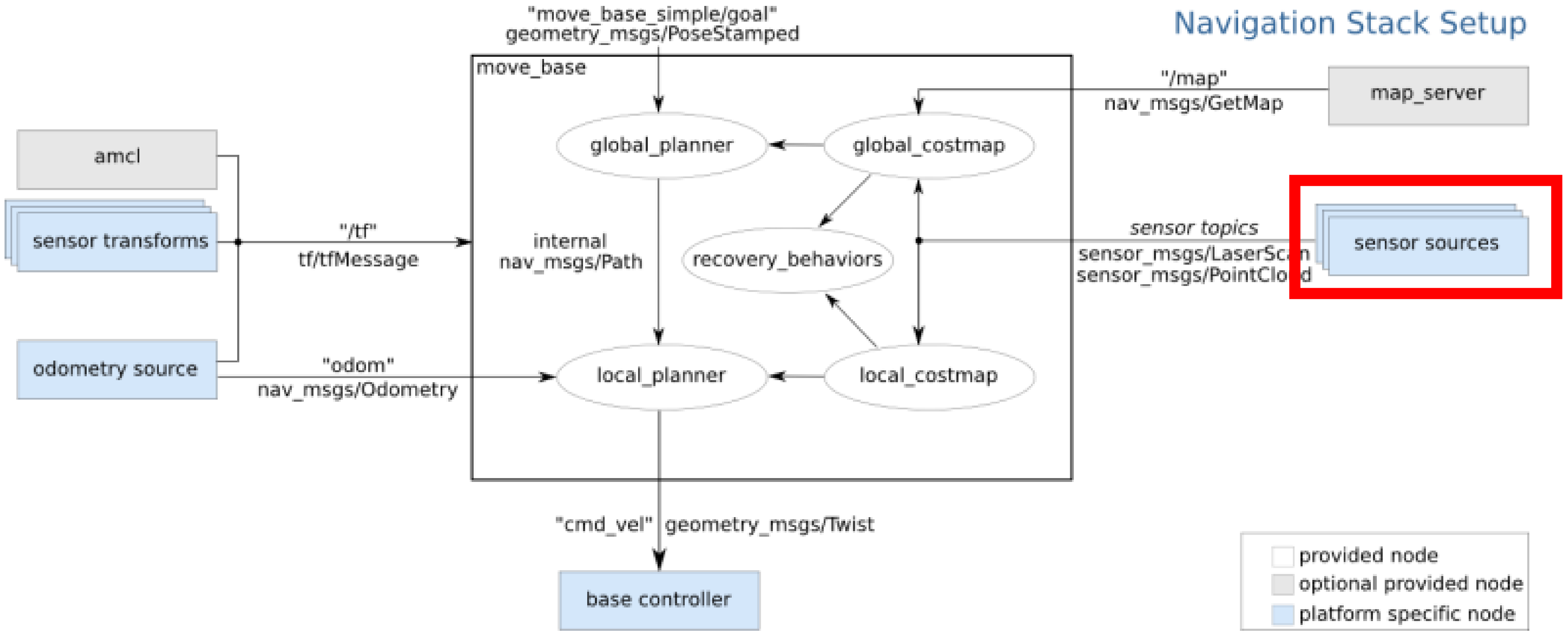
robotcitizens.org

ROS navigation

Map server [map_server]

