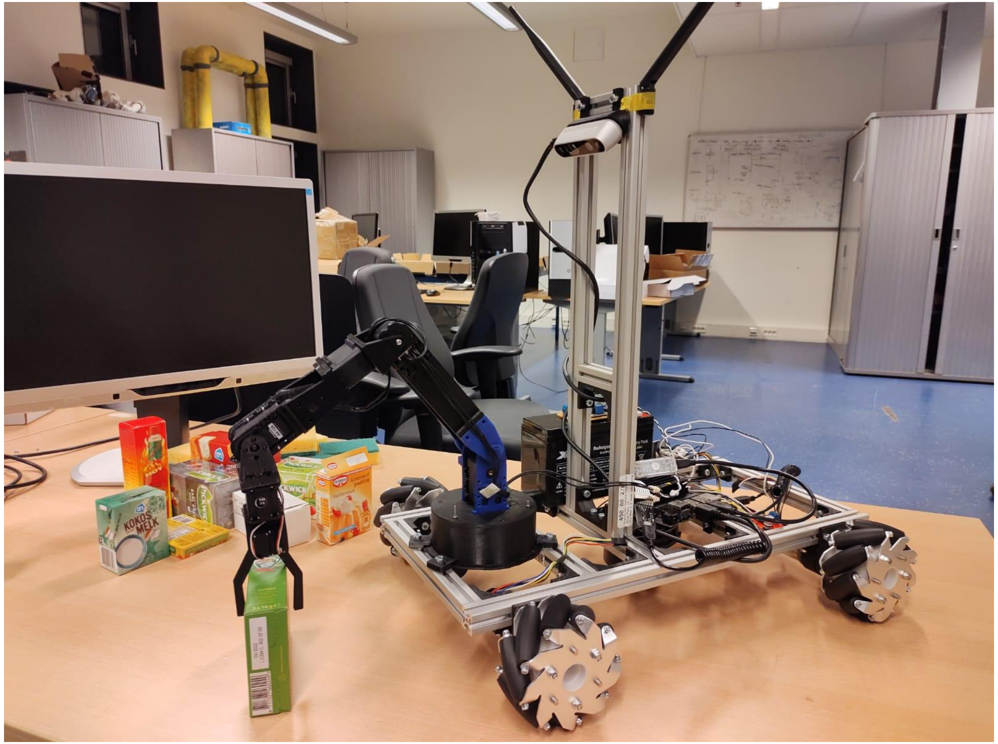
Robotics for AI

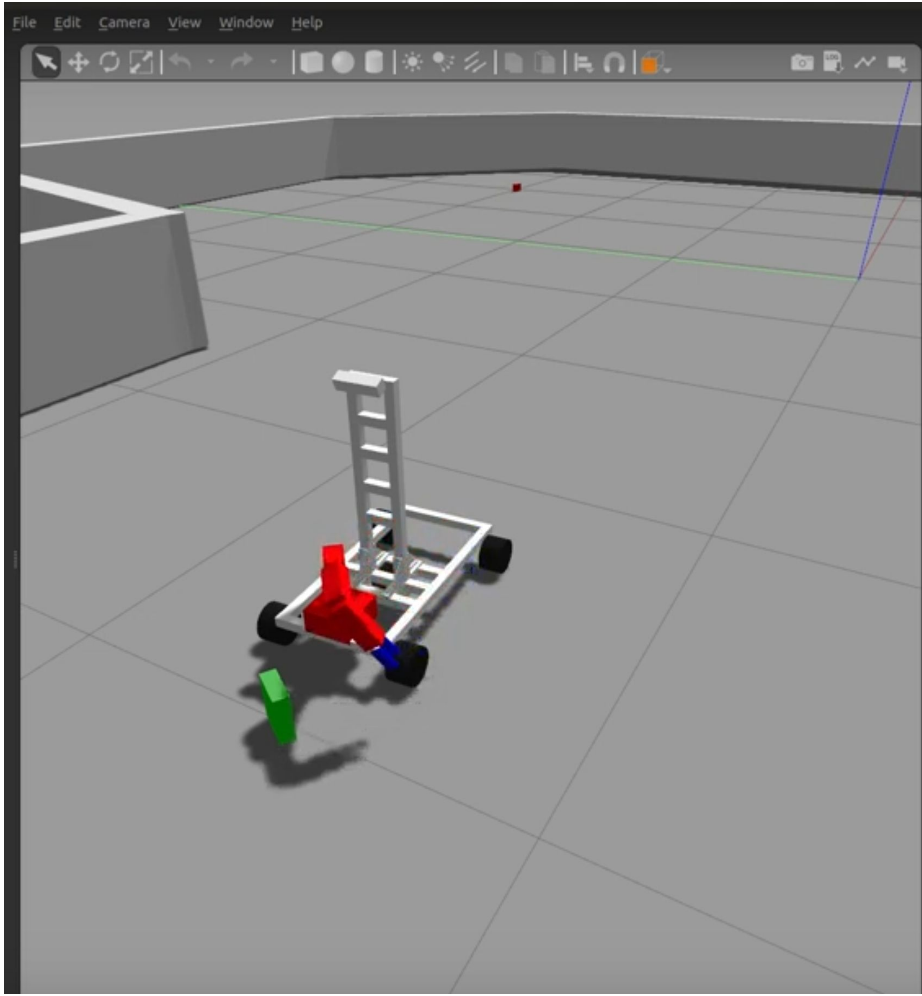
Development lecture 2: Autonomous Navigation

Table of content:



Gazebo

roslaunch body_gazebo simulation.launch

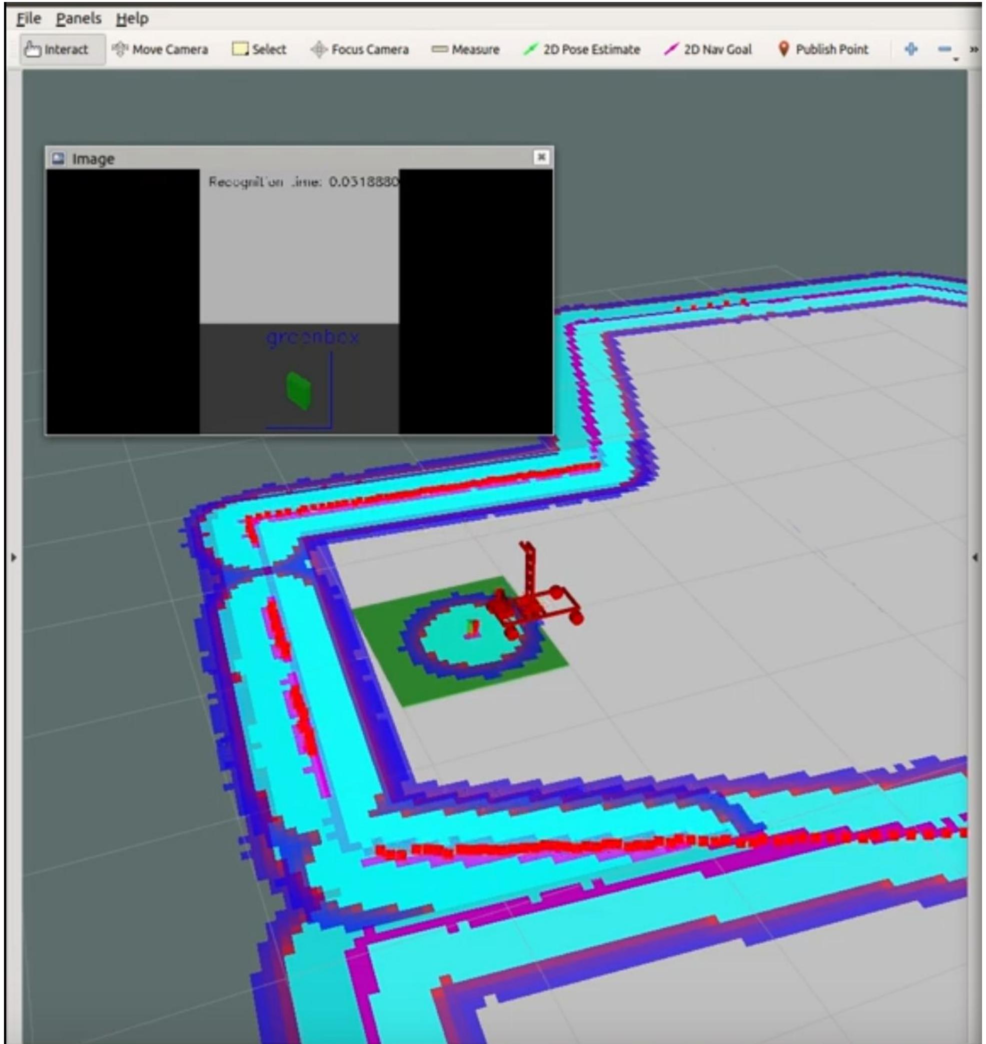


RViz

Visualisation tool for robotics

“See what the robot sees/believes”

