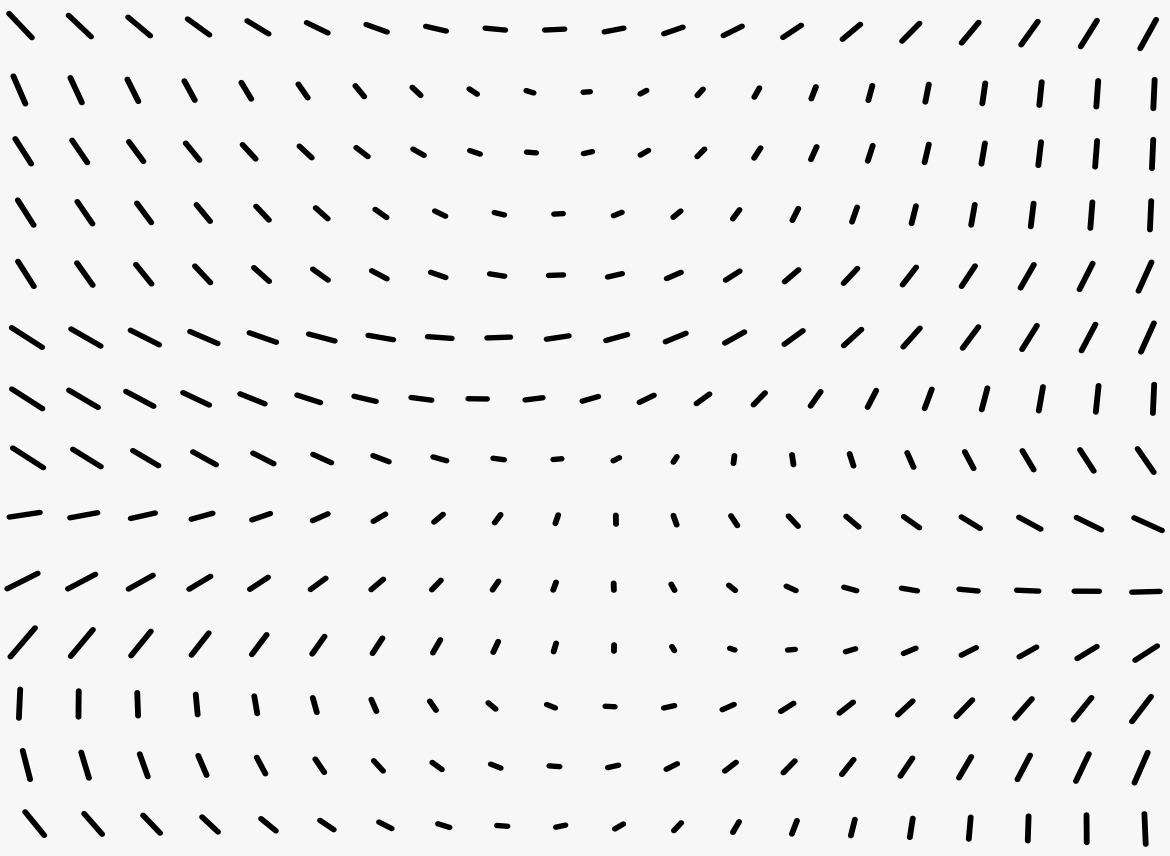
# ROS 2 Nav2



# ROS2 Nav2, from Edouard Renard, Udemy

NOTES by ALESSANDRO PUGLISI

## 1) INTRODUCTION

### COURSE INFORMATION

course on Nav2 stack for ROS2 → learn by doing!

- All done on Gazebo

KNOWLEDGE PREREQUISITE
- ROS2 basics
- Python
- Linux basics

PROGRAM
- Discover Nav2 stack by experimenting (perform SLAM on simulated robot, generate MAP and make robot Navigate on that Map) [PRACTICAL OVERVIEW]
- understand how Nav2 stack works
- How to create custom simulated world in Gazebo + step to adapt robot for Nav2 stack
- Write code to interact with Navigation from existing ROS2 Node

# WHAT IS Nav2 STACK, WHY WE NEED IT?

⇓

WHY NAVIGATION?

FIRST, let's consider:

Benefit of ROS2

- Create base layer of your robotics application reasily)
- Standard for robotics application
- Use for ANY Robot
- "avoid reimventing the wheel" → work on high level
- open source community
- Plug and Play packages

⇓

ROS speed up development time!

To implement navigation from scratch is too complex and time consuming...

⇒ we have **navigation stack**!

**Nav2** in ROS2 is successor of Nav1 in ROS1

↓

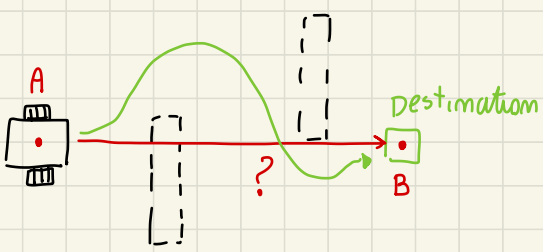
Nav2 **Stack** := A "stack" is a collection of ROS packages to achieve a specific goal

What we want to achieve in **navigation**?

**Main Goal:**

make Robot move from A to B in safe way

(find path to reach destination without obstacle / people collisions)




Achieved in a 2 STEP process

1) create environment map (SLAM)

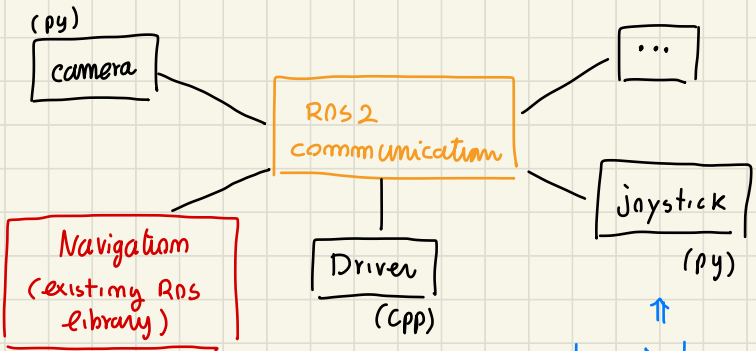
2) Make Robot navigate from A to B on that map

} all done using Nav2 functionality and tools

Once basic concept of Nav2 are clear:

⇓

Nav2 stack can be easily integrated in ROS2 applications




⇑

by independent py node, we can give commands to the stack and make robot navigate with simplified API

# 2) SETUP AND INSTALLATION

This course is done with

- ROS2 Humble (long term 2022 support ROS distribution) ← this ensure stable Nav2
- Ubuntu 22.04

↳ anyway this is valid for other ROS2 distribution

(With ROS2 Foxy, when the course was done, several bugs and issue occurs)

- FIRST, install ROS2 Humble in Ubuntu 22.04
- THEN, install packages needed for Nav2

ros-humble-navigation2
ros-humble-nav2-bringup
} will install collection of packages

+ turtlebot3: it is a mobile base robot with an already implemented complete simulation

ros-humble-turtlebot3(*) ← install all turtlebot3 packages

- FINALLY, install additional tools that will be used ...

colcon: build tool for ROS2, not installed by default when you install ROS2
↓
python3-colcon-common-extensions

git: needed to get code for GitHub repo

terminator: to open different terminal on same window
(useful to split horiz/vert the terminal)
- CTRL + SHIFT + O (horiz split)
- CTRL + SHIFT + E (vertical split)

good also for customization of terminal

VSCode IDE to write code in C++, Python

↓

in Extensions for ROS2 code
- ROS (valid also for ROS2)
  this will install also additional extensions
- CMake (good for CMake.txt coding)

Note for GAZEBO:

IF Gazebo give errors when launched, try running

. /usr/share/gazebo/setup.sh

↑

IF this solve the issue ⇒ add to .bashrc

# 3) GENERATE MAP WITH SLAM

To achieve the goal of navigation


" A to B Navigation avoiding collision "

↓ we start from MAP creation of the WORLD (the space where ROBOT can move)

↓

Generating a map is important in next step, so Nav stack can perform Navigation in any point of the world!

↑

Without a MAP could work Navigation, but without the map knowledge will take long time

that's why we separate in 2 main steps as seen before.

To Generate MAP, we rely on SLAM (Simultaneus Localization And Mapping)

↓

Allows Robot to localize in environment, relative to obtacles/walls etc around + map the environment

Once ROBOT and Simulation / Real world are set-up you can start SLAM to generate MAP

MAP is being generated as the ROBOT moves in the world

Once MAP is generated, we save it and use for Navigation!

## ROBOT MOVEMENT

FIRST of all, we need to have the ROBOT moving in a simulated world

FOR NOW based on turtlebot 3 ← all already implemented for this

Later on... Nav with custom robot

↓

for NOW, consider we need a ROBOT able to move in an environment with ROS2 + teleop node (move it!)