REF: http://wiki.ros.org/map_server?distro=noetic

ROS navigation



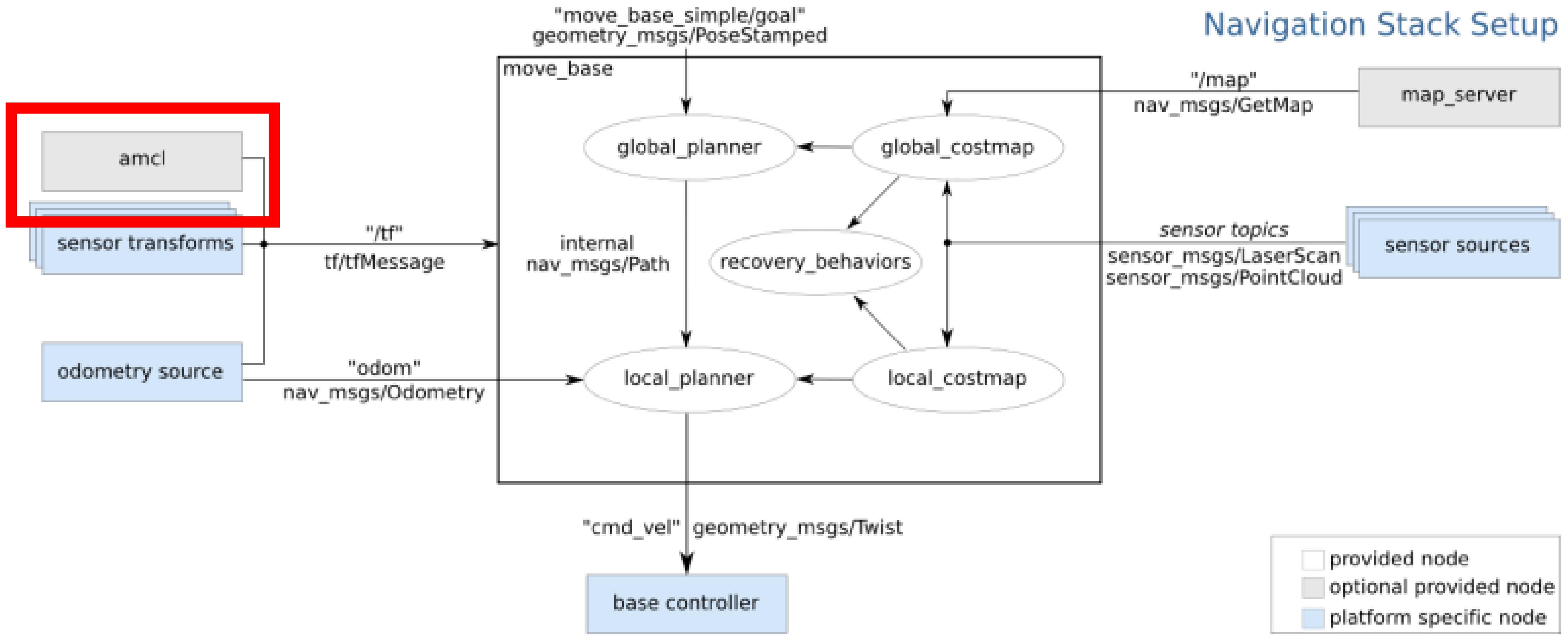
robotcitizens.org

ROS navigation

Sensor source

- LiDAR
- Camera
- 3D Camera

ROS navigation



ROS navigation

Adaptive Monte Carlo localization (AMCL)

Monte Carlo localization (MCL), also known as **particle filter localization** , is an algorithm for robots to localize using a particle filter. Given a map of the environment, the algorithm estimates the position and orientation of a robot as it moves and senses the environment.

AMCL dynamically adjusts the number of particles **based on KL distance** to ensure that the particle distribution converge to the true distribution of robot state based on all past sensor and motion measurements with high probability.

ROS navigation

Adaptive Monte Carlo localization (AMCL)

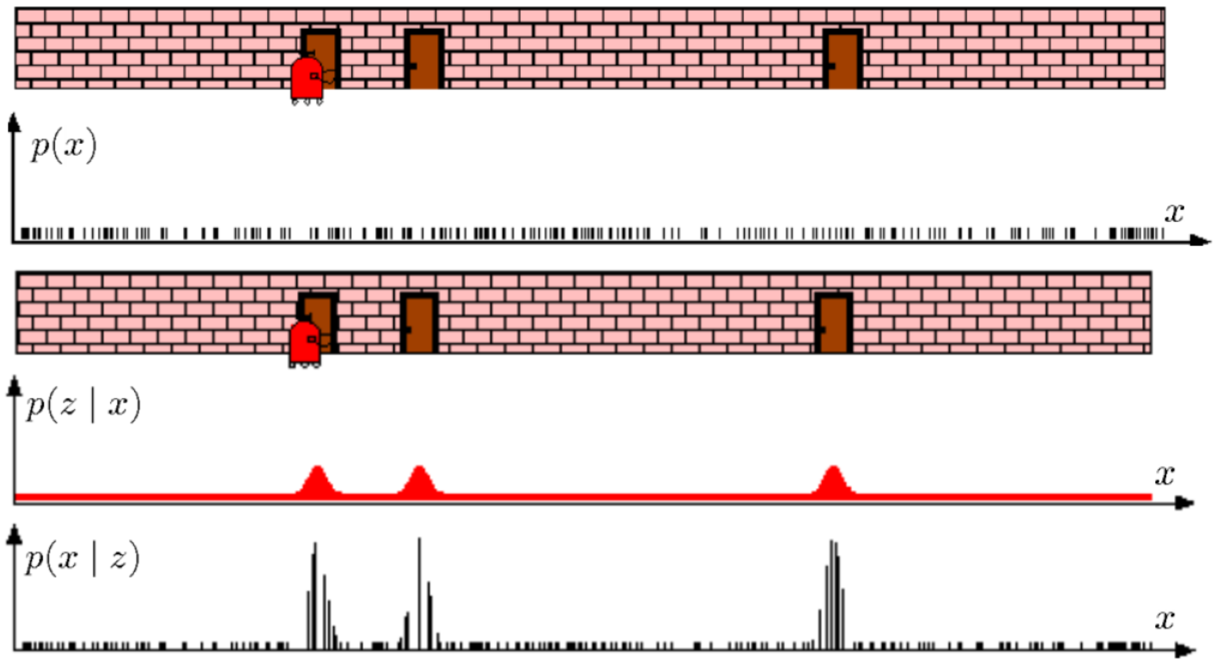
Particle Filters (PF)

- **Sample the next generation** for particles using the proposal distribution (Often use the motion model)
- Compute the importance **weights**
- **Resampling** : “Replace unlikely samples by more likely”

ROS navigation

Adaptive Monte Carlo localization (AMCL)