In terminal: rviz



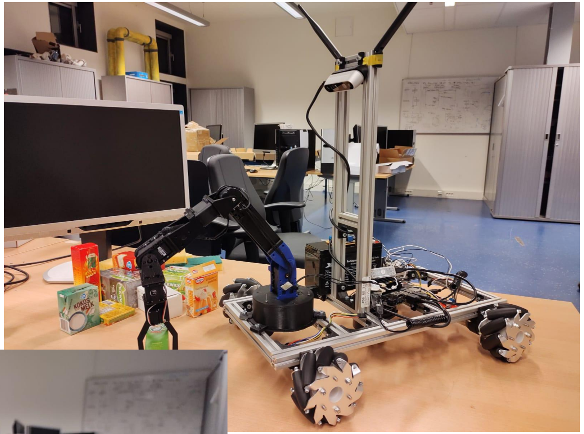
Sensors and actuators for navigation

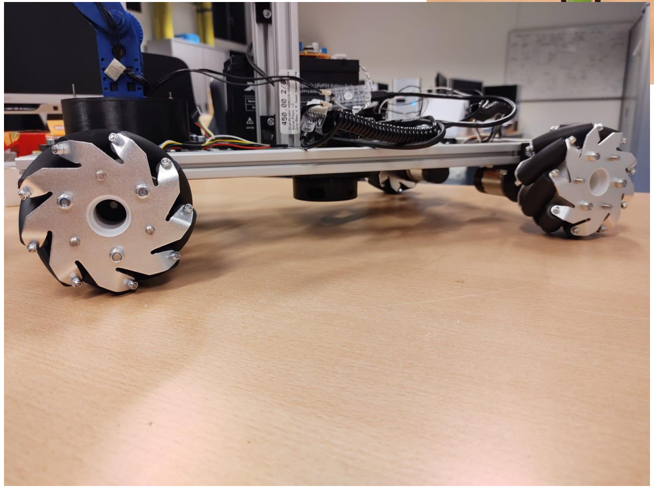
Sensors:

- Realsense camera
- Lidar
- Encoders

Actuators:

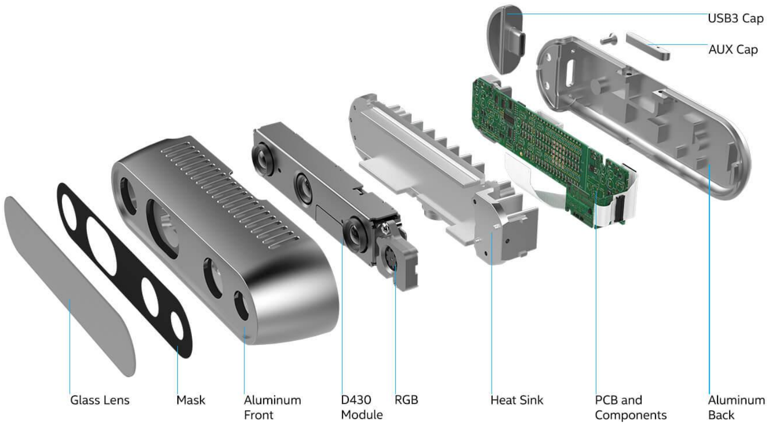
- 4x motor + mecanum wheels

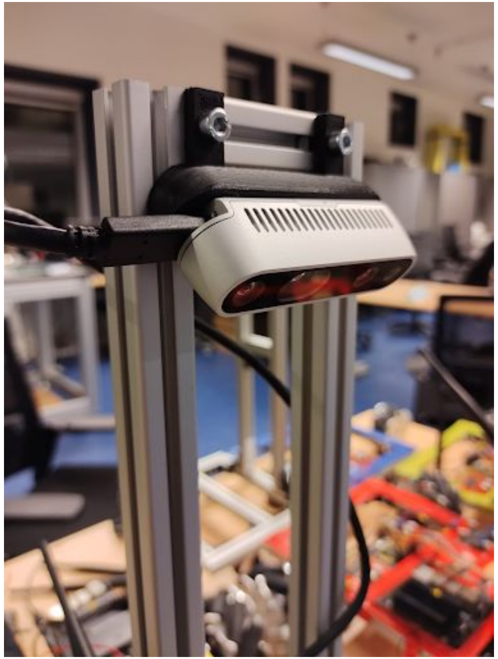




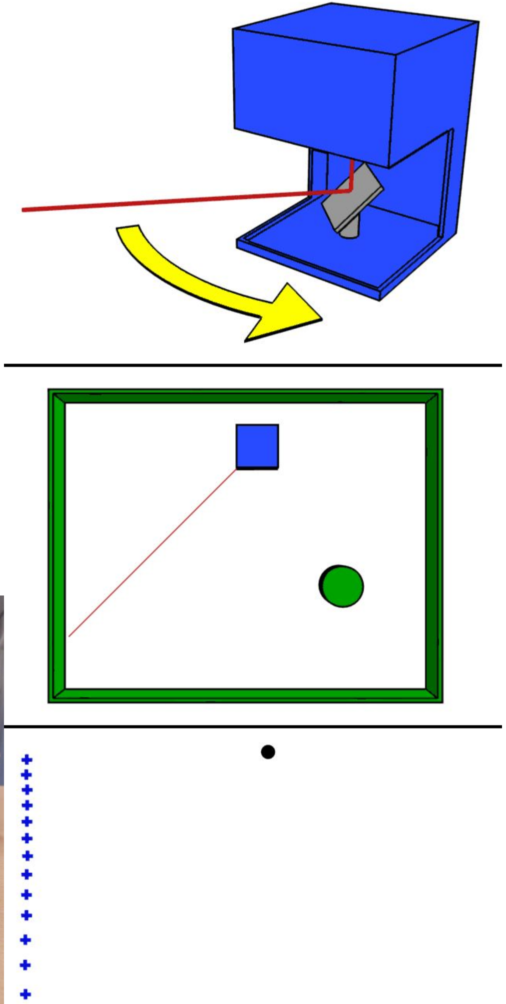
Sensors and actuators for navigation

- Realsense camera
  - Pointcloud data (3D voxels)
  - for every voxel:
    - r,g,b,x,y,z data
  - Not used in sim, to safe computational resources
  - Is used on real robot



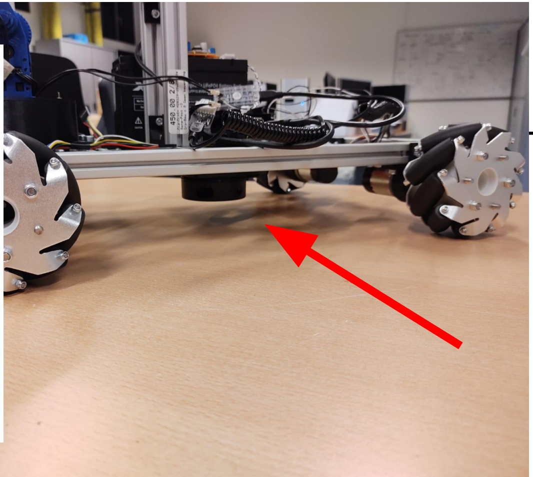


Sensors and actuators for navigation



LIDAR:

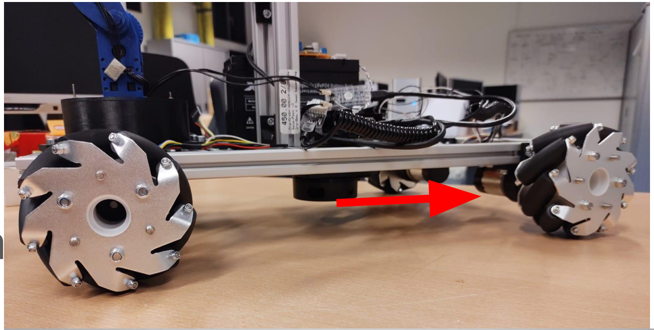
- A rotating laser for distance measurement
- Used for mapping and localisation