- how to make turtlebot 3 move ⇒ then we will see SCAM and map generation

↓

first specify model of ROBOT we are going to use, by exporting environment variable for example by modifying bashrc

example of turtlebot 3 model

```
export TURTLEBOT3_MODEL = waffle
```

to launch Gazebo simulation, we launch

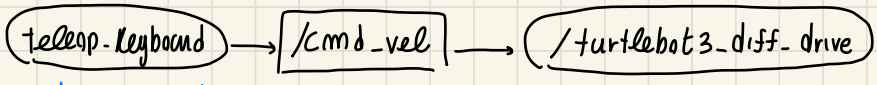
```
turtlebot3_gazebo turtlebot3_world.launch.py
```

⇑

differential drive Robot with a 2D LIDAR (~4 meter scan) ↑ to create an environment map (x,y plane)
spawn in a closed world

Then, to make ROBOT move we run

```
turtlebot3_teleop teleop_keyboard
```




send commands to /cmd_vel

robot listen to this topic with its controller, making robot move

This is all we need to MAP the environment

## GENERATE AND SAVE MAP

ROBOT + a way to move it, is what we need for SLAM and MAP creation

- launch the gazebo simulation world

- launch SLAM feature of turtlebot3

  turtlebot3_cartographer cartographer.launch.py use_sim_time := True

  additional argument because we work in Gazebo

  ↳ this will start Rviz2 (visualization tool)
  where the map is generated from the data generated by LASER SCAN

  ↓

  the goal is to generate entire MAP and save it

{ NOTICE: Gazebo replace real robot (simulation physics)
Rviz is a 3D visualization tool }

- launch teleop node to move the robot
  → during motion, in Rviz we see the map updating (while TF moves in Rviz)

MAP ~ { White pixels := free space
Black pixels := obstacles space
Grey pixels := still unknown }

Possible issues:

- Turning too fast create problems during mapping
- Being LaserScan in 2D plane, if ROBOT hit obstacle and oscillate vertically, map data can become noisy

in case of issues, restart SLAM and simulation

Once we finish the map exploration, we recover all map information.

↓

In Rviz we see the final map (minor missing pixels are NOT a problem for navigation)

Now we can SAVE the MAP before closing the SLAM + simulation

have a DIRECTORY for the /maps

then run

provide path and name

```
nav2_map_server map_saver_cli -f maps/<name>
```

no extension required

⇓

this will create a

<name>.pgm and <name>.yaml file

↓ ↓

WHAT IS INSIDE THE GENERATED MAP

Explore what are important things about the saved map.

<name>.pgm map image, White / black / gray pixels

↓ free / obstacles / unknown

this will be loaded when running Navigation to find easily path to destination

<name>.yaml contain different information

- image: relative path to image file (.pgm)
- resolution: in [m / pixel]
  IF for example 0.05 : each pixel is 5 cm (precision)
  the precision requirements depends on application and on the environment into account
- origin: coordinates of lowest left point in the map (bottom left corner)
  It will depends on where the robot mapping started

- negate: 0/1 IF 1 everything will be reversed free ⇆ obstacle

- occupied_thresh: clear separation free/obstacle... BUT in principle MAP is based on probability that pixel is occupied or NOT!
  ex) IF threshold is 0.65, means that IF pixel is occupied with more than 65% chance, it is considered occupied
- free_thresh: If probability of pixel to be occupied is less than this, the pixel is free
  ex) IF 0.25, cell free when less than 25% probability to be occupied

⇓

[ IF pixel more than 65% occupied ("into black direction") → black
  less than 25% occupied ("into dark value) → white ]

By opening the <names.pgm on the bash with `nano <names.pgm` we can see the pixels of the image

Nx ↑ # pixel horiz  Ny ↑ # pixel vertical ⇒ $N_x \cdot \text{resolution} = L_x$, $N_y \cdot \text{resolution} = L_y$ } give you the map dimension in meters

- **ASSIGNMENT 1**

Practice on generating map with SLAM.
Generate a map for a new world "turtlebot3_house"

Launching `turtlebot3_gazebo turtlebot3_house.launch.py`

→ in this world, we have a simple house environment (open in the outside, NOT closed in walls)

Follow the same procedure explained before to generate map and save it (just map around the rooms, NOT open world...)

[ NOTICE When mapping, be sure NOT to have "holes" on the walls and other noisy informations that will then effect navigation. ]

- RECAP: (pdf provided as course resource)

# SLAM Steps (ROS2 Nav2 Course - Section 3)

Those commands are the ones you will run in this section on SLAM. Use this PDF to easily access them while doing the exercises.

Some of the commands are specific to the Turtlebot3 robot, which we use as an example. Later on in the course (Section 7), you will also get the general commands to run for any robots.

## Steps

When you see some text in red, replace it with the correct value.

1. Start your robot stack (simulation environment)

```
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

2. Start SLAM (mapping from sensor data)

```
$ ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
```

3. Make the robot move to generate the map (teleoperate around to retrieve data)

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

4. Save the map

```
$ ros2 run nav2_map_server map_saver_cli -f ~/my_map
```

# 4) MAKE ROBOT NAVIGATE WITH Nav 2

- we have briefly described the 1st step of Robot Navigation
  creating a map ☑ achieved by SLAM algorithms.
  ↓
  we generate a map { <map>.pgm
  <map>.yaml

- now it's time for the 2nd step: Navigate from A to B using the map created ...
  again we will start the robot on the mapped environment simulation / real robot
  ⇓
  On this, we will start Navigation Stack giving as input the map previously generated → in RViz, we will have a new interface to use in RViz to give navigation commands

- Go to any point in the map (feasible) avoiding existing obstacles + new obstacles added dynamically during navigation

---

FIX BEFORE STARTING, to make Nav2 WORK IN HUMBLE

when loading map, the map may be NOT loaded properly in time, preventing successive steps...
↓ Fixing the bugs... in two main steps

(Data Distribution System, ROS 2 communication is based on this)

1) change DDS from FAST to CYCLONE

this will allow ROS2 Nav2 running properly

install: ros-humble-rmw-cyclonedds-cpp

and once installed, tell ROS2 to use it: (exporting env variable)
editing bashrc, we add to it:

export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp

im Humble the map and navigation is NOT correctly working with default DDS...
this 1ST FIX should solve the map issue

2) modify parameter file where turtlebot3 navigation package is installed...

NOTICE: this is NOT good practice since if im the future this package get updated, you will need to do it again
(do it only temporary, probably this issue will be fixed)

go to /opt/ros/humble where ROS is installed,
then the parameters files will be in .../share folder,
move to .../turtlebot3_navigation2
on this folder, go to .../param and edit the model.yaml with the model of TURTLEBOT3 im use
(im our case waffle.yaml) ⇐ edit it with sudo (super user needed)

Im this file, edit it to:

robot_model_type: "nav2_amcl::DifferentialMotionModel"

(reboot the computer after this modification)

# MAKE ROBOT NAVIGATE in the generated map

For now we will work with turtlebot3, later on we will run navigation on a custom robot

- FIRST, we start the robot simulation (gazebo) as usual
- start navigation, by launching

containing all nodes, etc. to run navigation

`turtlebot3_navigation2` `navigation2.launch.py`

`use_sim_time := True` `map := .../<map>.yaml`

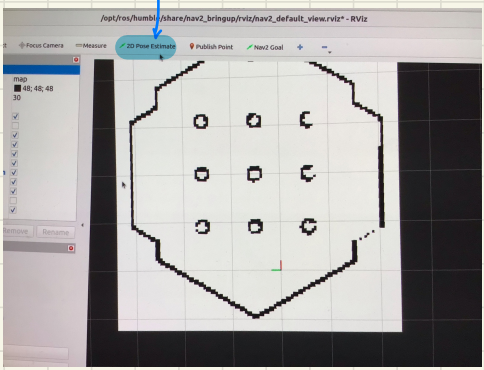
working in sim environment

provide the saved map to the navigation stack, by relative path to the map

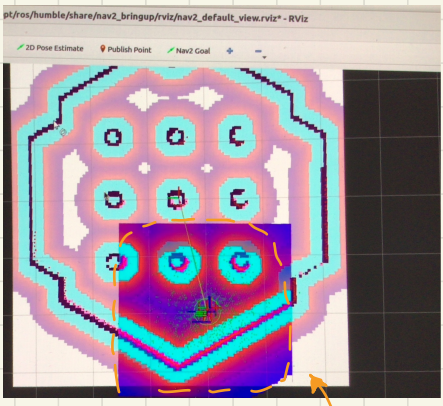
⇓

this will start Rviz with the map, with origin as specified in yaml file
(IF map NOT loading, try to restart)

- On Rviz, to start navigation, we need to give an **initial 2D Pose estimate**, estimation of where robot can be




click on it, and then place the robot pose according to simulation initial pose



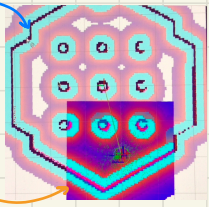

we have communicated to Rviz where is the Robot now...
So, navigation localize the ROBOT there, and using the data from sensors (gazebo), the environment is matched
When robot move, localize itself in the map

IF I give a WRONG 2D pose estimate... laser scan data will NOT correspond to map in Rviz

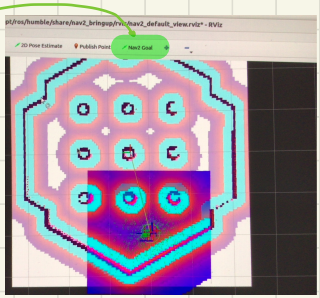
Be as close as possible to real pose!

the navigation is based on { global planner, local planner (controller) }

(details later on...)

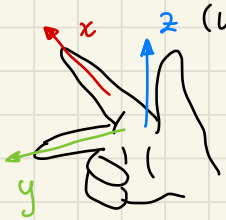



- once given 2D Pose estimate, the match map / sensors is correct
  ↓
  we can send a Nav2 Goal



by selecting it and define desired orientation.
this will start motion in simulation

NOTICE: coordinate system uses right-hand rule (used by TF to represent relative pose)




In Rviz we have special important frame for navigation, we will discuss details later on

IF we give unfeasible Goal pose...
ROBOT try to move, start Recovery behaviour... Feedback: aborted
↓
NO path found, Goal gets aborted (feedback from action)
If goal is feasible, we get Feedback: reached

When relaunching Navigation, from initial 2D Pose estimate, the robot try to match as close as possible to sensor data to localize. and map will converge to correct pose

In summary, to run Navigation { 1) 2D Pose estimate
2) Nav2 Goal

↓

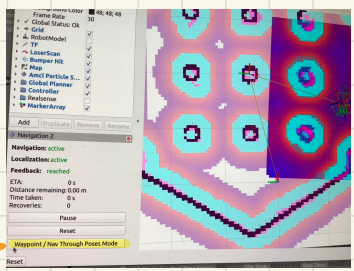
later on we will manage this two steps NOT in Rviz by hands, BUT through code, interacting with Nav2 stack

## WAYPOINT FOLLOWER, Multiple Nav2 Goals

Usefull functionality in Nav2, to define different points.
Instead of defining each point as Nav2 Goal, an alternative tools exist

↓

It is possible to send more goals all at once




after selecting this Nav2 tool, go to Nav2 Goal as usual, and you can define "wp_i": waypoint i

then >> start Waypoint Following on bottom left
Will start the navigation going on all the defined waypoints

If we select >> start Nav Through Poses instead, create path through All waypoints without stopping
usefull for fast navigation, but less stable than waypoint Following

## DYNAMIC OBSTACLE AVOIDANCE

Important feature of Nav2 stack ⇝ avoid obstacles dynamically

- the Map is already known, so the static obstacles are already known for the Robot
  What if a new object NOT previously in the map?
  ⇓
  Navstack should be able to avoid it and replan path.

- To test it in Gazebo simulation, we can >>insert New obstacles to the world
  ↓
  to see dynamic obstacle avoidance in action,
  we can PAUSE the simulation in Gazebo (during navigation)
  and add an obstacle object in the path

  Robot can still try to go through the obstacle...
  depends on sensor visibility!
  IF we add an obstacle that can't be properly estimated,
  then there will be issues for dynamic adjustment
  ↳anyway new path will be computed ⇒ Bad behaviours can occur!

  IF the obstacle is correctly perceived, even if NOT part of static map, it will update the path at runtime and then map will be cleared if dynamic obstacle get removed or move from previous position

  (this dynamic obstacle avoidance is achieved by the controller / local planner)

- **ASSIGNMENT 2** Using the "house world", experiment with navigation on this environment. Thanks to map of ASSIGNMENT 1
  (remember to load correct map.yaml as argument)
  Objective: Patrol over the whole house by Nav2 Goal or waypoint Follower

NOTE :
- when moving on cluttered environment, navigation may get stucked and even if a feasible path exists, the navigation is aborted.
  Especially when manouvers on narrow environments are required (like "U" turn inside rooms)

- Later on we will explore how to modify navigation parameters.
  This can tune navigation for your specific robot, and optimize it!

# Navigation Steps (ROS2 Nav2 Course - Section 4)

Those commands are the ones you will run in this section on Navigation. Use this PDF to easily access them while doing the exercises.

Some of the commands are specific to the Turtlebot3 robot, which we use as an example. Later on in the course (Section 7), you will also get the general commands to run for any robots.

## Steps

When you see some text in red, replace it with the correct value.

### 1. Start your robot stack

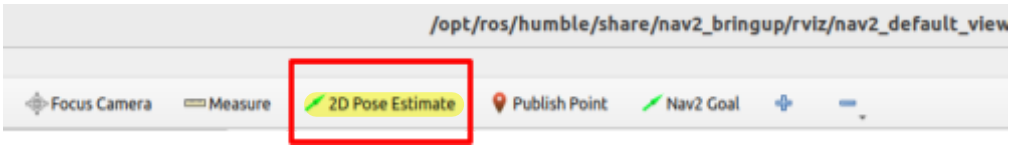
```
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