Sensor Information: Importance Sampling



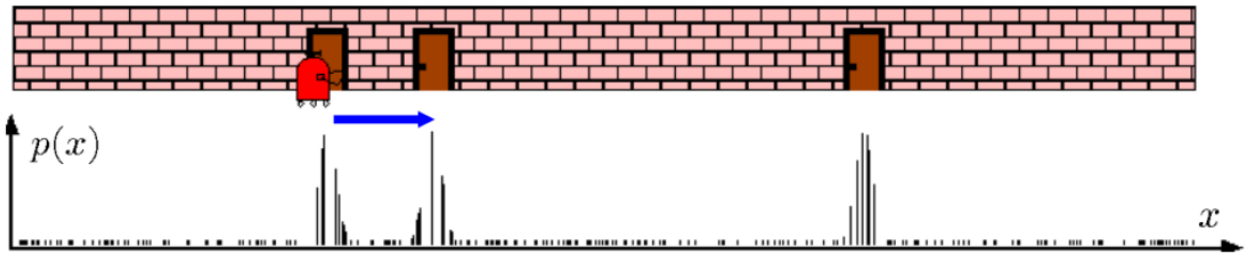
Sensor Model

Weighting

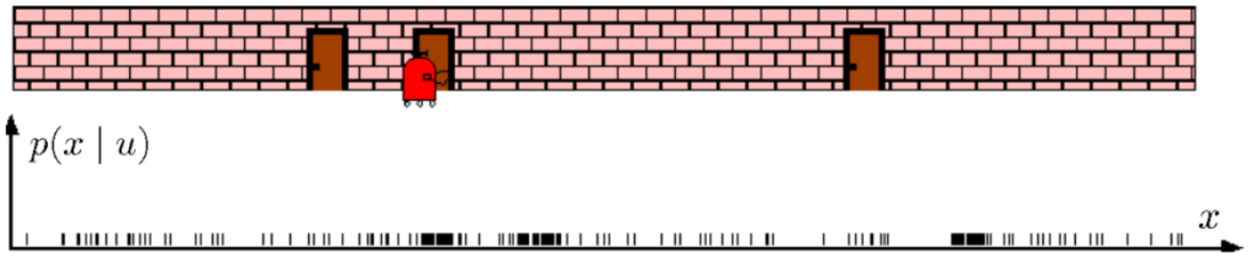
ROS navigation

Adaptive Monte Carlo localization (AMCL)

Robot Motion



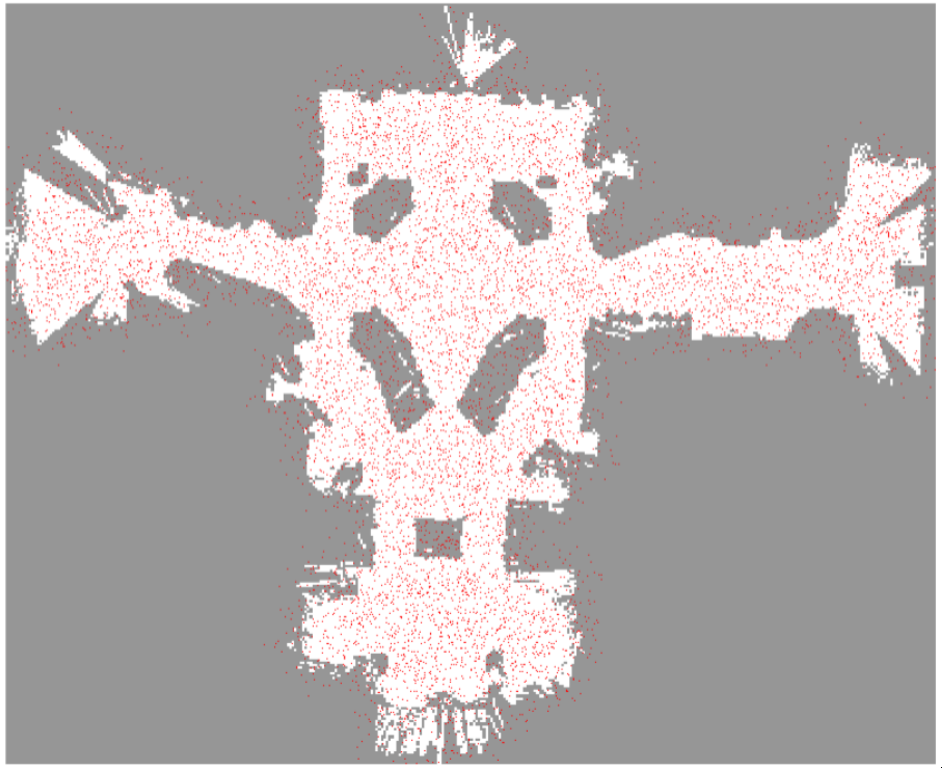
Sampling
Motion Model



Resampling

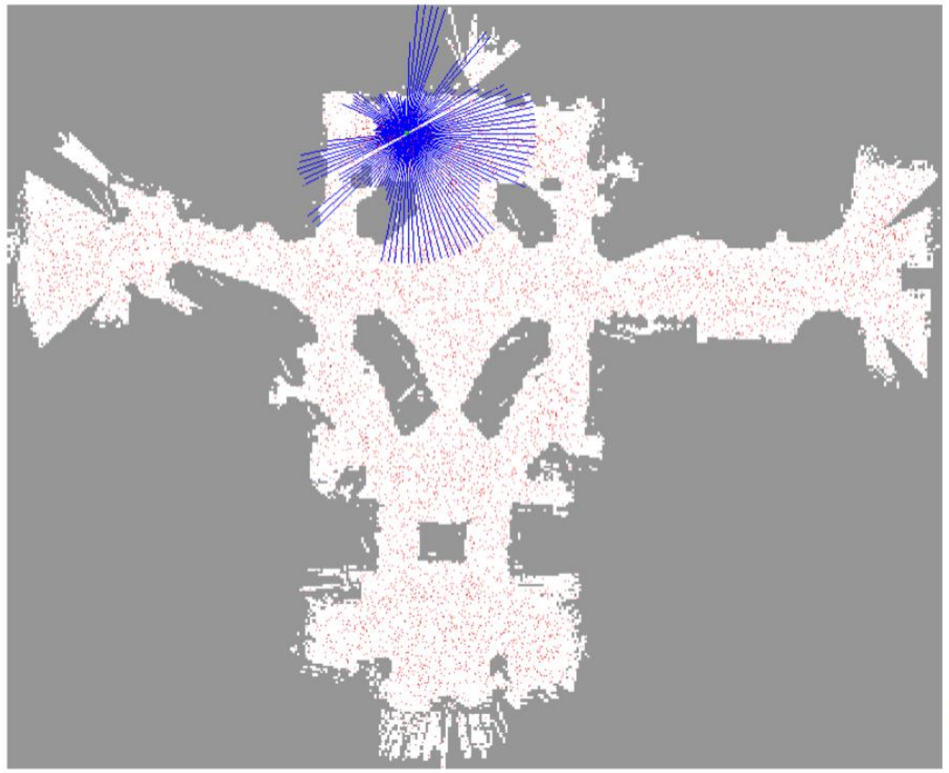
ROS navigation

Adaptive Monte Carlo localization (AMCL)