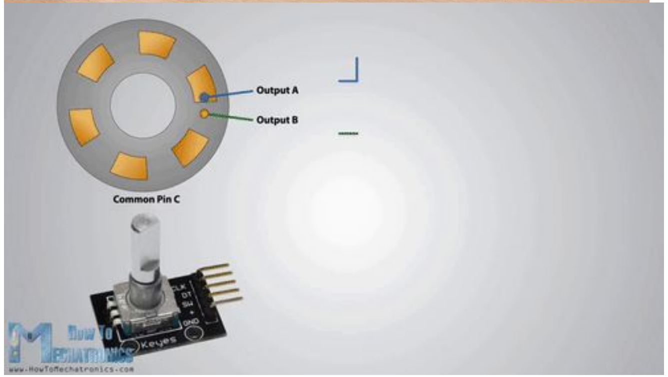


Sensors and actuators for navigation

Encoders:

- 64 ticks per motor revolution
- ~1:100 gearbox -> 6533 ticks per axis revolution
- Used to measure wheel velocity
- $\omega = (2\pi * nTicks / 6533) / \Delta t$
- $\Delta t = \sim 50$ ms
- provides feedback for PID loop for motor control



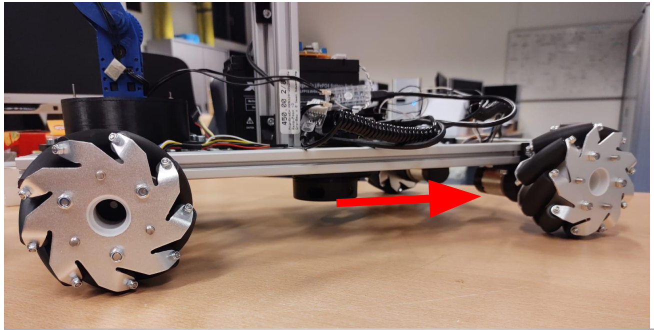


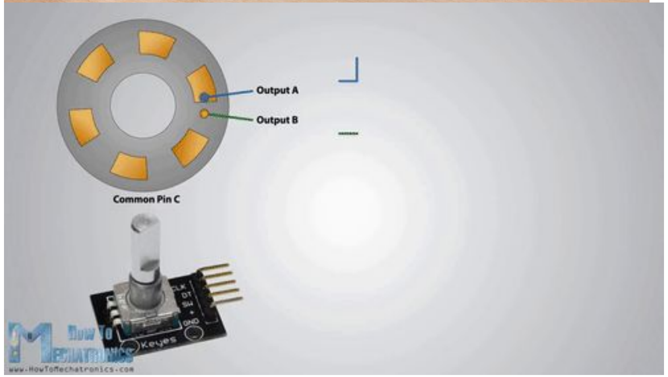
Sensors and actuators for navigation

Robot kinematics

- Robot velocity -> wheel velocity
- Execution of robot velocity commands

```
provided: The desired robot velocities
linear_x (m/s), linear_y (m/s), angular_z (rads/s)

Given:
left_right_distance (m) (The distance between the left and right wheels)
wheel_circumference (m)

First: convert the robot rotation velocity to a tangential velocity:
tangential_z = angular_z * left_right_distance (m/s)

express the required wheel velocities per x,y,theta
x_rot_vel = 2 * pi * linear_x / wheel_circumference (rads/s)
y_rot_vel = 2 * pi * linear_y / wheel_circumference (rads/s)
tangential_rot_vel = 2 * pi * tangential_z / wheel_circumference (rads/s)

omega_front_left_motor = x_rot_vel - y_rot_vel - tangential_rot_vel (rads/s)
omega_rear_left_motor = x_rot_vel + y_rot_vel - tangential_rot_vel (rads/s)
omega_front_right_motor = x_rot_vel + y_rot_vel + tangential_rot_vel (rads/s)
omega_rear_right_motor = x_rot_vel - y_rot_vel + tangential_rot_vel (rads/s)
```