### 2. Start Navigation 2

```
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True
map:=path/to/world_map.yaml
```

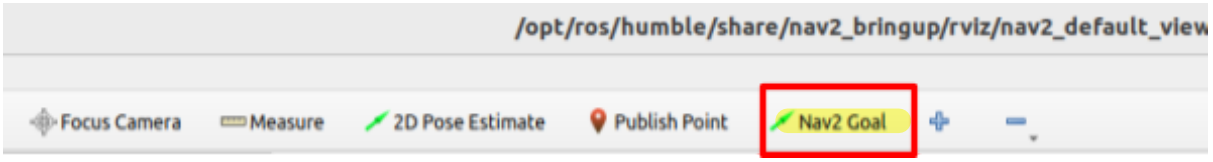
### 3. Set 2D Pose Estimate

Click on RViz




### 4. Send a Navigation2 Goal

Click on RViz



# 5) UNDERSTAND THE Nav2 STACK

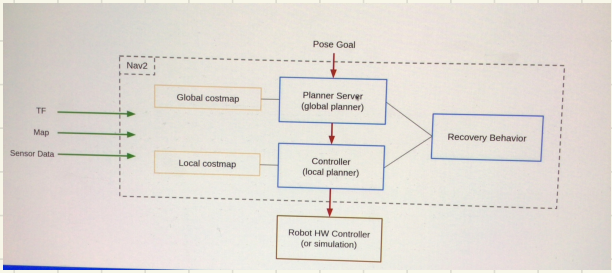
Once seen the two steps of Navigation stack { SLAM for map generation / How to navigate

↓

Let's see

- what is going on in Nav2 stack
- what are main components
- what is the architecture

← understand it by running it and focusing on one component per time




↙ at the end, a better grasp of the architecture will be given

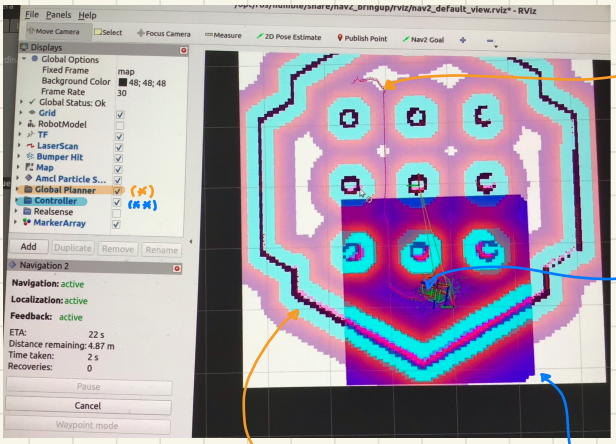
## GLOBAL / LOCAL PLANNER AND COSTMAPS

by examining the Nav2 with turtlebot as done during section (4)

↓

What happens when we give a Nav2 goal?

How Global and Local Planner works and interacts with each other?




(pink one) path to take to reach destination (global plan)

(blue one) local plan

the GLOBAL PLANNER (✗)

LOCAL PLANNER (controller) (✗✗)

When we send Nav2 Goal, this is sent to Global Planner: it will compute the path based on the entire map, to reach destination as fast as possible with best possible path!

(the Geobal planner)

To compute this path, it will use a GLOBAL COSTMAP (the colored one above the black/white pixel map)

↓

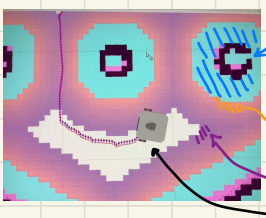
map in which each pixel have a cost,

- obstacles have highest cost

with some margin around obstacles (to avoid going close to it)

↓

around obstacles we have safety margin to avoid collisions




in the costmap we have different color shade to represent different cost

⇒ the closer we are to an obstacle and the higher is the cost

To compute the path, GLOBAL PLANNER will add all pixels it has to go through and find the minimal cost path.
Try to go as much as possible on free space and the one far from obstacle (this ensure manouvers far from obstacle)

This GLOBAL planner get updated at low frequency (~1 ÷ 5 Hz)
BUT this Geobal planner is not what really control Robot motion...
↳ the path is sent to Local Planner, which CONTROL motion.

