LIDAR, Costmaps and Path Planning

ECE 495/595 Lecture Slides

Winter 2017

Instructor: Micho Radovnikovich

Summary and Quick Links

These slides contain the following concepts:

LIDAR Sensors

- **RPLIDAR made by Slamtec:**
  - Range: 6 meters
  - Angular resolution: 1°
  - Field of view: 360°
  - ~$400

- **SICK**
  - Range: 10 meters
  - Angular resolution: 0.25°
  - Field of view: 270°
  - ~$3,000

LIDAR Sensors

- **Hokuyo URG-04LX:**
  - Range: 5 meters
  - Angular resolution: 0.36°
  - Field of view: 240°
  - ~$1,100

- **Hokuyo UTM-30LX**
  - Range: 30 meters
  - Angular resolution: 0.25°
  - Field of view: 270°
  - ~$5,000

LIDAR Sensors

- **Pepperl-Fuchs R2000:**
  - Range: 30 meters
  - Angular resolution: 0.014°
  - Field of view: 360°
  - ∼\$6,000

LIDAR Data

- 2D LIDAR scans are typically reported as an array of range measurements, where the index of the array indicates the angle of the particular beam.

- The distances and their corresponding angles can then be easily converted into Cartesian coordinates.

- In ROS, the *sensor_msgs/LaserScan* message is used to transmit LIDAR data from a driver node to any other nodes in the system.

LIDAR Data

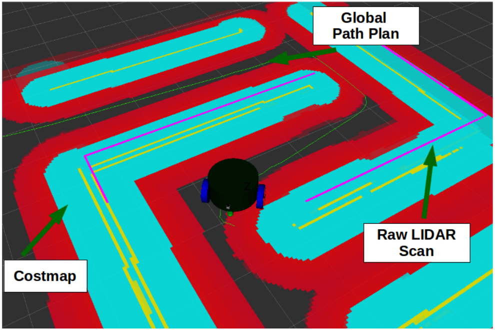
- The *sensor_msgs/LaserScan* message contains information to completely represent a LIDAR scan.
- *angle_min*, *angle_max* and *angle_increment* are used to convert the *ranges* array index into an angle.
- If the particular LIDAR sensor measures it, the intensity of the reflected laser can be specified in the *intensities* array.