ROS navigation

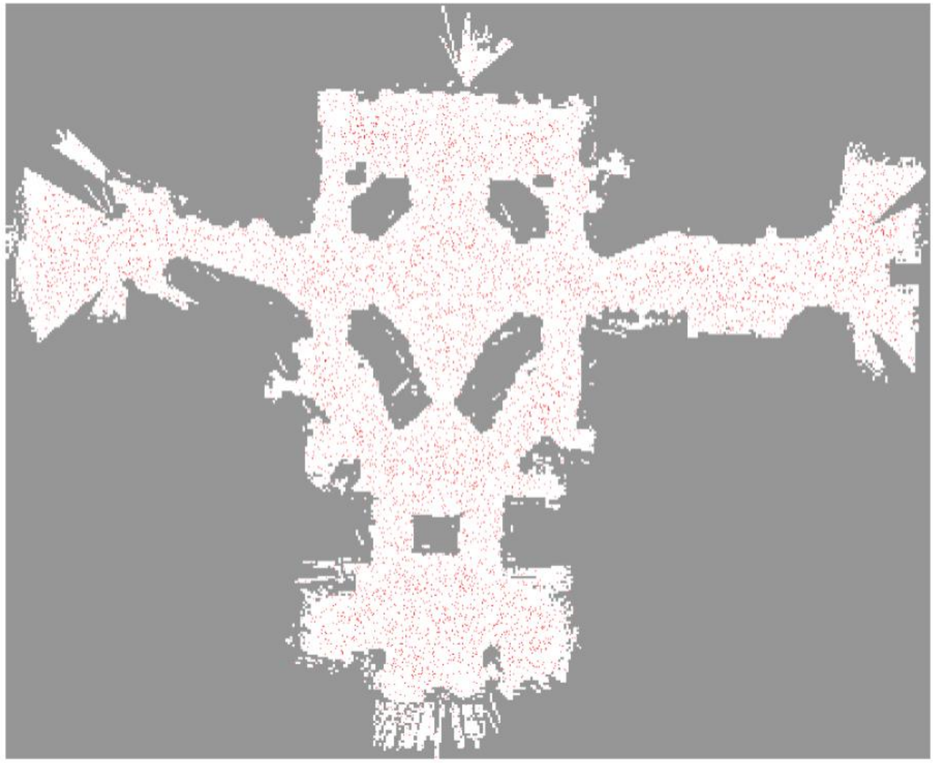
Adaptive Monte Carlo localization (AMCL)



Sensor Model

ROS navigation

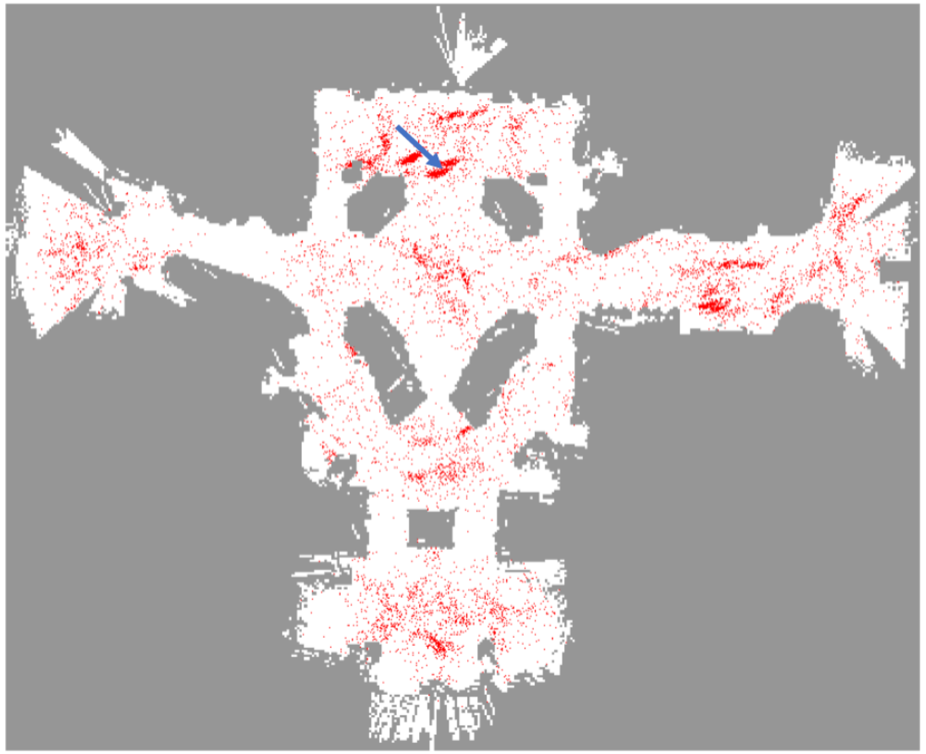
Adaptive Monte Carlo localization (AMCL)