↙

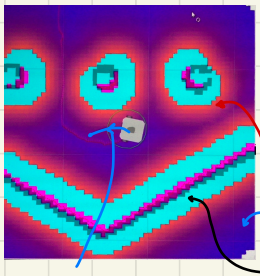
Local Planner has its own Local costmap.
After receiving geobal path, the controller try to follow it, and it is characterized by high frequency update (~ 20 ÷ 100 Hz)

- Analogy: driving a car with GPS: the GPS compute plan and update with low frequency (Geobal plan).
  Then, you will control the car, deading velocity, steering etc with high frequency, following GPS as much as possible.
  You also take into account possible correction of that global plan

In ROS2 Nav2 stack: { Global planner ~ 1Hz (GPS)
Local planner ~ 20Hz (car driver)

also, Global planner has smaller update frequency because it takes more computation power for this path
(IF running at high freq, it takes too much resources)

the LOCAL COST MAP



↓ works with same principle of the GLOBAL one, but only on the ROBOT surroundings

red higher cost

blue lower cost

light blue are obstacle safe zone to avoid

⇒ Local planner will follow the path, by avoiding Higher cost pixels

Local plan currently following ⇒ Global path updated a bit, and Local planner update with high frequency and take best short term decision to control robot motion

↓

the local planner will control by sending a velocity command to the robot

# PARAMETERS

Focus on a few of Nav2 parameters (some of them allow to tune costmap)

↓

We will see just some, because there are a LOT to define Nav behavior

Focus on the most important / usefull

(Once understood this ones... it will be easy to adapt the knowledge to others)

To find the parameters:

on a new terminal `rqt`

↓

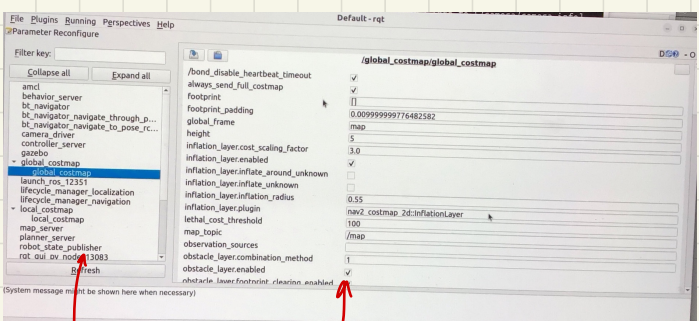
GUI with lots of plugings: > Plugins > Configuration > Dynamic Reconfigure

It will open a list on the left, we can see • global_costmap

↓

global_costmap

for example:




↙ all live parameters for global costmap

we have many parameters section

and for each section a lot of parameters...

Let's see the most important

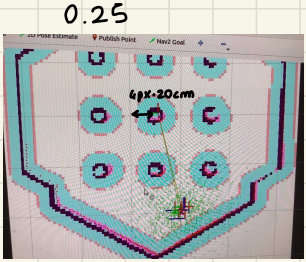
• publish_frequency : 1.0 [Hz] (frequency of update of global costmap)

• inflation_layer.inflation_radius: [m]

0.55

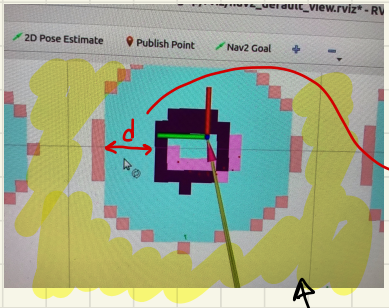
→

0.25




the map has more white space... this "inflation radius" represent the radius from obstacle where the pixel still have cost

↳




inflation radius: radius from the obstacles where we still have cost in the pixel

5 cm/pixel discretization of map

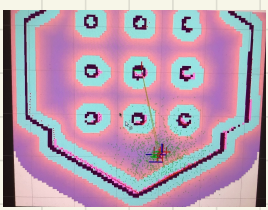
d ≃ 4 pixels : 4.5 = 20 cm of space considered as obstacle around (light blue)

+ 0.25 cm

inflation radius → pixel cost up to 5 pixels radius from obstacles

The same around the walls...
we have pixel cost up to 5 pixel far from it.
While all the other space is considered with 0 cost free space

0.8

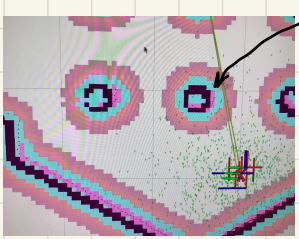


everything has higher cost,
up to 0.8 ~ 80 cm = 16 pixels radius of cost

- robot_radius : 0.22 [m]

used to compute where the Robot can go

IF I update to 0.1



obstacles light blue area get updated to smaller areas

⇒ this means that robot can traverse closer to the obstacle

It is good norm to use the correct "radius" of the Robot model used for navigation (22 cm is waffle turtlebot 3 radius)

- resolution 0.05 [m]

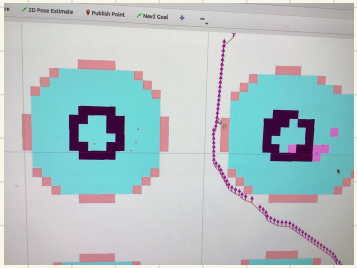
specify map pixel resolution (~5 cm)

- In `local_costmap` we have same parameters as global_costmap

for example • `inflation_radius`: 1.0 [m]

IF I update to 0.5 or smaller it get updated as for geobal one

IF this value is smaller in both, the robot will navigate closer to the obstacles,




this will be a faster root and going through free space with low cost

the choice of inflation_radius require a good trade off

- the lower: easy path computation but higher probability to hit obstacles in real life narrow environments
- the higher: less hitting probability but more complex path finding

- In `controller_server` ("local planner")

you will have usefull parameters like

- `max_vel` (change acceleration) [m/s]
- `controller_frequency`: frequency at which local planner (controller) is sending command velocity to the robot [Hz]

usually ~ 10 ÷ 100 Hz

- `goal_tolerance` when destination reached, how far it can be from destination (some tolerance)

...

⇒ from this description, experiment with the others!

# RECOVERY BEHAVIOR

Briefly recap: when sending a Nav2 Goal

↓

IF everything is IDEAL ← that's all we need!

- Global planner compute (path) based on Global costmap
  ↓
  path to local planner (controller), responsible to make robot move, and follow path in Local environment

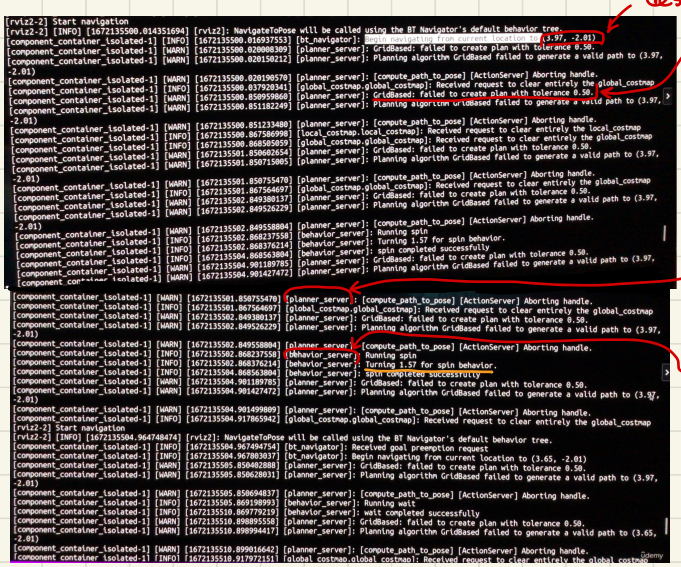
↳ BUT, in a real environment, the ROBOT sometimes doesn't manage to reach the destination, because:
- Non valid Goal
- dynamic obstacle unavoidable
- path too narrow
- algorithm fails to follow path

In those cases, Navigation stack will start the RECOVERY BEHAVIOR

↓

this is a behavior pre-defined for the ROBOT, that will try to fix the current issue, so that Robot can continue to move and reach destination.

When unreachable destination is sent:




...

until we get a "Goal failed"

What happens is that :

GLOBAL PLANNER : [planner_server] try to compute path and send to local planner

IF there is an ISSUE and global planner can't generate the path OR local planner can't follow that path

↓

recovery server will be called, and start a recovery behaviour

THIS RECOVERY BEHAVIOR can be: turn around, go backward a bit etc..

for ex: when driving car, you try to turn a bit and go backward IF there is NO visibility, to try to clear the path

SOMETIMES it succed to replan and recover the path

The logs in terminal show us how path following is going...

"Passing new path to controller" try to do something else until "Running backup"

↓

try to find new path... When issue with global planner / local planner behavior_server is called → start a series of recovery behaviors to try to clear the map and find valid path

IF it doesn't succeed, at a certain point the Goal will be Aborted

# TFs AND IMPORTANT FRAMES

Another important "component" to understand before the details of Nav2 stack Architecture are (TF) (NOT Navigation specific. Used for any Robot developed in ROS)

⇓

Just quick introduction of TF to understand what is it and how are used in navigation

"TransForm" visible in Rviz, in the TF section...

⇓

Problem to solve: In a Robotic application we have many frames representing:
- different robot parts
- origin of the map
- environment frames

Challenge:
To keep track of each frame relative to other frames

[ex: • where is the Robot relative to map origin?
• where is left-wheel pose relative to robot base?]

The default math solution is based on Rotation/translation computation in 3D. This has to be done for each frame in the Robot and environment
LOT OF COMPUTATION!

