Pose of the robot represented in RViz by a cloud of conglomerated arrows (green) by /particlecloud

RViz 2D Pose Estimate → click and drag
↳ to calibrate pose
2D Nav Goal → ..

copy → paste
cp /file_directory /file_destination

Robot Configuration ~~syst~~ → URDF (Unified Robot Description Format)
↳ xml describes a robot model
↳ defines its different parts, dimensions, kinematics, dynamics, sensors, ...

Navigation Stack is a set of ROS nodes and algorithms which are used to autonomously move a robot from one point to other, avoiding obstacles the robot finds in its way

↳ Input: current location of the robot
desired location (goal poses)
odometry data (wheel encoders, IMU, GPS, ...)
LIDAR cloudpoints → planar laser
↳ Output: velocity commands

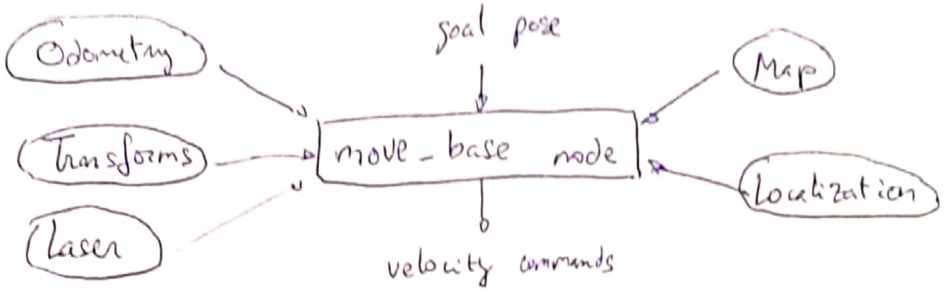
Functional blocks in order to work and communicate with the Navigation Stack

- Odometry source: Gives the robot position with respect to its starting point
The data can contain position and velocity
- Sensor source: Tasks: - localization
- Obstacle avoidance
- Sensor transforms / tf: the data captured by different robot sensors must be referenced to a common frame of reference (base_link) in order to be able to compare data coming from different sensors
- base_controller: convert the output of the navigation stack → geometry_msgs/Twist → motor velocities of the robot

Move_base node

Lo move a robot from its current position to a goal position with the help of navigation nodes

It links the global planner and local planner for the path planning, connecting to the rotate-recovery package if the robot is stuck in some obstacle, and connecting global costmap and local costmap for getting the map of obstacles of the environment



For mapping, in RViz is necessary

- Laser Scan
- Map

} display + Robot Model

- Laser Scan → topic : /kobuki/laser/scan → Size (make thicker red dots)
- Global Options → fixed frame : /odom
- Map → topic : /map
- TF →
- Robot Model

SLAM (Simultaneous Location and Mapping)

Lo building a map of an unknown environment while simultaneously keeping track of the robot's location on the map that is being built

gmapping package

↳ implementation of a specific SLAM algorithm called gmapping

node : /slam_gmapping → create 2D map using { laser, pose } data

data from the laser
transforms of the robot } → occupancy grid map (OGM)

slam_gmapping subscribed to these topics and published to /map topic

/map

↳ nav_msgs / OccupancyGrid → integer ∈ [0, 100] or -1 for unknown value

0 : completely free
100 : occupied

map_server package

↳ node : map_saver → allows you to access the map data from a ROS service and save it into a file

file → YAML : map metadata and image name → .yaml
↳ image : encoded data of the occupancy grid map → .pgm (portable gray map)

command to save the built map at any time → rosed before...

↳ rosrun map_server map_saver -f name_of_map

allows you to give the file a custom name

↳ node : map_server → reads a map file from the disk and provides it to any other node that requests it via ROS service

↑ many nodes request it the current map in which the robot is moving

↳ move_base node → to make → Path Planning
↳ localization node

Service to call in order to get the map

↳ static_map (nav_msgs/GetMap)

topics* that you can connect in order to get a ROS message with the map

↳ map_metadata (nav_msgs/MapMetaData)
↳ map (nav_msgs/OccupancyGrid)

* latched : the last message published to that topic will be saved
latch = True

map_server node
locate before in its directory
Lo ros2 run map_server map_server my_map.yaml

ros2 service call /static_map "{}"

slam_gmapping node → transform each incoming laser reading to the odom frame

Transforms

- If the robot is properly configured, you can trust that the laser readings will be correctly transformed into the Odom frame

In order to use the laser data, we need to tell the robot where (position and orientation) this sensor is mounted in the robot

Lo transform between frames

Center of the robot → /base_link
(rotational center)

For slam_gmapping node, it needs 2 transforms:

- the frame attached to (laser → base_link)
  Lo broadcasted periodically by a robot_state_publisher or static_transform_publisher
- (base_link → odom) → provided by odometry system