Update
Weight

ROS navigation

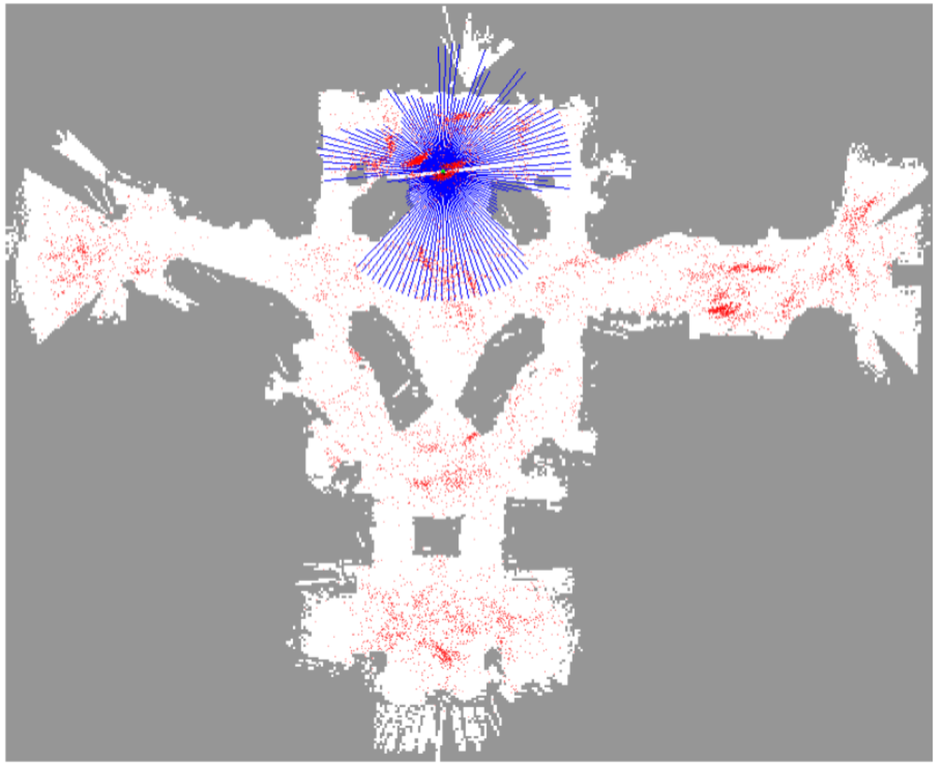
Adaptive Monte Carlo localization (AMCL)



Sampling
Motion Model

ROS navigation

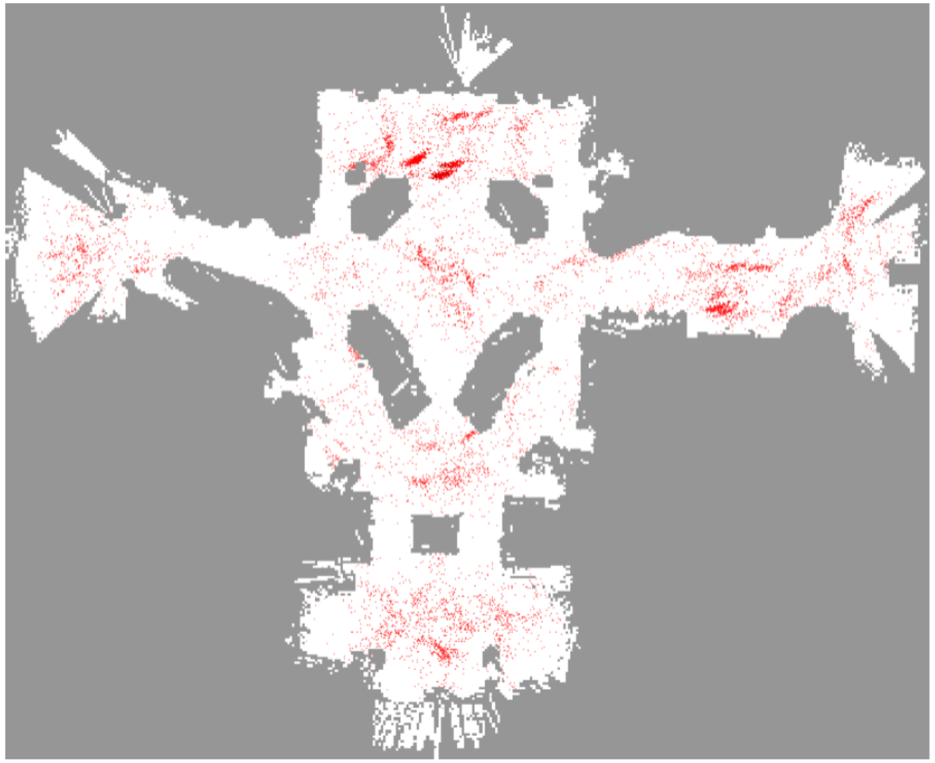
Adaptive Monte Carlo localization (AMCL)



Sensor Model

ROS navigation

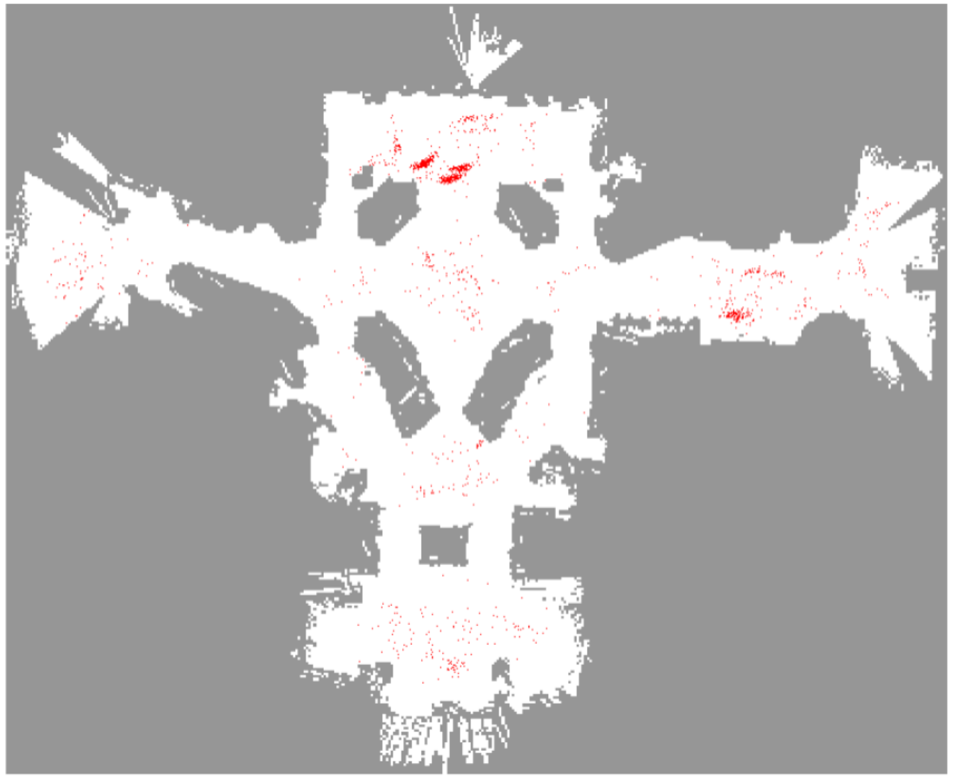
Adaptive Monte Carlo localization (AMCL)