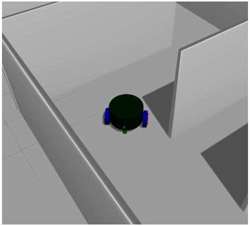
```
micho@vm: ~
micho@vm:~$ rosmsg show LaserScan
[sensor_msgs/LaserScan]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

ROS Navigation Stack

- Built into ROS is a collection of tools known as the *Navigation Stack*.

- The source code is freely available on GitHub: https://github.com/ros-planning/navigation.git





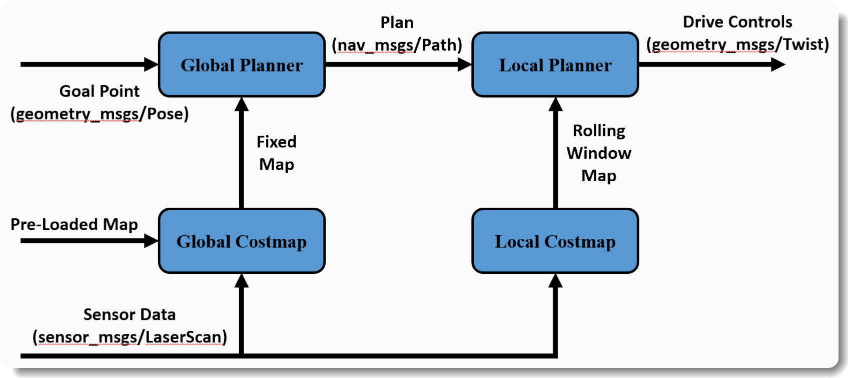
ROS Navigation Stack

- The Navigation Stack is comprised of several ROS packages. Some of them are described here:
  - **costmap_2d** – This package allows the user to configure *costmaps*, which are used by path planners to determine space that is safe to travel through.
  - **navfn** – This package provides a global planner, which is responsible for plotting a route through a costmap from the current position of the robot to a goal point.

ROS Navigation Stack

- Navigation Stack packages, cont.
  - **base_local_planner** – This is a local planner, which is responsible for following the global plan while also performing reactionary obstacle avoidance.
  - **amcl** – Augmented Monte Carlo Localization (AMCL) is a particle filter-based method for localizing a robot on a known map using LIDAR data.
- The entire Navigation Stack is designed for low-speed, circular, differential drive vehicles. Therefore, application of the Navigation Stack to other types of vehicles and environments may require different core algorithms.

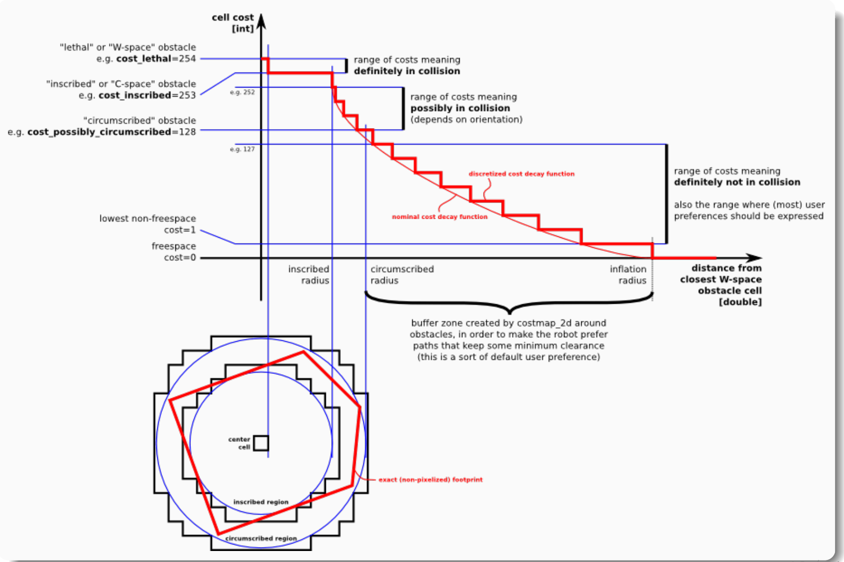
ROS Navigation Stack



2D Costmaps

- The *costmap_2d* package is used to configure costmaps to be used by the global and local planners.
- Full documentation is available at http://wiki.ros.org/costmap_2d
- Costmaps are arrays of cells whose data values indicate whether the particular cell is occupied or free.
- Costmap cells range in cost value from 0 to 255, where certain values represent special information.

2D Costmaps



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
e.g. 252
e.g. 127
lowest non-freespace cost=1
freespace cost=0
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

2D Costmaps

- Meanings of special values and ranges of cost:
  - **Unknown space (255)** – No sensor data has been observed for this cell.
  - **Lethal (254)** – This cell contains an obstacle. No part of the robot can enter this cell.
  - **Inscribed (253)** – This cell is within the inscribed radius of the robot from a lethal cell. Never plan through this cell.
  - **Circumscribed (128 — 252)** – This cell is within the circumscribed radius of the robot from a lethal cell. Only plan through this cell if absolutely necessary.

2D Costmaps

- Lethal, inscribed and circumscribed costs are set from the raw laser scan data along with user-specified parameters describing the footprint of the robot.

- Additional nonzero cost can be added to the costmap to discourage planners from getting uncomfortably close to objects. This is known as *inflation*.

- Inflation is specified by an exponential decay factor, and a radius.

2D Costmaps

- When using the navigation stack, it is typical to configure two costmaps: one global and one local.
- The global costmap is the size of the anticipated navigation area of the robot, and is fixed relative to a particular TF frame.
- The local costmap is much smaller, and is fixed relative to the vehicle's base TF frame.

2D Costmaps

- Global costmaps retain all information added to them by a laser scan, and can also be initialized with a saved map.
- Local costmaps operate like a sliding window, where any information added to them is forgotten when it falls off the edge of the map.
- The particular behavior of a costmap is set using ROS parameters.

2D Costmaps

- Example parameters for a global costmap

```
global_frame: /map
robot_base_frame: /base_footprint
transform_tolerance: 0.2 # secs

update_frequency: 5.0  # Hz
publish_frequency: 2.0  # Hz
rolling_window: false

width: 100     # m
height: 100     # m
resolution: 0.05   # m
origin_x: -50    # m
origin_y: -50    # m
```

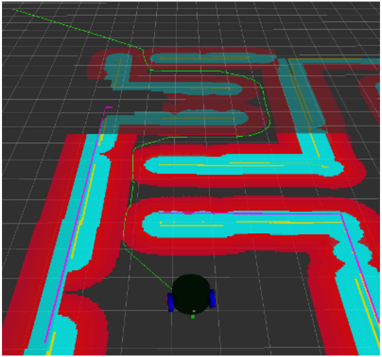
2D Costmaps

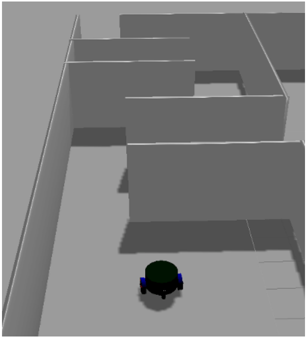
- Example parameters for a local costmap