⇓

ROS 2 approach: use tf2 package!
which keeps track of each frame over time, as structured tree containing all frames.
To get this TF tree we can subscribe to /tf or see it in Rviz

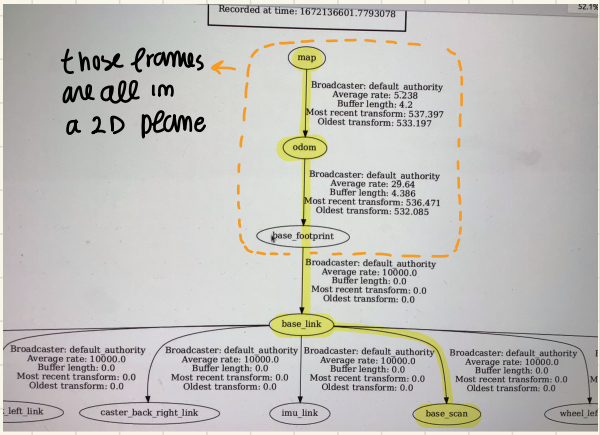
In Rviz we see the frames gets updated at runtime moving one with respect to the other.

In /tf topics we have a set of transforms published mapping parent to child relationship. As Rotation + Translation

Also, to visualize the TF tree, run

`tf2_tools view_frames`

that will export a pdf with a representation of it.




**base_footprint**: projection of the robot on the ground (being the base a bit elevated, the base is projected to z axis)

**base_link**: main robot chassis where all wheels and sensors are attached

## Required TFs for Nav2

⇓ when you have a Robot running with Nav stack you need:

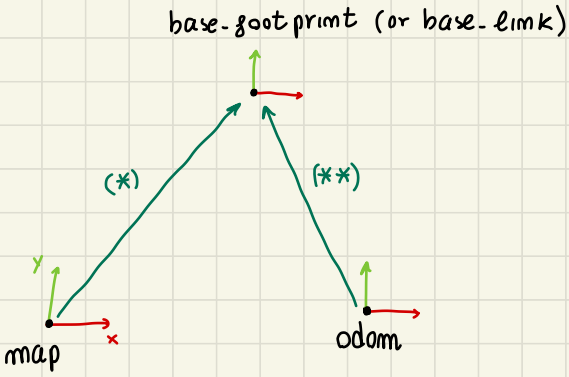
- map → odom
- odom → base_link (NOT direct, but through base_footprint)
- base_link → base_scan (where the lidar information are)

↑ NOT an issue, the odom to base_link can be still computed

usually this is fixed, being the LIDAR fixed on the main structure of the Robot

Let's focus on **map** and **odom** relation with base_link

↳




(*) • map → base_footprint : used to compute exact location of the Robot with SLAM using LaserScan from Lidar (or GPS)

ISSUE : the location can be precise over long time, but in short term is unstable (jump from one point to another, uncertain)
→ correct in long term but noisy in short term

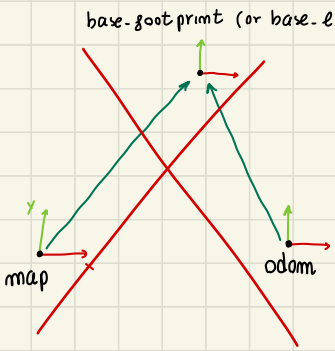
(**) • odom → base_footprint :
"odometry" = estimation of robot position using robot internal measurements (for ex. recorded wheels speed)

The location with odometry will be smooth in short term, but will drift over time (error cumulate)
ex. when estimating the distance you walk based on step size... you accumulate error over time!

Nav2 stack: it take the best of both, combining map and odom to have precise location of the Robot over time (map) with smooth/linear location NOT jumping around (odom)

⇓

due to how TF are designed in ROS: any frame can have at most one parent! (and many children)

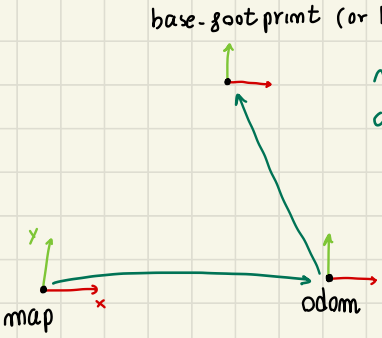



← it has two parent ?

NOT possible in this way

⇓

⇒ It has been decided to structure TF tree as:




map as odom parent and odom as parent of base_footprint

even if this map → odom relationship seems to NOT have any sense

↪ this is a workaround to make TF works with one parent per link

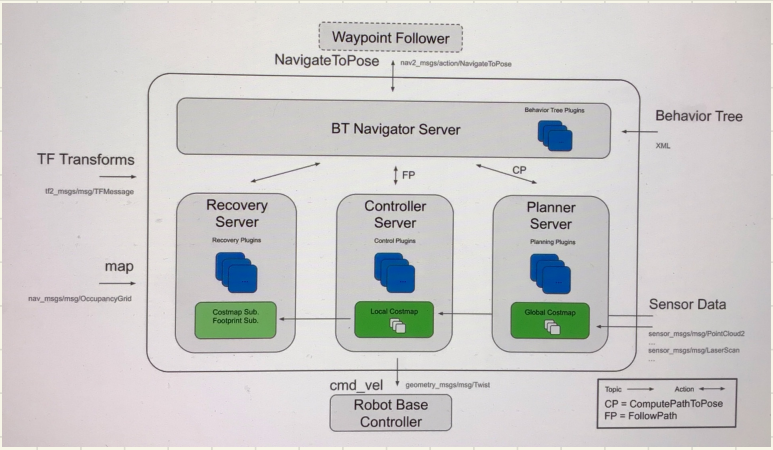
the important aspect is:
- map for long term precise location
- odom for smooth location in short term

↳ it may happen that odom "jump around" relative to map, but this will be compensated for base_footprint (odom to base_footprint adapt automatically to compensate map to odom noise)

robot will not "jump around" ←

# Nav 2 ARCHITECTURE

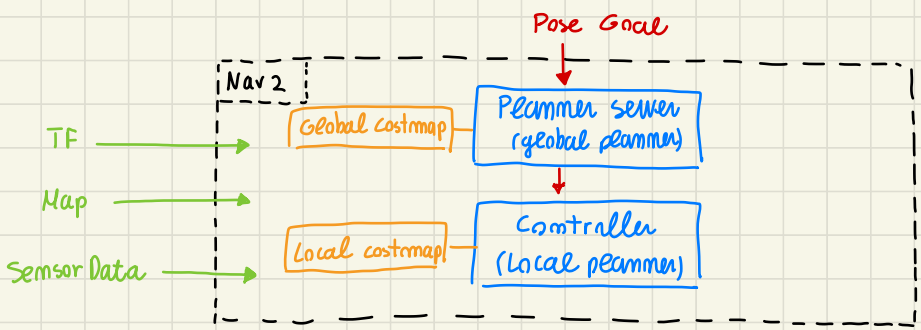
We can now put all together the main components of the stack in the global Architecture.




↳ Offical Nav2 stack architecture in documentation

⇓

Let's simplify it a bit, building our own representation step by step




to make Nav2 stack works, we need some inputs

TF, Map (generated by Mapping before Navigation, the Map is given to the stack)

Sensor Data (such as Laser scan or camera, 3D scanner etc)

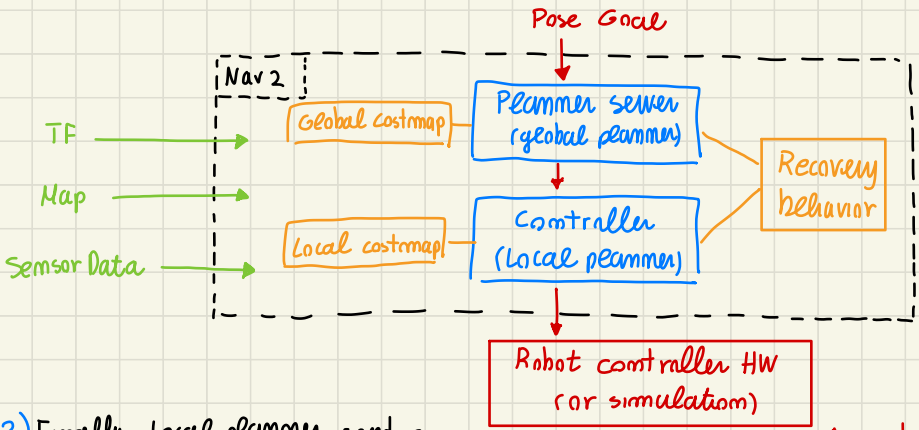
When you send a Nav Goal in terms of Pose (position + orientation)

1) global planner (called "Planner server") is responsible to find a valid path, using both Map and Global costmap

↓

once a path is found

2) Local planner (controller) try to make sure robot follow the path and reach destination. Using Local costmap AND the data from all sensors



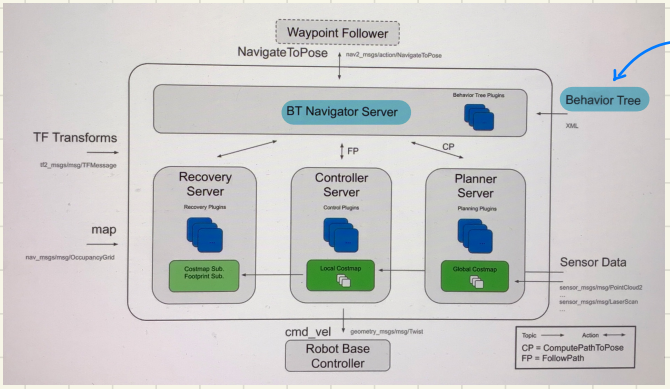

3) Finally, Local planner send a command to Robot Controller itself

↑ indipendent from Nav 2 stack!