Since the robot needs this information anytime, we'll publish it to a transform tree (database)

~~to~~ ros2 run tf2_tools view_frames

If you add a new sensor to a robot, you need to reference it with a frame and its transformation

Lo static_transform_publisher x y z yaw pitch roll frame_id child_frame_id ... period_in_ms

or

```
<launch>
  <node pkg="tf" type="static_transform_publisher" name="name_of_node"
    args="x y z yaw ..." />
</launch>
```

Instead of creating a transform broadcaster, the transforms of the sensors and everything else is contained in the URDF files. But you use that function for the cases:

- If you temporarily add a sensor to the robot
- You have a sensor external from your robot

Slam - gmapping node is highly configurable → launch file parameters

- Parameters
  - base_frame (default: "base_link")
  - map_frame (default: "map")
  - odom_frame (default: "odom")

  } frame attached to

  ↳ ROS Parameter Server
- map_update_interval (default: 5.0)
- Laser
  - maxRange (float) → Set it something higher than the real sensor's maximum range
  - maxUrange (default: 80.0) → laser beams will be cropped to this value
  - minimumScore (default: 0.0) → to consider a laser reading good
- map
  - x min (value: - meters)
  - x max (value: + meters)
  - y min (value: - meters)
  - y max (value: + meters)
- linearUpdate (default: 1.0) → sets linear distance the robot has to move in order to process a laser reading
- angularUpdate (default: 0.5)
- temporalUpdate (default: -1.0) → sets the time in seconds to wait between laser readings (-1.0 turned off)
- particles (default: 30) → # of particles in the filter

You can change the parameters indirectly not ~~using~~ modifying the launch file

↳ parameters are usually loaded from an external file → .yaml

Then in the launch file you add:

<rosparam file = "$(find my_mapping_launcher)/params/sr-rrdy-params.yaml"

command = "load" />

It is possible also to build a map a posteriori the robot scanned its environment.

↳ ROS bag files with the topics: /laser scan
/transforms

↓ in the desired directory

rosbag record -O mylaserdata /laser_topic /tf_topic

↳ remember to kill it after the desired time

then

rosrun gmapping slam_mapping scan:= kobuki/laser/scan

rosbag play mylaserdata.bag

rosrun map_server map_saver -f map_name

slam_gmapping node is only necessary when you are creating a map

POSE = POSITION + ORIENTATION

To see robot pose in RViz

- Pose Array → the cloud shrinks when the certainty increases which happens when the robot gathers more and more data from the environment
- Robot Model
- TF

↳ amcl refine the estimation of the robot's pose
monte carlo location

Monte Carlo location (MCL) or Particle filter localization

Because the robot may always not move as expecte → random guesses generation as to where is going to move next

Guesses → particles → full description of possible future poses

When the robot observes its environment (via sensor readings), it discards particles that don't match and generates more nearby particles

↳ most puticles will converge

The more you move, the more data you will get from your sensors, hence the localization will be more precise.

The AMCL package (Adaptative Monte Carlo location)

↳ node uses the system in order to track the localization in a 2D space

- ↳ it subscribes to the data: of the laser (/scan)
  the laser-based map (/map)
  transformations of the robot (/tf)
- ↳ it publishes its estimated position in the map → {/amcl_pose, /particlecloud}
- ↳ it initialises according to the parameters provided

- Transforms incoming laser scans to the odometry frame
- looks up the transform between the laser's frame and the base frame and it latches forever. It cannot handle a laser scanner that moves with respect to the base

If the number of the particles in the particle cloud is too small, it will cause the location of the robot to be imprecise

Covariance of /amcl_pose → Σ_x (1st value)

↳ if it's lower than 0.65 Σ_y (8th value)

↳ good localization Σ_z (last value)

Initializing the particles through /global_localization might be very useful if you don't have any idea where the robot is

rosservice call /global_localization "{}"

The move_base package

↳ move_base node links all the elements in the navigation process

Goal: move the robot from its current position to a goal position

↳ Simple Action Server that takes a goal pose (geometry_msgs/PoseStamped)

↳ We can send position goal by using Simple Action Client

/move_base/goal

Global Planner

When a new goal is received by move_base node, is sent immediately to the global planner

In charge of calculating a safe path in order to arrive at that goal pose

This path is calculated before the robot starts moving

↳ /plan

You might be interested in just visualizing the global plan without executing it → /make_plan

There exist different global planners

- NavFn (most common)

  Uses Dijkstra's algorithm in order to calculate the shortest path between the initial and goal pose

- Carrot Planner → You can send pose goals in obstacles, then it will adjust the real pose

  Useful for: Moving the robot as close as possible to an obstacle

  Not practical for: Complicated indoor environments

- Global planner is a more flexible replacement for NavFn

  It supports options for A*, tossing quadratic approximation and tossing grid path algorithms