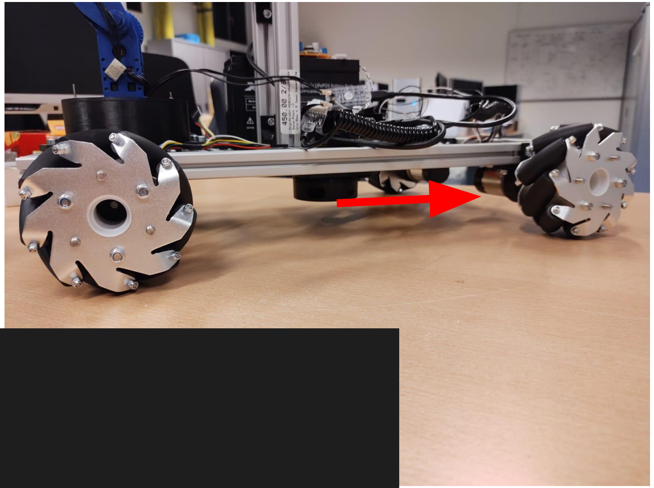




Sensors and actuators for navigation

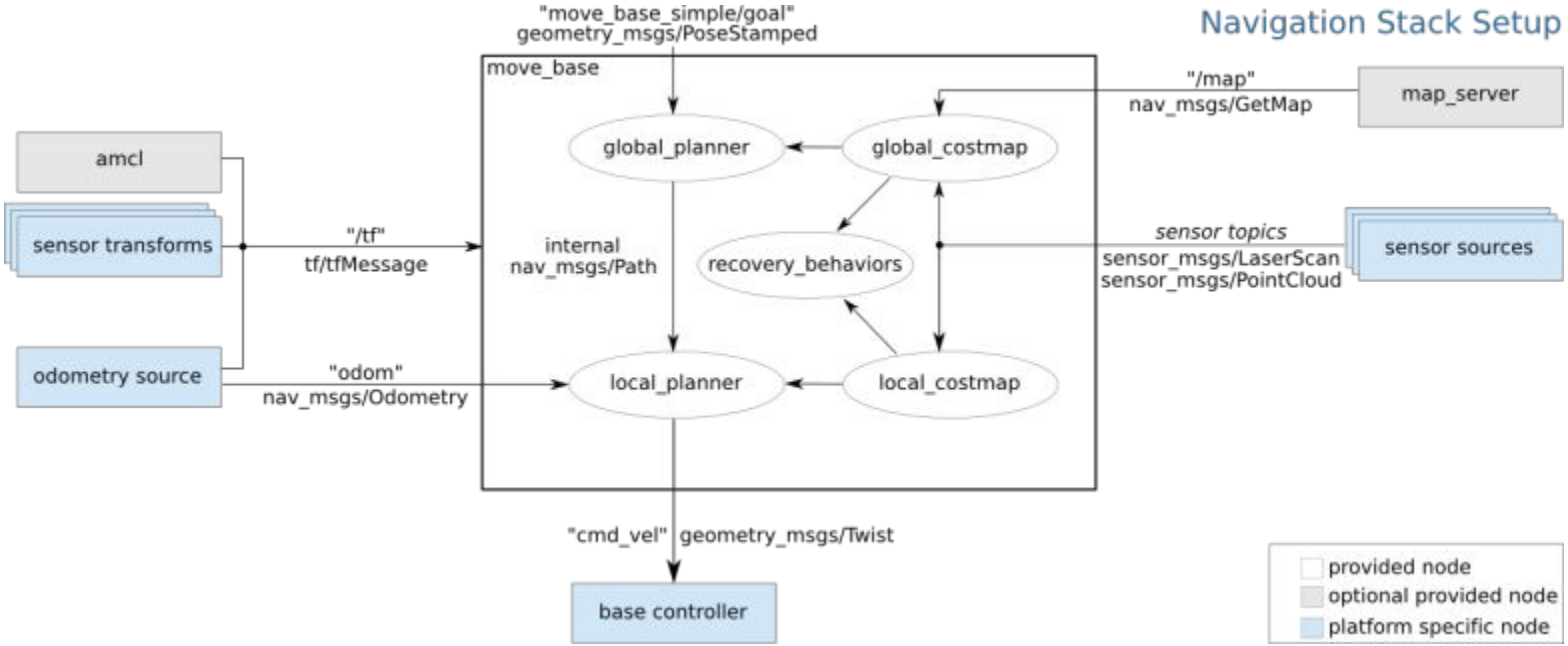
Robot kinematics

- Wheel velocity -> Robot velocity
- Basis for odometry

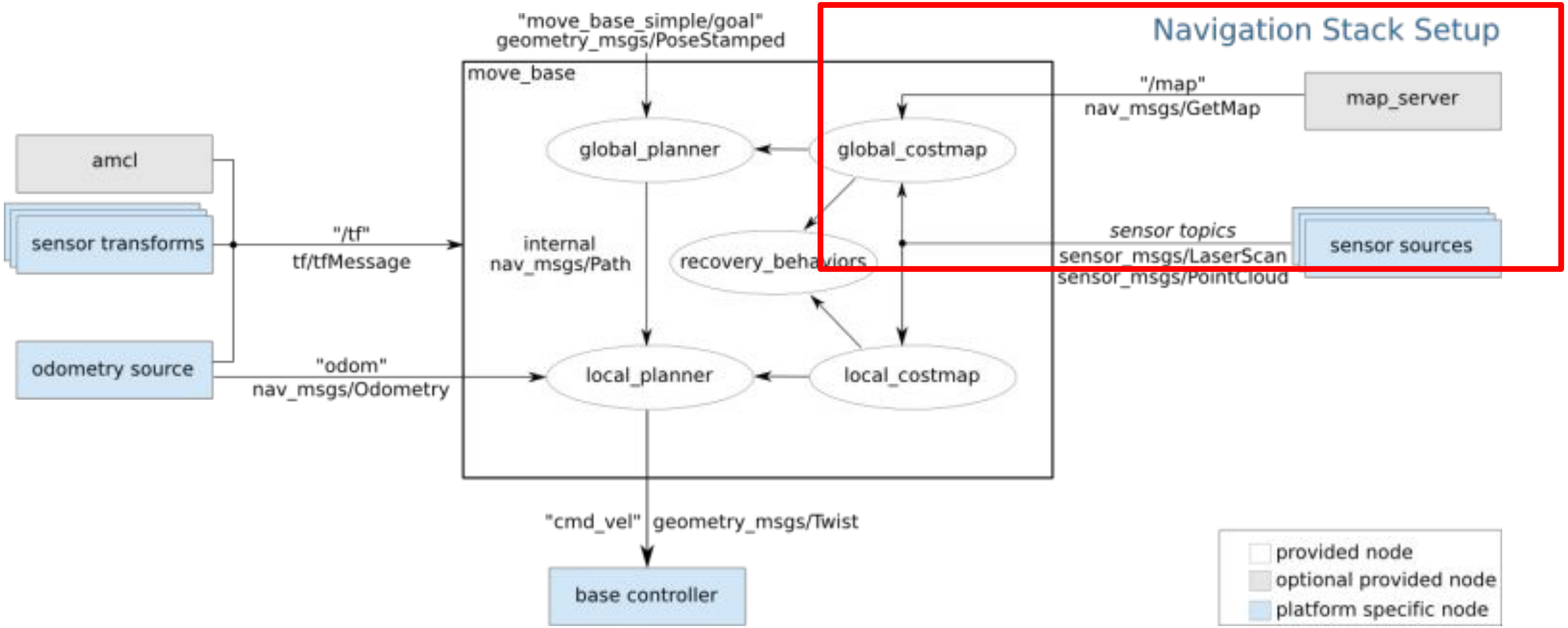


```
provided: The wheel velocities:
omega_front_left_motor (rads/s)
omega_rear_left_motor (rads/s)
omega_front_right_motor (rads/s)
omega_rear_right_motor (rads/s)

Given:
left_right_distance (m) (The distance between the left and right wheels)
front_rear_distance (m) (The distance between the front and rear wheels)
wheel_circumference (m)

average_x_rotation = (omega_front_left_motor + omega_rear_left_motor + omega_front_right_motor + omega_rear_right_motor) / 4
average_Y_rotation = (-1 * omega_front_left_motor + omega_rear_left_motor + omega_front_right_motor - omega_rear_right_motor) / 4
average_tangential_rotation = (-1 * omega_front_left_motor + omega_rear_left_motor - omega_front_right_motor + omega_rear_right_motor) / 4

linear_x = average_x_rotation * wheel_circumference (m/s)
linear_y = average_y_rotation * wheel_circumference (m/s)
angular_z = average_tangential_rotation * wheel_circumference / (0.5 * front_rear_distance + 0.5 * left_right_distance) (rads/s)
```

MoveBase


Navigation Stack Setup
"move_base_simple/goal"
geometry_msgs/PoseStamped
move_base
global_planner
global_costmap
recovery_behaviors
local_planner
local_costmap
internal
nav_msgs/Path
"/map"
nav_msgs/GetMap
map_server
amcl
sensor transforms
odometry source
"/tf"
tf/tfMessage
"odom"
nav_msgs/Odometry
sensor topics
sensor_msgs/LaserScan
sensor_msgs/PointCloud
sensor sources
"cmd_vel"
geometry_msgs/Twist
base controller
provided node
optional provided node
platform specific node

MoveBase



Navigation Stack Setup
"move_base_simple/goal"
geometry_msgs/PoseStamped
move_base
"/map"
nav_msgs/GetMap
map_server
amcl
global_planner
global_costmap
sensor transforms
"/tf"
tf/tfMessage
internal
nav_msgs/Path
recovery_behaviors
sensor topics
sensor_msgs/LaserScan
sensor_msgs/PointCloud
sensor sources
odometry source
"odom"
nav_msgs/Odometry
local_planner
local_costmap
"cmd_vel"
geometry_msgs/Twist
base controller
provided node
optional provided node
platform specific node

MoveBase

Making a map with laser detections & odometry (Gmapping)

- Starting at map coordinates (x,y,theta) = (0,0,0)
- While driving around manually: (teleoperation)
    - Update the current location in the map using odometry data / AMCL
    - Use LIDAR data to observe obstacle data w.r.t. the robot
    - Add these obstacle detection to the map, using the robot location data.

Make sure that you leave no “unobserved” areas in the map!

MoveBase

Making a map with laser detections & odometry (Gmapping)

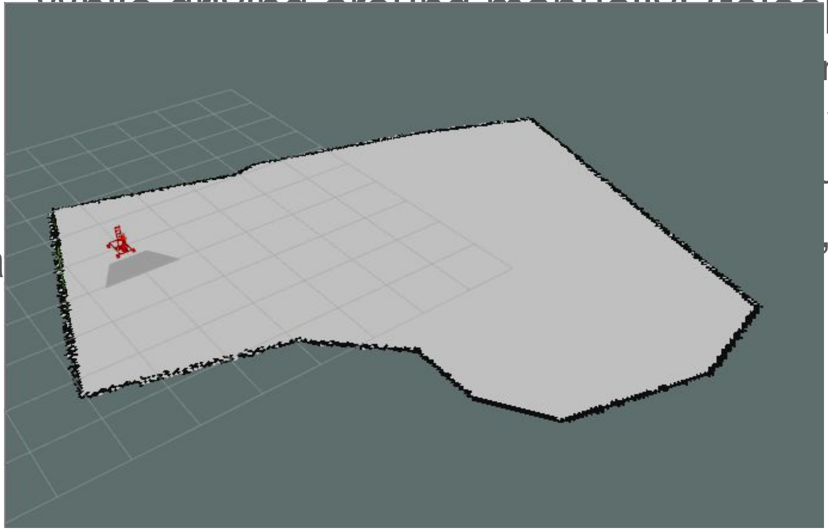
- Starting at map coordinates (x,y,theta) = (0,0,0)
- While driving around manually: (teleoperation)