Update
Weight

ROS navigation

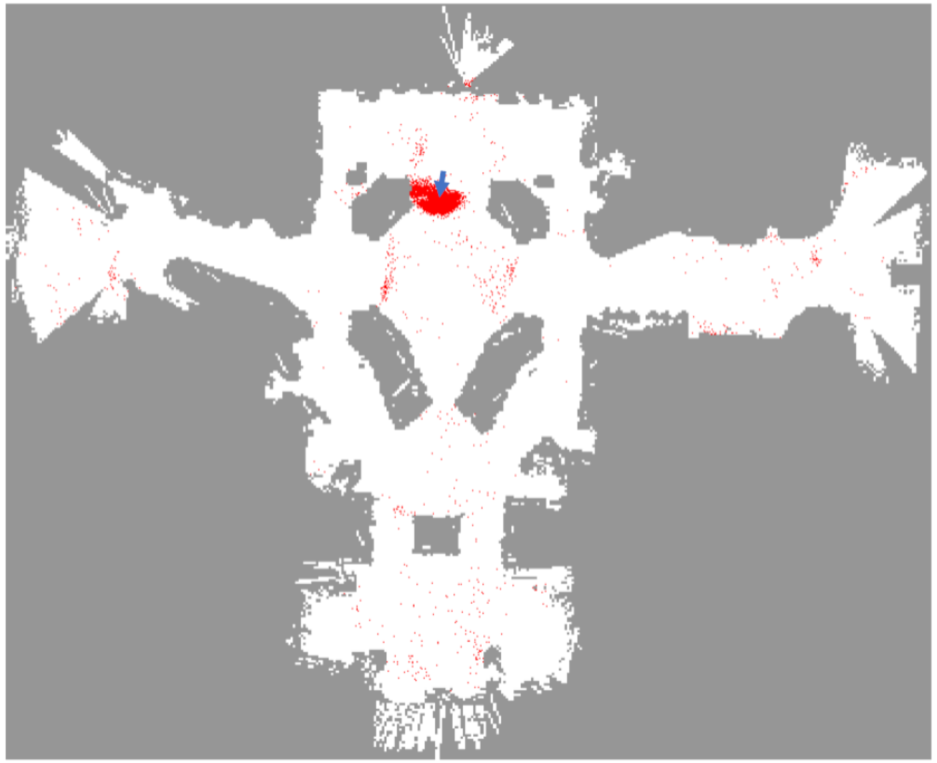
Adaptive Monte Carlo localization (AMCL)



Resampling
Reset Weight

ROS navigation

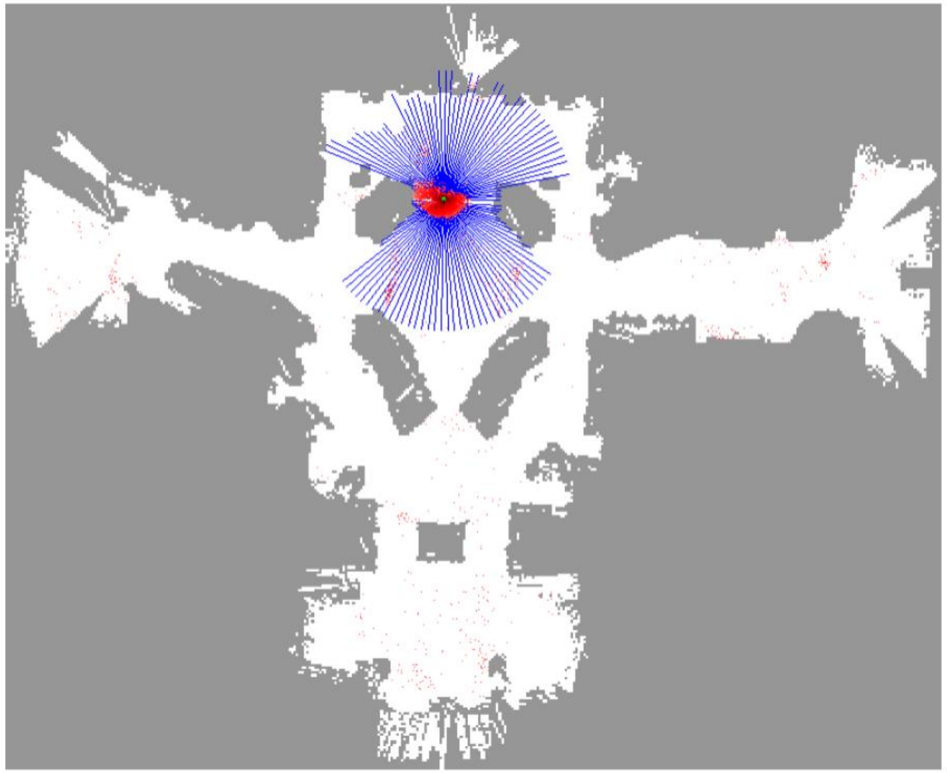
Adaptive Monte Carlo localization (AMCL)