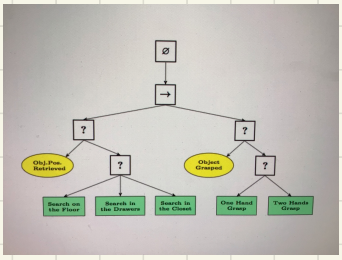
" custom controller of the Robot that take velocity command and translate it into a command sent to motor (so that Robot move)

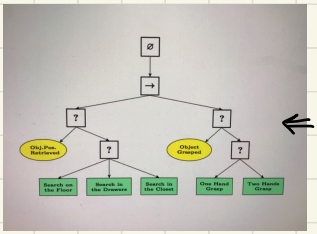
4) To complete the architecture, When path is NOT followed properly... a Recovery behavior is integrated

From this simplified Nav 2 stack Architecture, IF we look back to the offical one: It is almost the same...




this is the big difference from our simple version "Behavior Tree" which is NOT Nav 2 specific, an example of Behavior Tree:






BT example:

You want to GRASP an object

1) search for the object around until you found
2) try to grasp it (one/two hands)

In Navigation: When the stack receive Pose Goal

1) Use Planner Server to find valid path
2) Use controller to make robot follow path
3) IF issue occurs: recovery server is called, to try Recovery Behavior

# 6) BUILD YOUR OWN WORLD FOR NAVIGATION IN GAZEBO

We have seen how to work with Nav2 and what it is, by using existing simulation worlds

↓

we want the robot to move into a custom world.

Building a simulation of your real world is very useful to test your software application

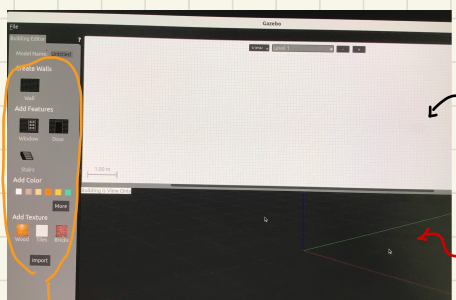
## BUILD AND SAVE A WORLD IN THE GAZEBO BUILDING EDITOR

First, just run gazebo on its own: `gazebo`

↓

there, go to > Edit > Building Editor

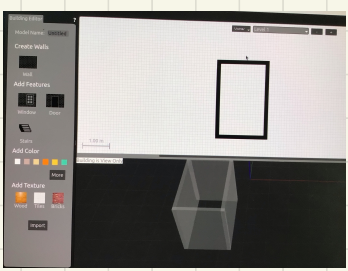
this space allow you to build a world with many elements




top-view (2D plan)

perspective view

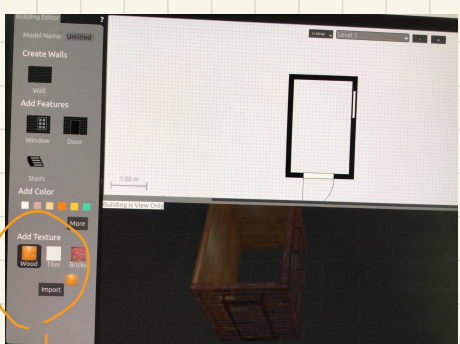
click here to add and customize elements



for example, adding 4 walls in a rectangular shape

then we can add doors, windows, textures and so on...

very easy world creation (even if now doors seems closed, it will be an opening wall later on)




add textures by clicking on the desired texture, then select element to apply it in the world 3D

+ adding stairs and create multiple levels is possible!

Then, To save this wored: File > save as > <name>

⇓

it will be saved in a new folder by default

it will generate two files: <name>.sdf , <name>.config

<name>.sdf: XML file, sdf is the wored description

<name>.config: just provide some model information

⇓

NOW, to import the wored created in gazebo, stunt it and then:

> Insert > path to custom wored > model_name

IF you don't find this path, click to "Add Path" optiom in Insert

↳ and you can drag and drop this as many time as you want

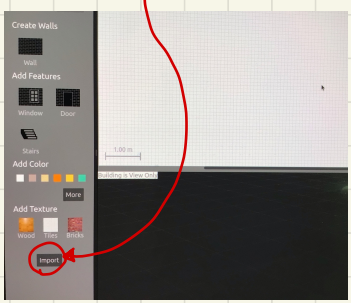
## IMPORT A FLOOR PLAN

Building the wored from scratch is useful, but it is possible to start from a given floor plan.

this has all the rooms dimensiom

↓

we can import an image in the Gazebo wored editor, and start from that to build your wored




click on import and select the file, after selecting it, we need to set scale: to know how big are the image dimensiom

to set scale: use two points with known distance.
Just click on two points with a distance in meters that you know, and set the distance ↳ this will fix the entire scale

IF you already have a scale factor metric, you can use that to set the plane scale.
↓
finally click "OK" ↝ you will see that floor plan in 2D view

Now we can start to build on top of the image, using walls, doors and windows
(by holding SHIFT during wall construction I have more precision)

By building on top of the floor plan, with given scale, we will obtain a 3D world of the correct dimension

## ADD OBJECTS TO THE WORLD

we build a custom world with rooms / doors / windows...
What about objects?
↳ start gazebo and insert your custom model plan
⇓
Then, on the Insert tab, we have different possible path from which insert models.
FROM https://models.gazebo... we can add common objects to the world
(sometimes it may take a while or crash when adding objects)

Add the object you want in the desired positions, Then, to save this world you can go to

>File> save World As> ... save it with <world_names.world

while in the building editor we save it with ".sdf", ".config" extensions, When saving as Gazebo world, we save with ". world" extension

this is also an xml world description (sdf)

⇓

I can start gazebo with this custom world by launching it as

gazebo <world_name>.world

## MAKE TURTLEBOT3 NAVIGATE IN THAT WORLD

### STEP 1, CREATE TURTLEBOT3_GAZEBO OVERLAY

Once we create our custom world, we can use this simulated world when testing our Robot

GOAL: Adapt launch file so that we can spawn turtlebot 3 in this custom world and test Navigation / mapping on this world.

to run turtlebot3 in a simulated environment, we launched

turtlebot3_gazebo turtlebot3_world.launch.py

the launch configuration is in this package

go to /opt/ros/humble/share/turtlebot3_gazebo/worlds

↑ here I have all installed packages

⇓

we need to adapt this package!

BUT we are not going to modify it where it is installed, Instead we will clone it in our own workspace

for example

- ros2_ws then on it follow usual steps to create a workspace

We will clone in ros2_ws/src/ the turtlebot3_gazebo package and override it!

so, we will have the underlay in /opt/ros/humble/share
while the overlay in /ros2_ws/src
↑
we create an overlay of turtlebot3_gazebo package, to modify it and use it, as well as call in the underlay

find turtlebot3_gazebo in github → we will have to clone turtlebot3_simulations containing turtlebot3_gazebo
↓
use git clone as usual in /src

git clone <URL>

Then, we need to checkout the correct branch, with your ROS2 distribution

git checkout humble-devel
your ros version

Now you can colcon build the workspace from /ros2_ws

(remember to set-up the workspace, so let's add the
source ~/ros2_ws/install/setup.bash)
↑ this has to be AFTER the source /opt/ros/humble/setup.bash to work correctly!

⇓
by launching turtlebot3_gazebo, now the launcher will be loaded NOT from global ros installation, but from our own workspace
↳ we can modify and customize it in our overlay!

# STEP 2, ORGANIZE CUSTOM LAUNCH FILE FOR OUR Gazebo WORLD

Let's modify the turtlebot3_gazebo package to use our custom world and start turtlebot 3 robot there.

Looking in the /launch folder, we have to create our own launch file there.
And a parameter loaded from /worlds define which world to load
we need 2 steps:

1) Add custom world
2) New launch file

## 1) Move custom world to .../worlds

move <world>.world inside turtlebot3_gazebo/worlds
(in our overlay)

↓

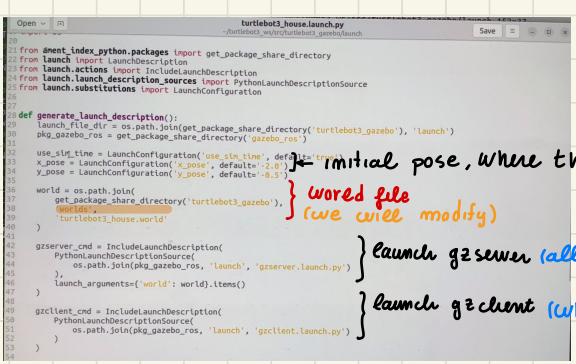
to make it appear as the other .world file, add the
<?xml version="1.0" ?> at the beginning of the file

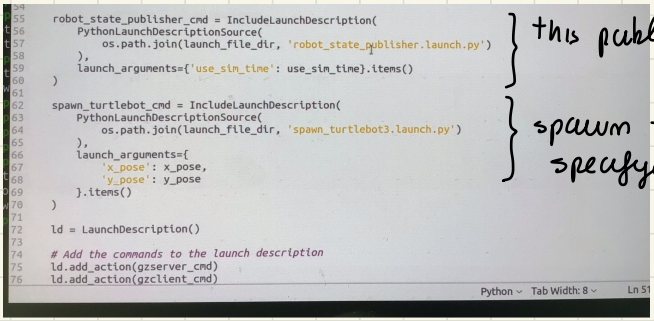
so, also
it will be recognized as xml by text editor

## 2) Define a custom launch file

We will copy one existing turtlebot3_gazebo launcher and properly modify its parameters

For example: turtlebot3_house.launch.py





```
robot_state_publisher_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        os.path.join(launch_file_dir, 'robot_state_publisher.launch.py')
    ),
    launch_arguments={'use_sim_time': use_sim_time}.items()
)

spawn_turtlebot_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        os.path.join(launch_file_dir, 'spawn_turtlebot3.launch.py')
    ),
    launch_arguments={
        'x_pose': x_pose,
        'y_pose': y_pose
    }.items()
)

ld = LaunchDescription()

# Add the commands to the launch description
ld.add_action(gzserver_cmd)
ld.add_action(gzclient_cmd)
```

Python  Tab Width: 8  Ln 51

this publish TF for the robot

spawn turtlebot3 with specified x,y pose

⇓

we can copy this launch file in a new file

turtlebot3_my_world.launch.py

↳ then, edit it:

just edit the worlds by changing world name to my_world.world

NOW we can build the workspace!