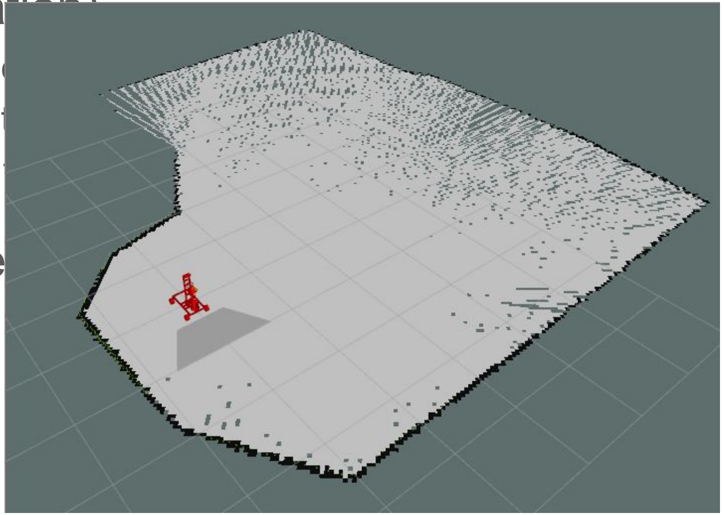
ng od

w.r.t. t

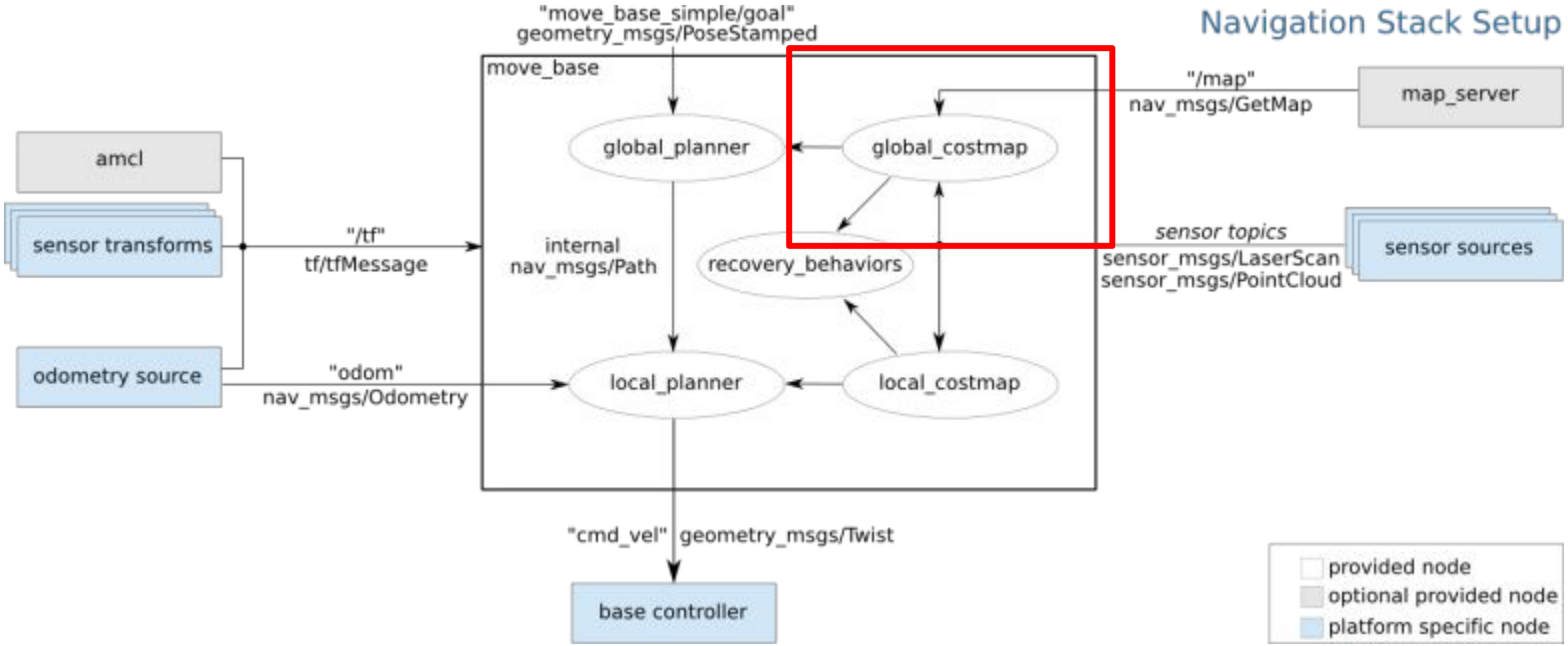
using

Ma ” are





MoveBase



Navigation Stack Setup
"move_base_simple/goal"
geometry_msgs/PoseStamped
move_base
global_planner
global_costmap
"/map"
nav_msgs/GetMap
map_server
amcl
sensor transforms
"/tf"
tf/tfMessage
internal
nav_msgs/Path
recovery_behaviors
sensor topics
sensor_msgs/LaserScan
sensor_msgs/PointCloud
sensor sources
odometry source
"odom"
nav_msgs/Odometry
local_planner
local_costmap
"cmd_vel"
geometry_msgs/Twist
base controller
provided node
optional provided node
platform specific node

MoveBase

GlobalCostmap:

- Cost := obstacle or nearby an obstacle
- Use for planning: Make a plan with minimal cost
- Based on the map + inflation of obstacles

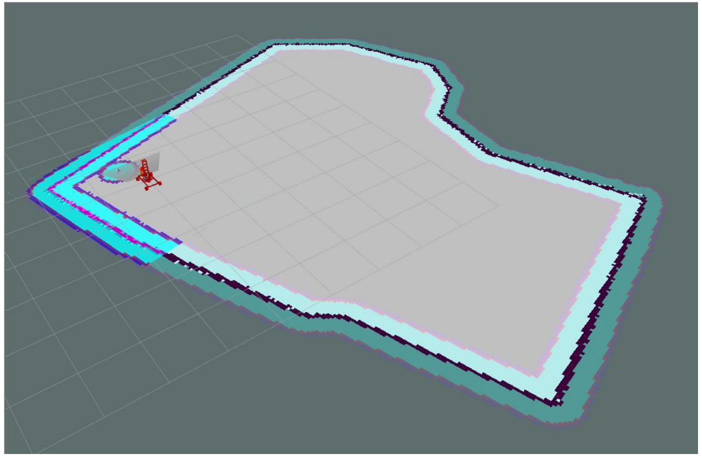
MoveBase

GlobalCostmap:

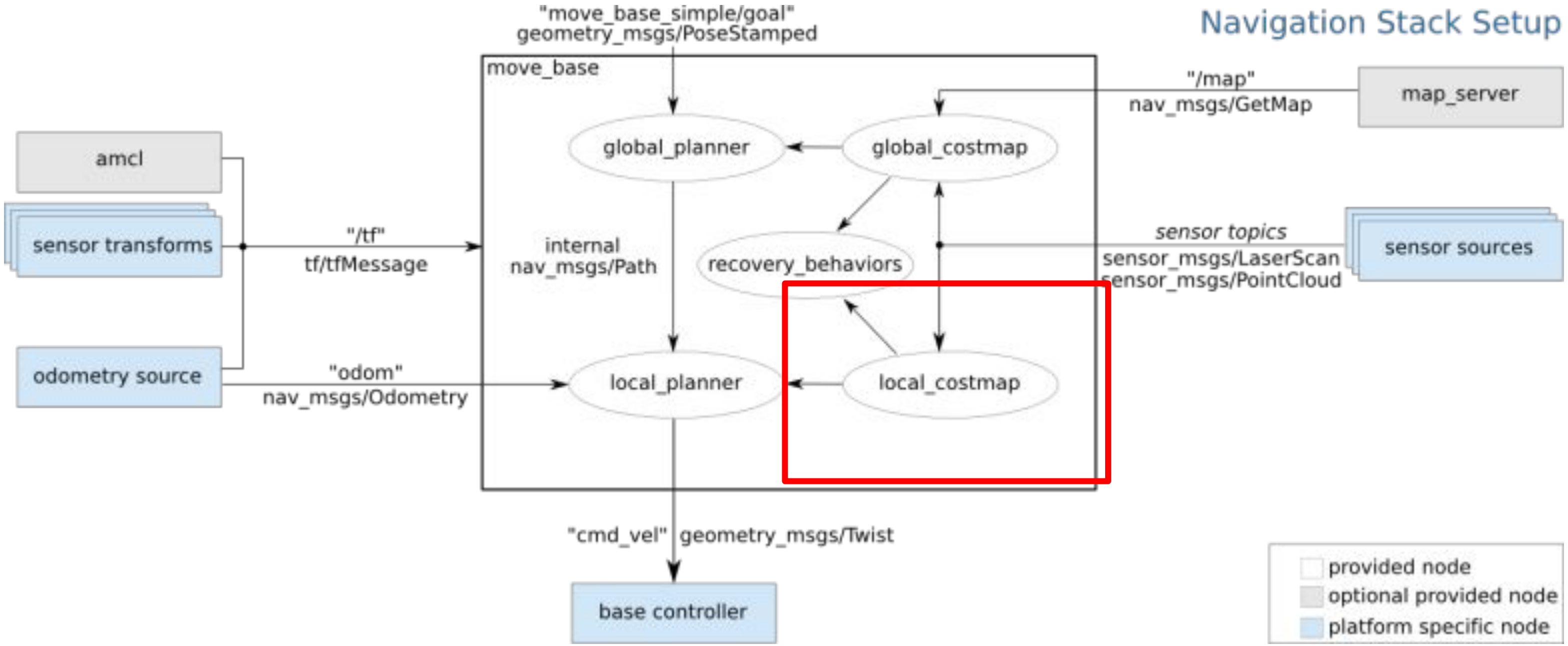
Cost := obstacle or nearby an obstacle

Use for planning: Make a plan with minimal cost

Based on the map + inflation of obstacles



MoveBase


Navigation Stack Setup
"move_base_simple/goal"
geometry_msgs/PoseStamped
move_base
"/map"
nav_msgs/GetMap
map_server
amcl
global_planner
global_costmap
sensor transforms
"/tf"
tf/tfMessage
internal
nav_msgs/Path
recovery_behaviors
sensor topics
sensor_msgs/LaserScan
sensor_msgs/PointCloud
sensor sources
odometry source
"odom"
nav_msgs/Odometry
local_planner
local_costmap
"cmd_vel"
geometry_msgs/Twist
base controller
provided node
optional provided node
platform specific node

MoveBase

LocalCostmap:

- Similar as Global Costmap, but now only live data is used, instead of the map
- If a human walks in the room, this will be captured in the local costmap
- Detections from the local costmap will be added to the global costmap, but will be removed again if the lidar says that that spot is clear again.