Sampling
Motion Model

ROS navigation

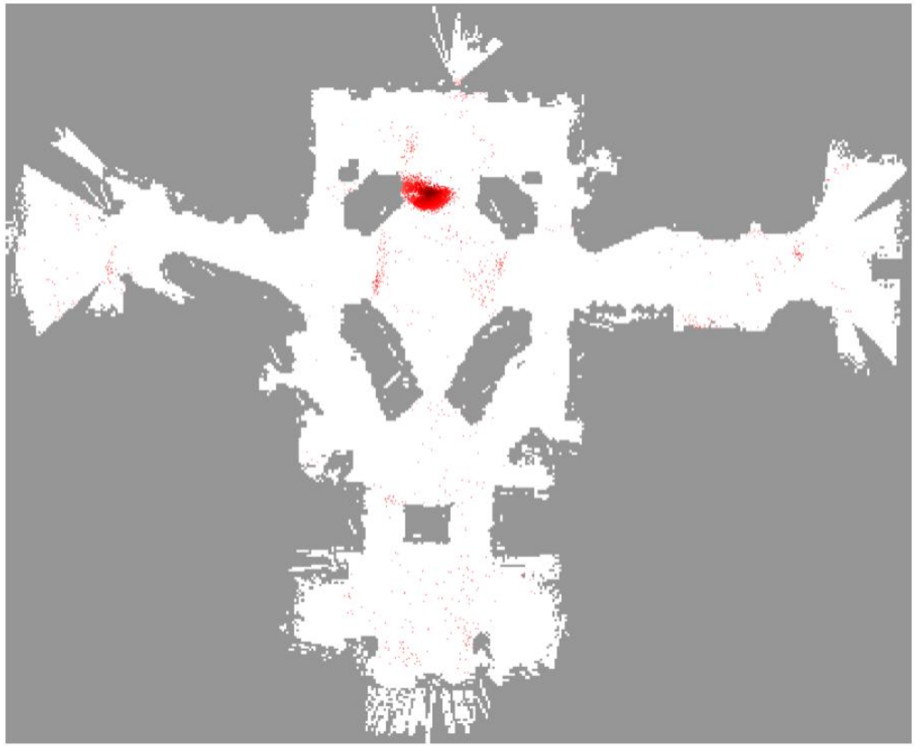
Adaptive Monte Carlo localization (AMCL)



Sensor Model

ROS navigation

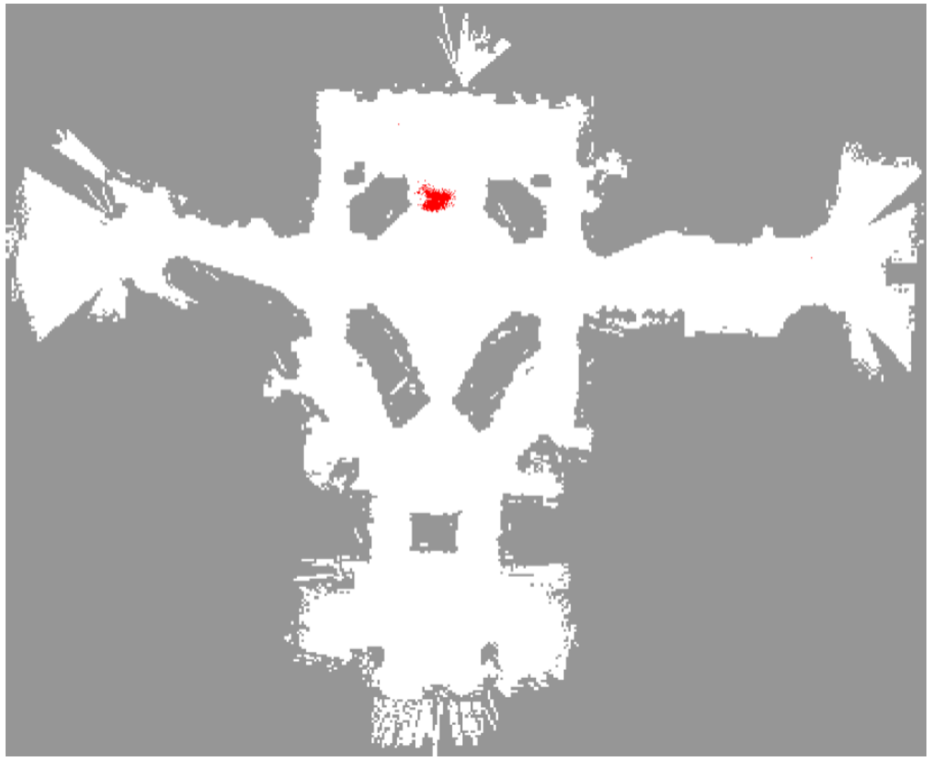
Adaptive Monte Carlo localization (AMCL)