(remember to source or open new terminal)

And you can test the custom world launching

turtlebot3_gazebo turtlebot3_my_world.launch.py

will start

ROBOT in the world, in the position x,y specified in our launch file in x_pose, y_pose

⇓

we can customize robot spawn position with desired initial position accordingly to our world (ex -1.0, -5.0 in the floor map)

⇓

Once turtlebot3 spawn in this custom world, we can move it with the teleop_keyboard node as before!

## STEP 3, MAP AND NAVIGATE IN THE CUSTOM WORLD

Now robot is fully integrated in custom world.
We are ready to reproduce the mapping and navigation procedures explained on previous lectures

1) Launch simulation
2) run teleoperation
3) Launch SLAM with turtlebot3_cartographer
   ... at the end of map reconstruction, save the map with map_saver_cli

IF you have missing world parts it is possible to fix that by hand, in a post-process phase!
to make it more reliable for navigation
(we will see later how to do this)

NOTE: according to the 2D plane of Lidar, we may have some issue detecting some parts of the environment if at lower height, or due to visibility limit.
(you can clean the map later on)

CLOSE ALL TERMINALS, relaunch simulation
4) Launch Navigation tools with navigation2.launch.py passing the new map as argument
   - select 2D Pose estimate
   - send Nav2 Goal / waypoint

# TIPS: HOW TO FIX AND IMPROVE MAPS WITH GIMP

once we reconstruct the map, it is possible to fix the missing map informations (missing pixels).

A solution can be to restart SLAM and remap, but it is time consuming...

- uncomplete wall
- free space still GREY unrecognized
- twisted map during SLAM

being a map
just an image (set of pixels)
→ to make it more accurate, we can just edit the image

This can be done with GIMP, free editing program for images
easy to use! you can use Photoshop or others if you want

sudo snap install gimp

then you can launch it with gimp
and modify the map.img by simply drag and drop the image in GIMP UI

- Few tips on GIMP usage:

  - CTRL + Mouse to zoom in/out

We just need black/white color to modify the map.pgm
by using Pencil tool, select the size properly

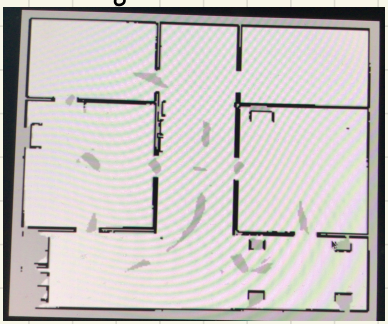
Black pencil: fix the missing wall pixels, objects perimeter etc.

complete wall
pixels closing lines, just for navigation purpose!
We don't need accurate lines

Also add elements in the ground NOT detected by LIDAR if you know about some undected obstacles

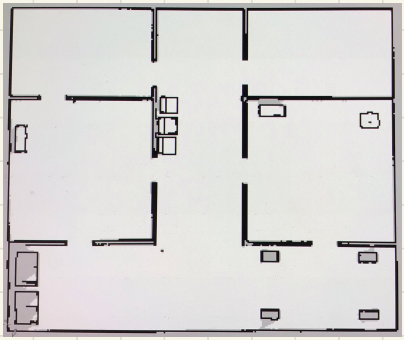
Then, using White pencil: clear the part of the map we know is free space, for wrong obstacles or unknown grey space.

before



⇒

AFTER GIMP post-process



>File > Export As ...

There is no need to rename / modify the <maps.yaml, be sure that the image parameter in configuration yaml file link to the new version of the map

↓

then, when launching navigation tools, the map loaded is the modified one

Another use of map editing with GIMP: when we have big environment, mapping ALL at once may be a problem...

→ we can make some maps of some areas and then merge more images into one

# • ASSIGNMENT 3 · MAKE A ROBOT NAVIGATE INTO A GENERATED MAZE

- Create your own custom world
- Adapt turtlebot3 for custom world
- Make Robot navigate

} follow same steps as the ones of this lecture

We are going to generate map by a maze generator online

→ mazegenerator.net

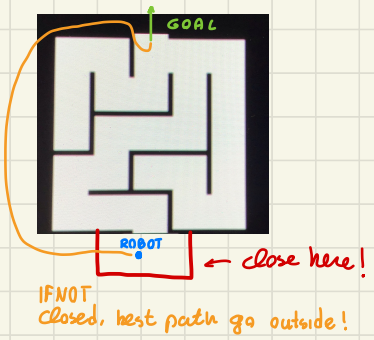
> Generate new and save as >PNG (NOT as PDF) Them, download

> 5×5 dimension is suggested, too big becomes hard to map

## Assignment

1) Create world in Gazebo from maze as:
   - ".world" extension,
   - close one of the entrance at least → because if we want the Robot to traverse the maze, if we leave both dors open, it will go to the goal from the outside of the maze
   - Use 1.2m scale for door frame

you can add objects if you want!




2) Adapt turtlebot3_gazebo package to spawn turtlebot on this world
   - spawn it at one of the entrances

3) Generate map of maze with SLAM + Navigate on it

## USEFULL COMMENTS ON THE SOLUTION

PART 1)
- Avoid excessive narrow spaces when recreating the map with Gazebo Building Editor. otherwise when navigating we will face issues during navigation.
- After creating the maze, generate a .world file from gazebo itself

PART 2) Just follow the same step to create a new launch file

PART 3)
- When you send a Nav2 Goal in unknown (grey) space, the Nav2 stack generate a straight trajectory to the goal, Unknown will be considered as free space.

- When navigating from start to end of the maze, the global path is easily found, but you can encounter issues when "U shape" manouvers has to be done.
  you can try to redefine a waypoint following over the planned path to complete navigation.

# 7) INTRO TO ADAPTING A CUSTOM ROBOT FOR Nav2 (STEPS OVERVIEW)

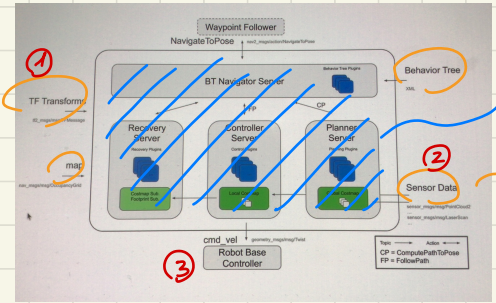
We have seen how Nav2 stack work with SLAM and Navigation + how to create a custom world for a Robot

Robot has to be configured for ROS2 and Nav2 ← how to adapt a custom Robot for the navigation stack ?

We are going to see just an overview of the steps needed, NOT a step by step tutorial (the entire process is quite complicated)

↳ when configuring a Real robot for ROS2 and Nav2 this may take a while...

WHAT ARE THE STEPS ? ↳ Let's look at Nav2 Architecture:




the inner part is already implemented

we need to provide the IN/OUT for the Navigation

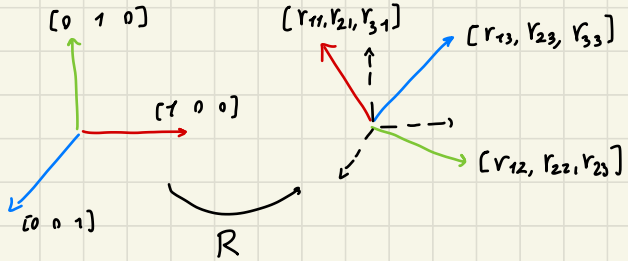
we will focus on

1) TF you need to create
2) odometry + other sensor data to publish on specific topics
3) output is Hardware specific, Controller that takes cmd_vel and translate it in real motor commands

⇒ once this steps are completed, we can start Navigation for your own robot with properly created
- launch files
- parameters

# TF / URDF

Very quickly, TFs: coordinate frames of your robot links.
We need it to keep track of each frame of Robot and environment relative to each other...




ex: where is one robot with respect to another robot ?
where is a laser scan respect robot chassis ?