- Based on the costmap, not /map

Costmap represents places that are safe for the robot to be in a grid of cells

Usually the values in the costmap are binary
- free space
- where the robot would be in collision

Each cell of a costmap has an integer $\in [0, 255]$
- 255 (NO_INFORMATION) → Unknown information
- 254 (LETHAL_OBSTACLE) → collision-causing obstacle detected
- 253 (INSCRIBED_INFLATED_OBSTACLE) → No obstacle, but moving the center of the robot to this location will result in a collition
- 0 (~~FREE~~ SPACE)

- global costmap created from a static map
- local costmap created from robot's sensor readings

The global planner uses the global costmap

YAML file for global costmap parameters

- static_map
- rolling_window } opposite boolean
- plugins → developed in common_costmap_parameters.yaml
  - local_ " " " "
  - global_ " " " "

The local planner

Once the global planner has calculated the path to follow, this path is sent to the local planner.

It will execute each segment of the global plan

Global plan + Map → local plan → Velocity commands

It monitors the odometry and laser data

It can be computed on the fly

Types of local planner

- base_local_planner
  - Implementations of
    - Trajectory Rollout
    - Dynamic Window Approach (DWA)

- How it works
  - Discrete sample from the robot's control space
  - For each sampled velocity, perform forward simulations from current state to predict what would happen
  - Evaluate each trajectory
  - Discard illegal trajectories
  - Pick the highest-scoring one and send associated velocities to the mobile base
  - Rinse and repeat

- The difference between both implementations is in how the robot's space is sampled.

Perform similarly
- Trajectory Rollout → samples are set of achievable velocities over the entire forward simulation period given the acceleration ~~velocities~~ limits
- DWA → samples are from set of achievable velocities for just one simulation step given the acceleration limits
  - More efficient but outperformed by Trajectory Rollout with low acceleration limits
  - It does not forward simulate with constant accelerations
  - It's improved in dwa_local_planner

- dwa_local_planner
  - the same as but with much cleaner code and the most commonly used
- eband_local_planner → implements the elastic band method to calculate local plan
- teb_local_planner → " " time " " " " "

You can change the local planner as you wish with move_base node parameter file

rosparam set /move_base/base_local_planner options

" "

" set

- dwa local planner parameters

Local Costmap

- It is created directly from the robot's sensors readings. Given a width and a height (defined by user), it keeps the robot in the center of the costmap as it moves around the environment, dropping obstacle information
- It detect new objects that appear in the simulation whereas the global costmap it doesn't
- The global costmap is created from a static map file, so it won't change even if the environment does
- Settings: by setting static_map = false } we don't want the
  rolling_window = true } local costmap to be initialized from a static map
- Layers
  - Obstacle avoidance
  - Inflate obstacles
- The update cycles at a specified rate so as follow:
  - Sensor data comes in
  - marking and clearing operations are performed
  - The appropriate cost values are assigned to each cell
  - Obstacle inflation is performed on each cell with an obstacle. Cost values propagate outwards from each occupied cell out to a specified inflation radius

- costmap automatically subscribes to the sensor topics and updates itself according to the data it receives from them.
  Each sensor is used either to mark (insert obstacle information into the costmap) and clear (remove obstacle information from the costmap)

obstacle layer:
- Marking operation → index into an array to change the cost of a cell
- clearing " → ray tracing through a grid from the origin of the sensor outwards for each observation received

Common costmap parameters

- footprint: contour of the mobile base → used to inflate obstacles for safety, increment it
- robot_radius: in case the robot is circular, we specify this instead of the footprint
- layers parameters (ObstacleLayer)
  - Obstacle layer uses different plugins for the local costmap and global costmap (VoxelLayer)
  - Inflation radius

Recovery Behaviours

It could happen that while trying to perform a trajectory, the robot gets stucks for some reason. ROS Navigation Stack provides methods that can help the robot to get unstuck and continue navigating

- Clear costmap
- Rotate recovery

} recovery behaviours ← move_base parameter: recovery_behavior_enabled: true

Oscillation Suppression

It occurs when in any of the x, y or theta dimensions, positive and negative values are chosen consecutively
To prevent oscillations, when the robot moves in any direction is marked invalid for the next cycle until the robot has moved beyond a certain distance from the position the flag was set

Parameter file list for path planning

- move_base_params.yaml
- global_planner_params.yaml
- local_planner_params.yaml
- common_costmap_params.yaml
- global_costmap_params.yaml
- local_costmap_params.yaml

Footprint in RViz

- Polygon subscribed to /move_base/local_costmap/footprint

Current Goal in RViz

- Pose subscribed to /move_base/current_goal

Navigation Stack