- If the costmaps becomes too cluttered for navigating:
    - rosservice call /move_base/clear_costmaps

MoveBase

Lo

live

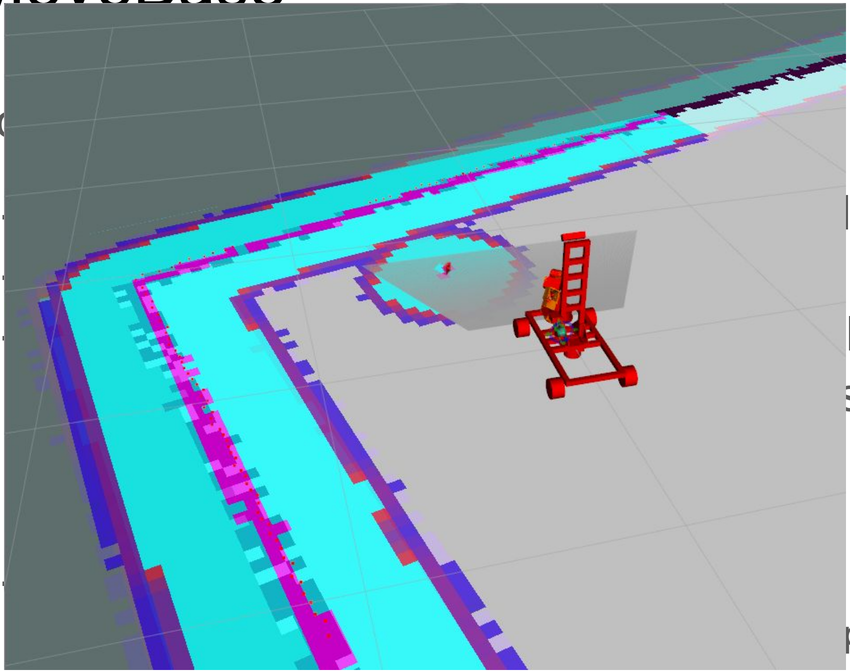
l be

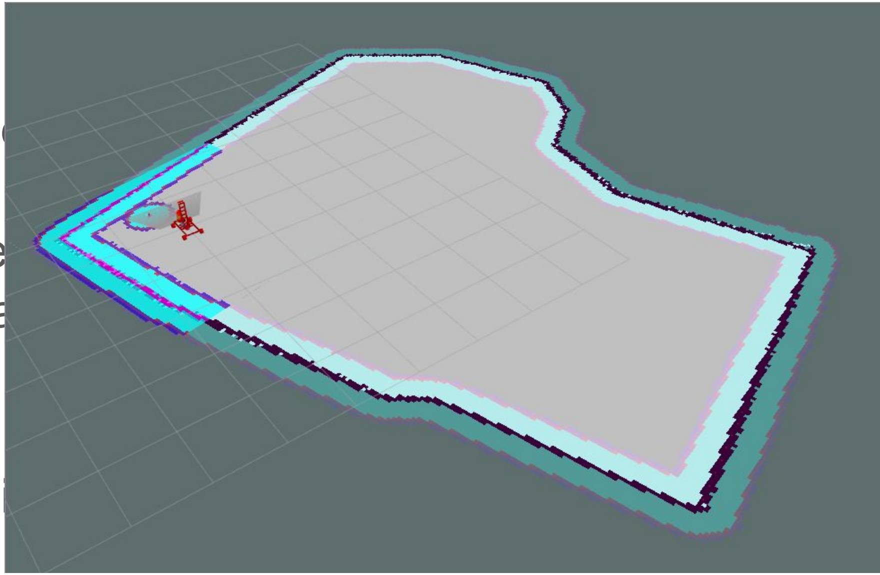
ll be

s tha

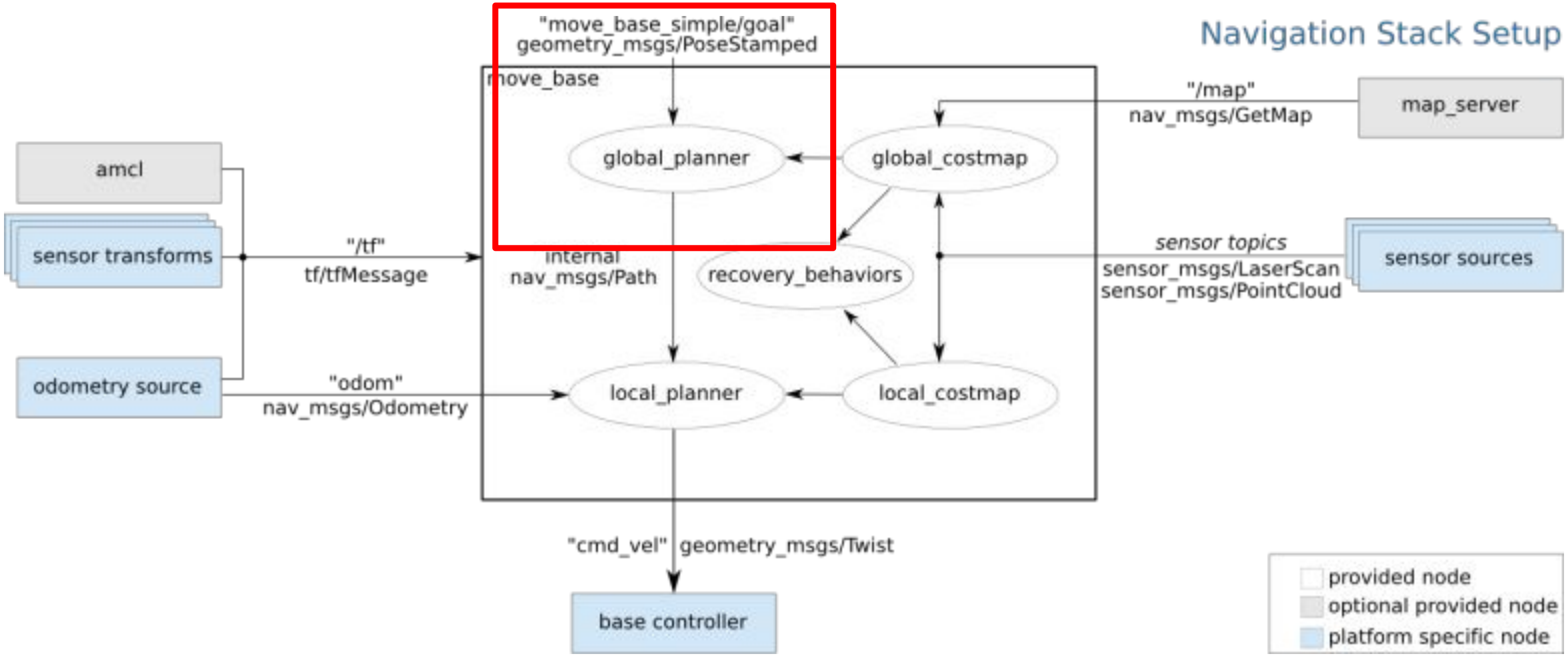
navi

ps





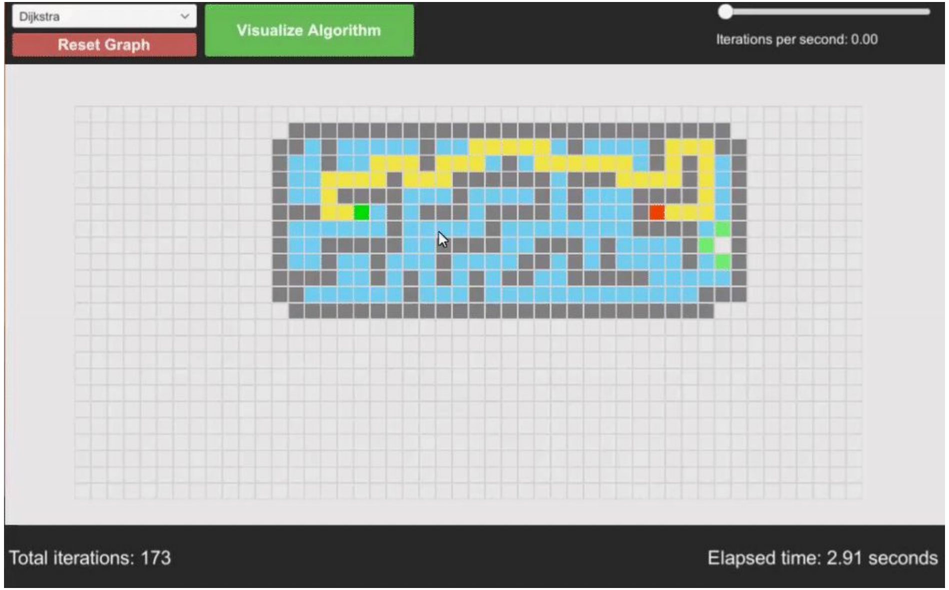
MoveBase


Navigation Stack Setup
"move_base_simple/goal"
geometry_msgs/PoseStamped
move_base
global_planner
global_costmap
"/map"
nav_msgs/GetMap
map_server
amcl
sensor transforms
"/tf"
tf/tfMessage
internal
nav_msgs/Path
recovery_behaviors
sensor topics
sensor_msgs/LaserScan
sensor_msgs/PointCloud
sensor sources
odometry source
"odom"
nav_msgs/Odometry
local_planner
local_costmap
"cmd_vel"
geometry_msgs/Twist
base controller
provided node
optional provided node
platform specific node

MoveBase

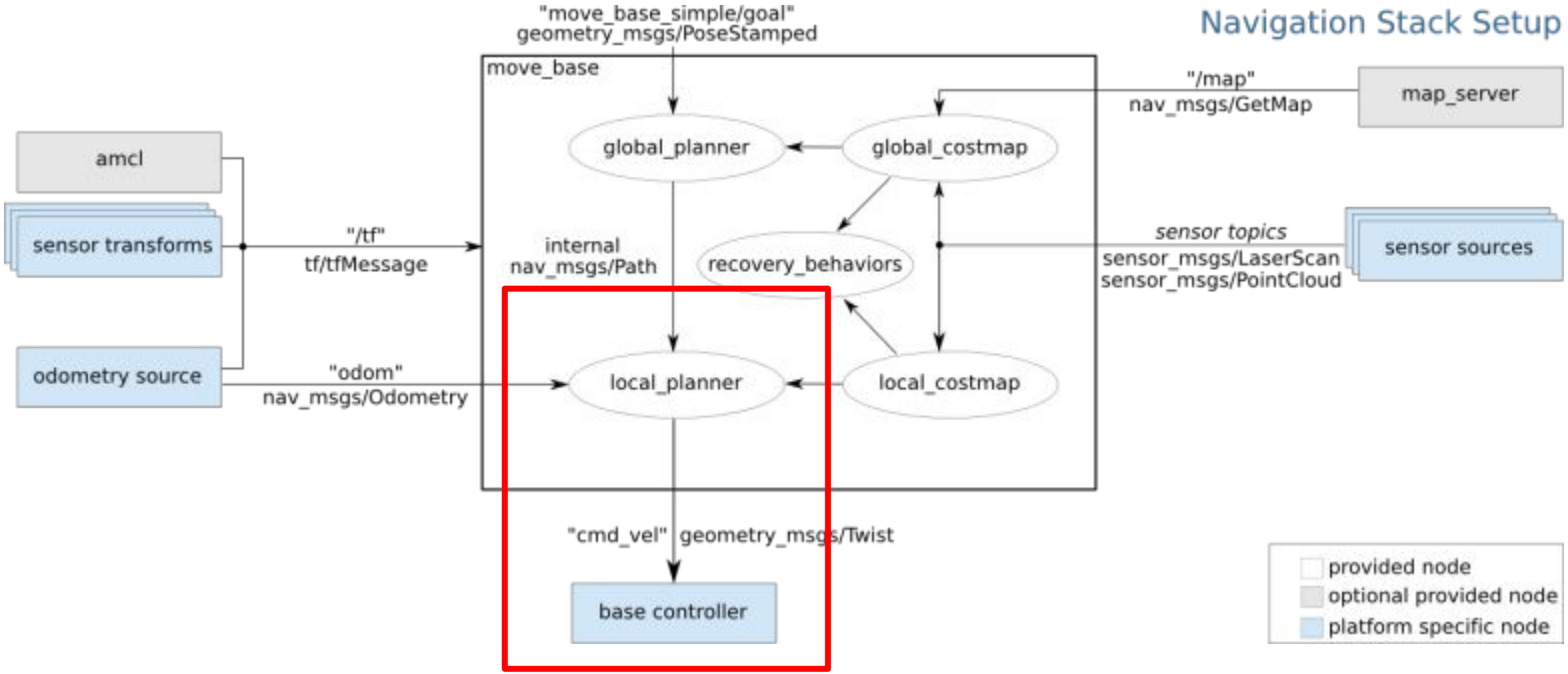
GlobalPlanner:

- Path finding in the GlobalCostmap using Dijkstra's Algorithm.



https://unitylist.com/p/p7z/Unity-Path-Finding

MoveBase


Navigation Stack Setup
"move_base_simple/goal"
geometry_msgs/PoseStamped
move_base
global_planner
global_costmap
"/map"
nav_msgs/GetMap
map_server
amcl
sensor transforms
"/tf"
tf/tfMessage
internal
nav_msgs/Path
recovery_behaviors
sensor topics
sensor_msgs/LaserScan
sensor_msgs/PointCloud
sensor sources
odometry source
"odom"
nav_msgs/Odometry
local_planner
local_costmap
"cmd_vel"
geometry_msgs/Twist
base controller
provided node
optional provided node
platform specific node

MoveBase

LocalPlanner:

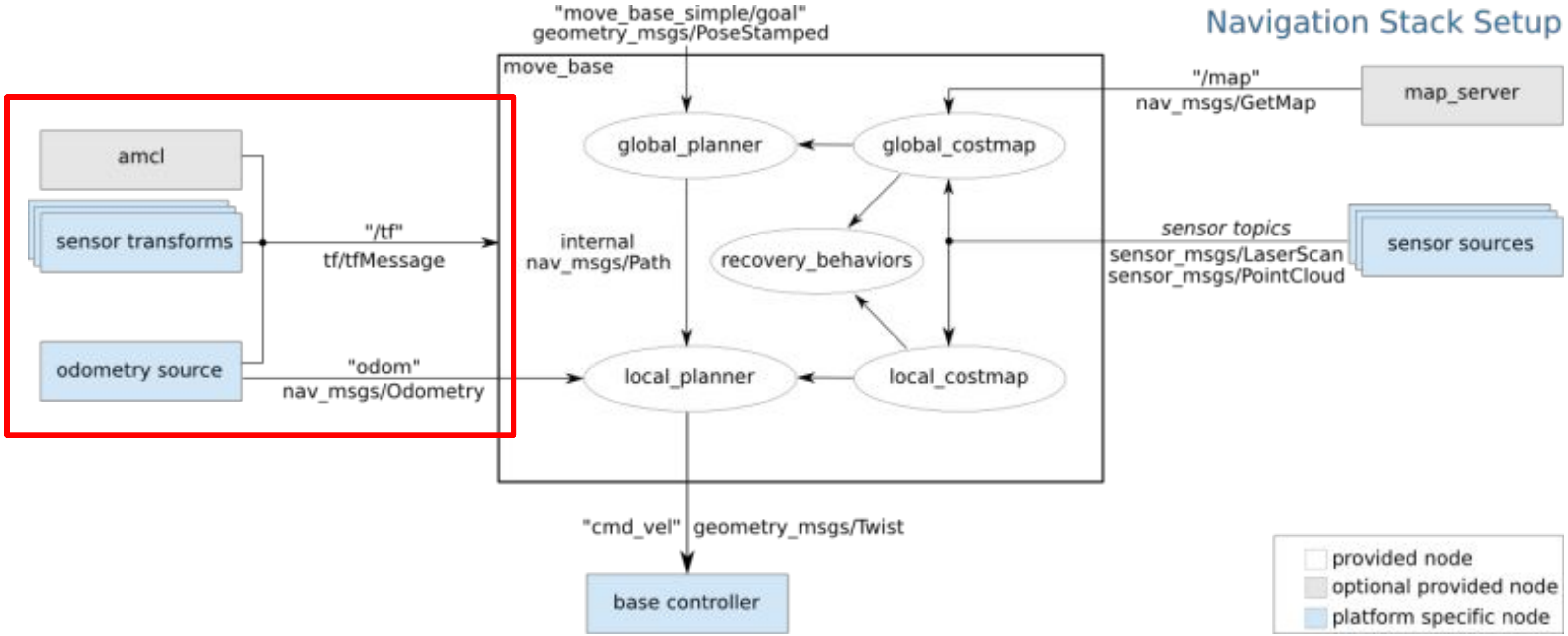
- Make a very local path, with tries to follow the global path planning
- Avoids obstacles that are detected in the local costmap
- Outputs robot velocity commands to the motor driver
- Multiple local planners exist. Here the DWA planner is used.