Update
Weight

ROS navigation

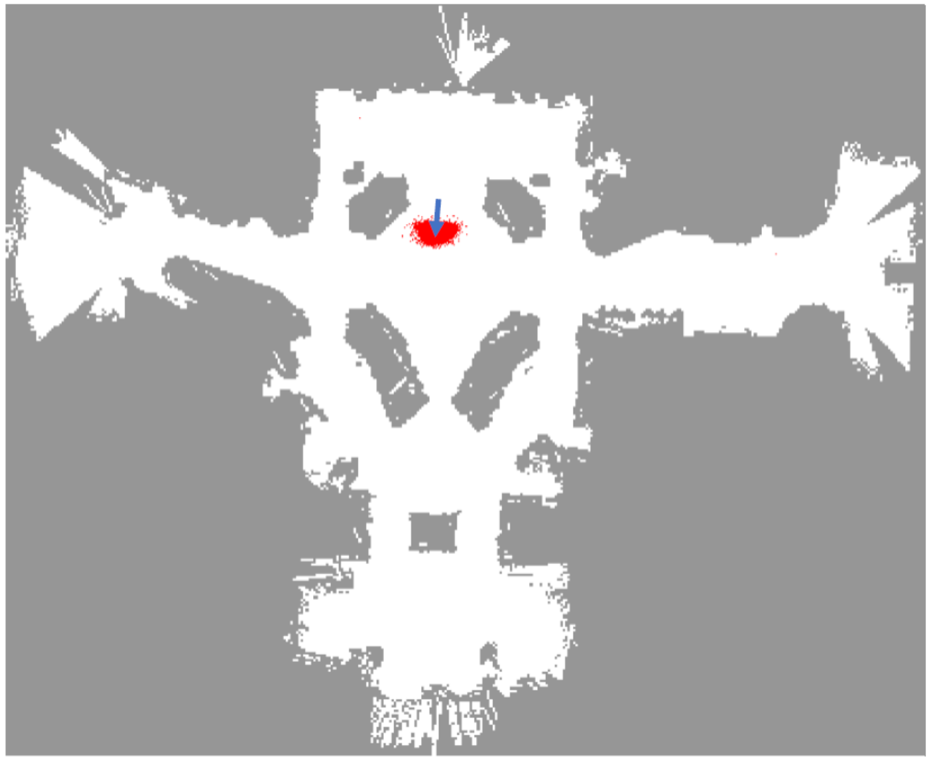
Adaptive Monte Carlo localization (AMCL)



Resampling
Reset Weight

ROS navigation

Adaptive Monte Carlo localization (AMCL)



Sampling
Motion Model

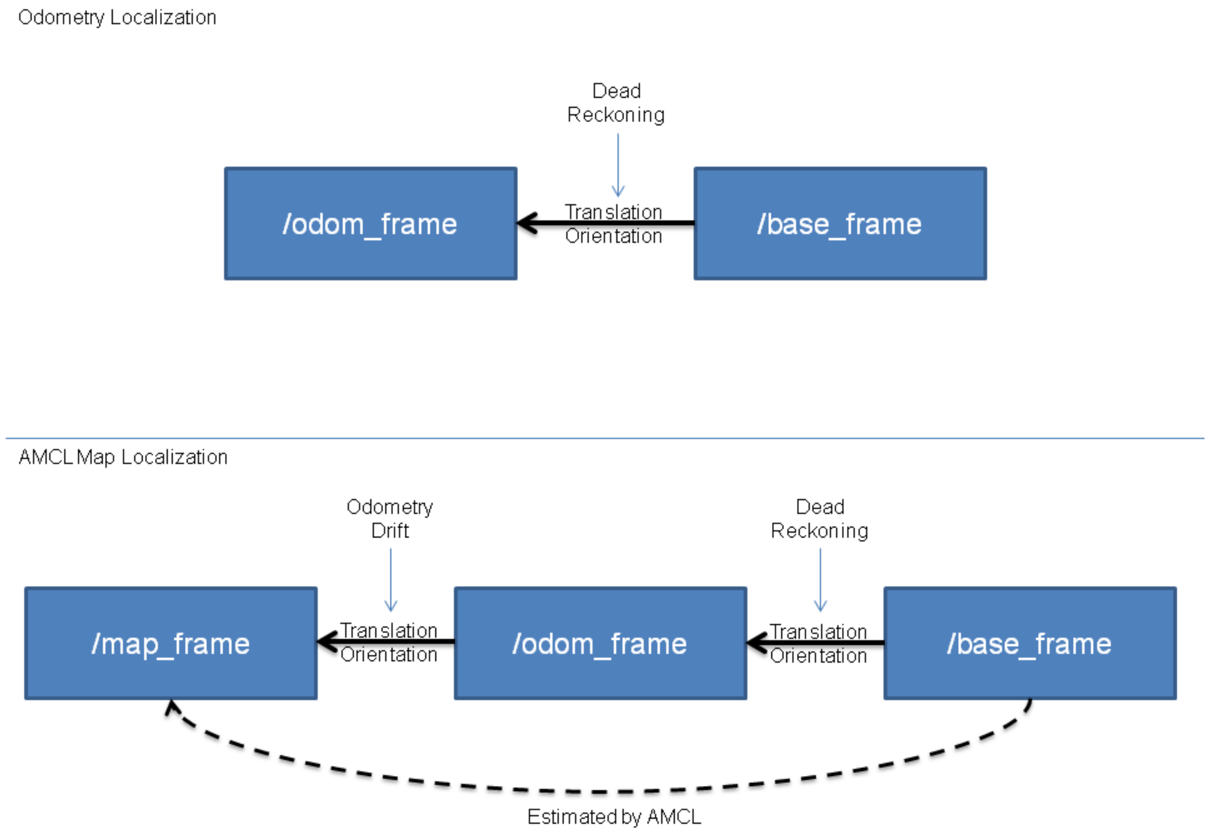
ROS navigation

AMCL [amcl]

amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map.

ROS navigation

AMCL [amcl]


Odometry Localization
Dead Reckoning
/odom_frame
Translation Orientation
/base_frame
AMCL Map Localization
Odometry Drift
Dead Reckoning
/map_frame
Translation Orientation
/odom_frame
Translation Orientation
/base_frame
Estimated by AMCL