```
global_frame: /map
robot_base_frame: /base_footprint
transform_tolerance: 0.2 # secs

update_frequency: 5.0 # Hz
publish_frequency: 2.0 # Hz
rolling_window: true

width: 20 # m
height: 20 # m
resolution: 0.05 # m
```

Global Path Planning

- The *navfn* package is the default global planner for the ROS navigation stack.
- Uses current state of costmaps to plan a path to a goal point.





Global Path Planning

- The *navfn* planner finds an optimal path to the goal point using *Dijkstra's Algorithm*.
- Dijkstra's algorithm is designed to find the shortest distance between nodes in a graph.
- In this case, the nodes are the individual costmap cells, and the vertexes are the 4-connected paths between the cells (no diagonals)

Global Path Planning

- The algorithm is broadly summarized as follows:
  - Start and goal points are converted to the corresponding cell array indices.
  - Graph is initialized with the current state of the costmap.
  - Algorithm builds hypothetical paths until one reaches the goal point, at which point the algorithm stops.
  - The method of searching through the graph guarantees an optimal solution path.

Global Path Planning

- From the starting node, the algorithm marks each of its neighbors as an active node.
- Each active node assigns scores to each of its neighbors that haven't been explored yet. The assigned score is equal to the score of the particular node, plus a constant value.
- Any node assigned a score is marked as active for the next iteration, unless it is an occupied node.
- Iteration completes when the goal node is explored, and the minimum-scored path from the start is the output path.
- This gif animation from Wikipedia illustrates the algorithm iteration process nicely: Wiki page GIF

Global Planning

- Another algorithm that is widely-used for path solving is the *A** algorithm.
- The idea is very similar to Dijkstra's algorithm, but it adds a *heuristic* function to the score of the nodes.
- For A*, the heuristic function is simply the distance to the goal node.
- A* was invented in the late 1960's and early 1970's at the Stanford Research Institute. It was used for path planning by Shakey, one of the first ever autonomous robots.

Global Planning

- Because of the goal distance heuristic function, A* tends to be a more *depth-first* search focused algorithm, whereas Dijkstra's algorithm is more of a *breadth-first* search.
- Depth first searches tend to find a solution faster than breadth first algorithms, but can sometimes find a sub-optimal one.
- Having to keep track of the goal distance of each node also causes A* to require more memory.
- More details about A* here: Wiki page GIF

Global Planning

- These are the default values of th ROS parameters that affect the behavior of the navfn planner:

```
allow_unknown: true
planner_window_x: 0.0
planner_window_y: 0.0
default_tolerance: 0.0
visualize_potential: false
```

  - **allow_unknown** – Lets the planner plan through unknown cost cells.
  - **planner_window** – Constrains the planner to stay within a certain window.
  - **default_tolerance** – Lets the planner plan close to the goal instead of being perfectly precise.
  - **visualize_potential** – Constructs a point cloud to visualize the potential function of the algorithm.

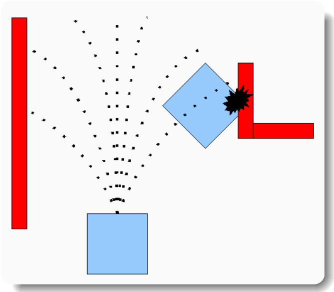
Local Path Planning

- The *base_local_planner* package is the default local planner for the ROS navigation stack.
- Full documentation is available at http://wiki.ros.org/base_local_planner
- Uses the costmaps and the output path from the global planner to compute drive control commands for the robot.

Local Path Planning

- The local planner in the *base_local_planner* package generates a sample of different trajectories and then assigns a cost to each one.

- The trajectories are generated by selecting discrete sets of speed and yaw rate commands, and constructing a trajectory based on each permutation of the sets.



Local Path Planning

- A trajectory's cost function is simply the addition of three components, which can be tuned by setting ROS parameters.
- The *pdist_scale* parameter value is multiplied by the distance from the end point of a trajectory and added to the cost.
- The *gdist_scale* parameter value is multiplied by the distance from the end point to the goal point and added to the cost.
- The *occdist_scale* parameter value is multiplied by the largest value costmap cell that the trajectory passes through and is added to the cost.

Local Path Planning

```
sim_time: 1.0
vx_samples: 3
vtheta_samples: 10
controller_frequency: 20.0

min_vel_x: 0.1
max_vel_x: 0.7
min_vel_theta: -1.0
max_vel_theta: 1.0
acc_lim_x: 2.5
acc_lim_theta: 3.2

gdist_scale: 0.8
pdist_scale: 0.8
occdist_scale: 0.01

global_frame_id: /map
holonomic_robot: false
```

Local Path Planning

- After computing the cost of each individual trajectory, the one with the lowest cost is selected.
- The particular speed and yaw rate corresponding to the selected trajectory are sent to the drive control system to make the vehicle actually folllow.
- The drive control system would then convert the requested speed and yaw rate commands into individual wheel speeds and command the motors accordingly.