MoveBase

DWA Planner (Dynamic Window Approach:

- Discretely sample in the robot’s control space (dx,dy,dtheta)
- 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.
- 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)
- Pick the highest-scoring trajectory and send the associated velocity to the mobile base

MoveBase



Navigation Stack Setup
"move_base_simple/goal"
geometry_msgs/PoseStamped
move_base
"/map"
nav_msgs/GetMap
map_server
amcl
global_planner
global_costmap
sensor transforms
"/tf"
tf/tfMessage
internal
nav_msgs/Path
recovery_behaviors
sensor topics
sensor_msgs/LaserScan
sensor_msgs/PointCloud
sensor sources
odometry source
"odom"
nav_msgs/Odometry
local_planner
local_costmap
"cmd_vel"
geometry_msgs/Twist
base controller
provided node
optional provided node
platform specific node

MoveBase

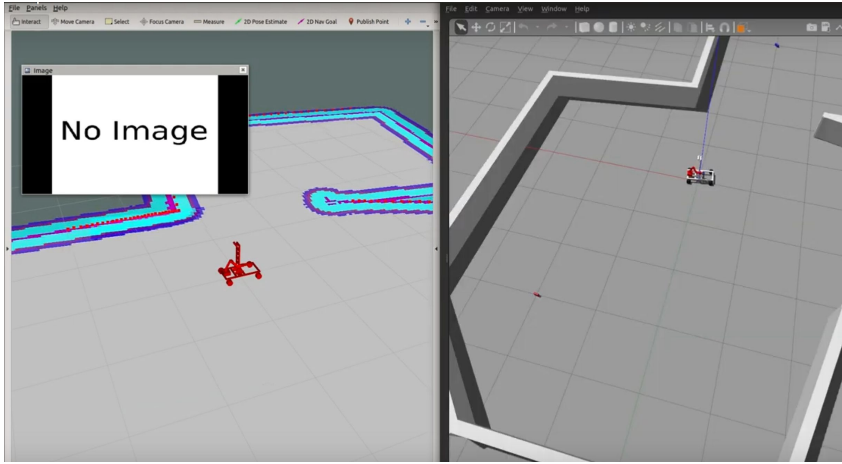
Localisation: (AMCL: Adaptive Monte Carlo Localisation)

- Particle filter approach (simplified explanation)
  - Distribute random particles around map, and consider them as potential locations
  - While running:
    - Update the particle positions with odometry data
    - Compare the measured LIDAR data with the expected LIDAR data
    - Give each particle a likelihood/weight based on the data correlation
    - Redistribute the particles based on these likelihoods/weights
    - Consider the most probable location the robot's location.

The particles will cluster at the most probable location(s) of the robot

Demo video: https://youtu.be/HXVk_BERMqI

Mapping



First, start the simulation environment:

- roslaunch body_gazebo simulation.launch

Then, start the teleop program:

- rosrun keyboard_teleop keyboard_teleop_node

  Now, you can drive the robot around :)

  WASD for lateral movement, QE for rotation, space for braking

Mapping

Then, start the mapping program (gmapping)

- rosrun gmapping slam_gmapping scan:=scan_filtered

Open RViz to inspect your mapping progress:

- rviz

Drive the robot around, until there are no unexplored areas left, then save the map:

- rosrun map_server map_saver -f sim_world

Navigation

After mapping, make sure that you close gmapping and keyboard_teleop, using ‘ctrl + c’ in the respective terminal.

Now, start MoveBase:

- roslaunch navigation move_base.launch

Help the robot to localize itself in the map, using 2D Pose Estimate in RViz, and click + drag a vector in the map at approximately the correct location.

Now, in RViz click on ‘2D Nav Goal’ and provide a vector in the map where the robot should go. Now the robot starts navigating to the goal location.

Navigation

After you gave a ‘2D Nav Goal’, it will show the coordinates + orientation in the terminal where you started RViz.

To define waypoints, write down these coordinates if you’re happy with that location.

Store your final waypoints in a yaml file, so that your behaviour can load them later.

Behaviours

- Make a sub-behaviour for navigation.

- This sub-behaviour gives a navigation goal to the MoveBase action server, and keeps track of the navigation progress using the feedback and result

- The main-behaviour loads the waypoint yaml-file, and passes the required goal to the subbehaviour

The MoveBase Goal

move_base_msgs/MoveBaseGoal Message

File: move_base_msgs/MoveBaseGoal.msg

Raw Message Definition

```
# ====== DO NOT MODIFY! AUTOGENERATED FROM AN ACTION DEFINITION ======
geometry_msgs/PoseStamped target_pose
```

Compact Message Definition

```
geometry_msgs/PoseStamped target_pose
```

http://docs.ros.org/en/fuerte/api/move_base_msgs/html/msg/MoveBaseGoal.html

The MoveBase Goal

geometry_msgs/PoseStamped Message

File: geometry_msgs/PoseStamped.msg

Raw Message Definition

```
# A Pose with reference coordinate frame and timestamp
Header header
Pose pose
```

Compact Message Definition

std_msgs/Header header
geometry_msgs/Pose pose

autogenerated on Sat, 28 Dec 2013 16:52:49

http://docs.ros.org/en/fuerte/api/geometry_msgs/html/msg/PoseStamped.html

The MoveBase Goal

```python
import rospy
from std_msgs.msg import Header
from geometry_msgs.msg import PoseStamped

#a member integer variable to hold count of nr of messages sent
idx = 0

## Assuming some message object 'msg' exists/ Lets say a PoseStamped
msg = PoseStamped()
msg.header.seq = idx  ##This variable might be deprecated. You probably don't need to set it.
idx += 1
msg.header.stamp = rospy.Time.now()

#The frame of reference in which coordinates in the message are expressed
# E.g. for nav waypoints, the "map" frame.
msg.header.frame_id = "/map".
```

std_msgs/Header Message

File: std_msgs/Header.msg

Raw Message Definition

```
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.
#
# sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
# 0: no frame
# 1: global frame
string frame_id
```

Compact Message Definition

```
uint32 seq
time stamp
string frame_id
```

http://docs.ros.org/en/fuerte/api/std_msgs/html/msg/Header.html

The MoveBase Goal

geometry_msgs/PoseStamped Message

File: geometry_msgs/PoseStamped.msg

Raw Message Definition

```
# A Pose with reference coordinate frame and timestamp
Header header
Pose pose
```

Compact Message Definition

std_msgs/Header header
geometry_msgs/Pose pose

autogenerated on Sat, 28 Dec 2013 16:52:49

http://docs.ros.org/en/fuerte/api/geometry_msgs/html/msg/PoseStamped.html

The MoveBase Goal

geometry_msgs/Pose Message

File: geometry_msgs/Pose.msg

Raw Message Definition

```
# A representation of pose in free space, composed of p
Point position
Quaternion orientation
```

Compact Message Definition

geometry_msgs/Point position
geometry_msgs/Quaternion orientation

http://docs.ros.org/en/fuerte/api/geometry_msgs/html/msg/Pose.html

The MoveBase Goal

geometry_msgs/Point Message

File: geometry_msgs/Point.msg

Raw Message Definition

```
# This contains the position of a point in free space
float64 x
float64 y
float64 z
```

Compact Message Definition

float64 x
float64 y
float64 z

http://docs.ros.org/en/fuerte/api/geometry_msgs/html/msg/Point.html

The MoveBase Goal

geometry_msgs/Pose Message

File: geometry_msgs/Pose.msg

Raw Message Definition

```
# A representation of pose in free space, composed of p
Point position
Quaternion orientation
```

Compact Message Definition

geometry_msgs/Point position
geometry_msgs/Quaternion orientation

http://docs.ros.org/en/fuerte/api/geometry_msgs/html/msg/Pose.html

The MoveBase Goal

geometry_msgs/Quaternion Message

File: geometry_msgs/Quaternion.msg

Raw Message Definition

```
# This represents an orientation in free space in quaternion form.

float64 x
float64 y
float64 z
float64 w
```

Compact Message Definition

```
float64 x
float64 y
float64 z
float64 w
```

http://docs.ros.org/en/fuerte/api/geometry_msgs/html/msg/Quaternion.html


That's all Folks!