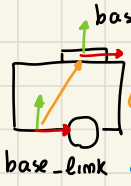
⇓

3D ROTATION + TRANSLATION, managed by Tf2 package

{ Keep track of each 3D coordinate frame as structured tree. }

In particular, for Nav2 we need:

1) map → odom  published by localization feature
2) odom → base_link
3) base_link → base_scan  ⤵ let's focus on this



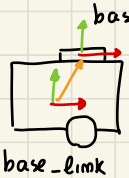

(where LIDAR is situated, where laser scan come from)
we are interested in the TF between
(attached to main Robot structure, chassis)
usually first frame you design

where is laser scan relative to Robot origin

⇓

Without this imformation, How can Nav2 knows where are the obstacles detected with LIDAR ?

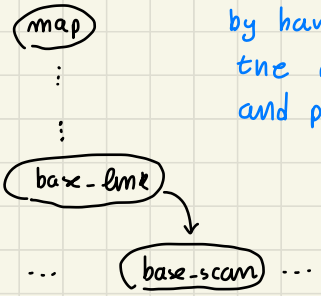
The frames can be at the bottom of the link or at the center




NOT an issue, frames remain all related, the TF is managed by tf2 package

Be sure that base_scan frame is correctly placed where is the LIDAR!

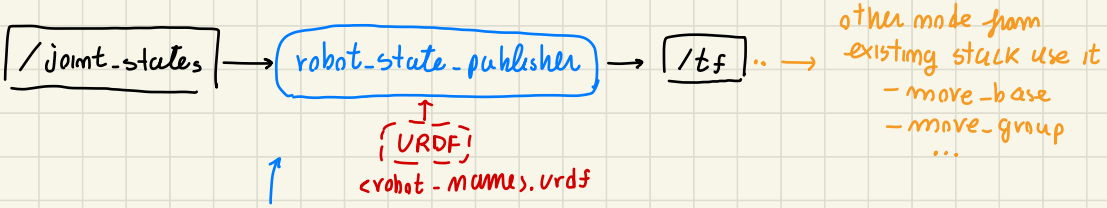
Im the TF tree




by having this tree, Navigation can compute the map → base_scan TF to use Lidar Data correctly, and place the obstacles correctly on the map frame

To create those TFs we rely on the Robot URDF File, to describe all elements and frames of it
(Unified Robot Description Format)
- used by other packages to control the Robot
- it is an XML File
- contain different frames description
  ⇐ including base_link, base_scan and their relationship

Creating TFs ?
we don't need to create TF ourself,
once the robot URDF has been created, a node robot_state_publisher takes care of broadcasting TFs in /tf topic




this receive URDF and joint_states data (published by controller)
as input, then it
computes and publish
Transforms of Robot, then used by Nav 2 stack
for ex: what is wheels position

EXAMPLE, URDF: once we have a URDF file `my_robot.urdf`, we can visualize it in Rviz2

⇓

It is possible to visualize it easily from an existing package `urdf_tutorial` and launch

`urdf_tutorial display.launch.py model:=<robot_name>.urdf`

↓

this will start Rviz with visualization of the Robot

Inside `<robot_name>.urdf`, written in XML format, we have:
all under `<robot>` we will have frames under `<link>` tags, and joints under `<joint>` tag.

characterized by `<parent>` and `<child>`, with explicit relationship in x y z translation, r p y rotation

...

```
<!-- base_link and base_scan (required for Nav2) -->

<link name="base_link">
    <visual>
        <geometry>
            <box size="0.6 0.4 0.2" />
        </geometry>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <material name="blue" />
    </visual>
</link>

<joint name="base_scan_joint" type="fixed">
    <parent link="base_link"/>
    <child link="base_scan"/>
    <origin xyz="0 0 0.13" rpy="0 0 0"/>
</joint>

<link name="base_scan">
    <visual>
        <geometry>
            <cylinder radius="0.1" length="0.06"/>
        </geometry>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <material name="grey" />
    </visual>
</link>
```

↵ numerical values will depends on hardware characteristic

refer to URDF documentation for further details...

...

this is a standard package name, containing URDF and 3D meshes

IF COURIOS, in `/opt/ros/humble/share/turtlebot3_description/urdf/<models>.urdf` you can see the URDF of the turtlebot3 used until now

# INPUT/OUTPUT - ODOMETRY, SENSORS and CONTROLLER

- odometry
- sensors
- controller : how to process output (HW controller, being close-loop, this give back some data, so it is both IN/OUT)

} remaining inputs of nav stack

## • ODOMETRY

very important for navigation, help us to compute and publish

**odom → base_link**

↓

it is used to localize a robot from its starting point, using its own motion

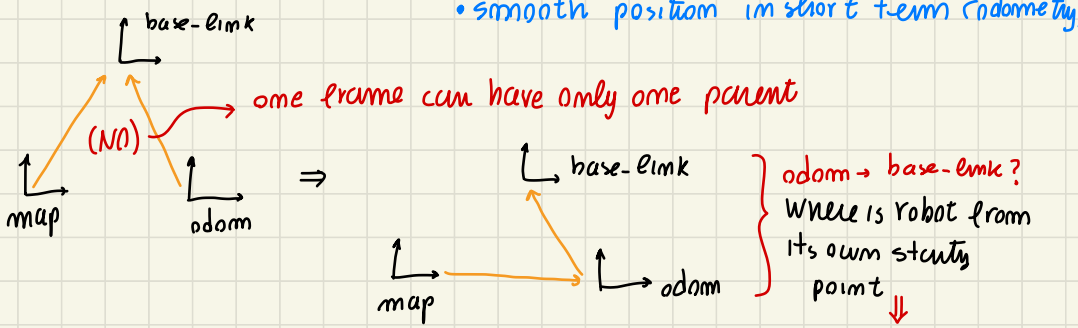
ex. walking an localizing yourself by counting steps... this introduce an error which cumulate with each step

↳ error accumulation = "DRIFT"

- odometry will DRIFT over time and/or distance (depending on data/sensor used)

EVEN if drifting occurs → it will be compensated with map frame by nav stack

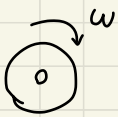
As seen, we will have both
- long-term precise location (map)
- smooth position in short term (odometry)




Let's start by looking the most basic odometry you can compute, by using **wheel encoders**

tells you at what velocity wheels turn

**Wheel encoders**

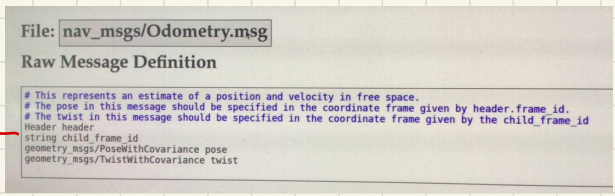



by doing simple integration, you can also compute position over the distance

⇓ we have 2 options:

**OPTION 1)**

- Read velocity and compute position yourself
- publish nav_msgs/Odometry type on the /odom topic, (creating a topic publisher)



File: nav_msgs/Odometry.msg

Raw Message Definition

```
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given by header.frame_id.
# The twist in this message should be specified in the coordinate frame given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

first you provide info about between which frames you are publishing odometry as odom in header.frame_id
base_link in child_frame_id

+ pose field (position computed)
++twist (velocity computed from encoders)

→ you will publish to /odom topic and also ↓

- odom → base_link TF on /tf topic must be published (TF broadcaster)

**OPTION 2)** Instead of doing all by yourself, you can use a framework such as ros2_control

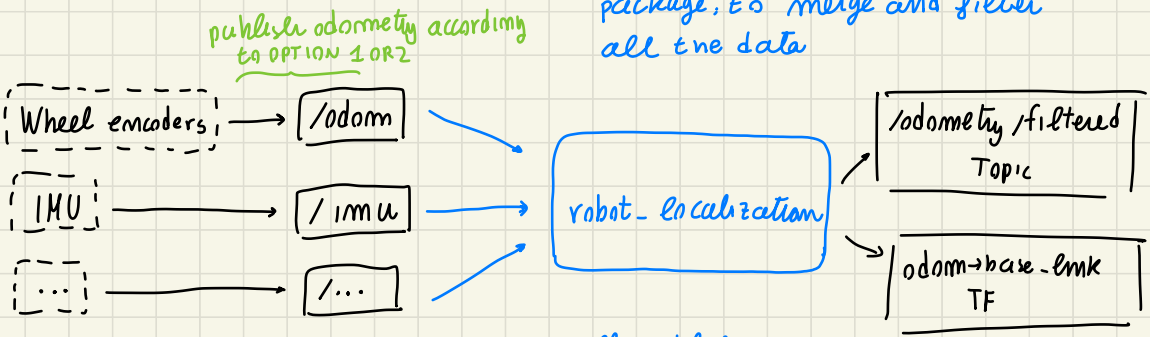
this framework allow you to bridge output of nav stack and your hardware...

for ex: diff_drive_controller in a diff drive robot does all this for you (you need just to configure it)

to use ros2_control you need to understand it and adapt your Robot to be used with it

⇑ those options are valid when we use just Wheels encoder...
We may have several sensors!
Many sources to compute a better and more precise odometry

IF you work with many sensors: → you can use an already existing package, to merge and filter all the data




Will publish new odom to /odometry/filtered topic + odom→base_link transforms

- **SETUP SENSORS** What additional sensor Data Nav stack need as input?

Obviously it will depends on what sensors you have on your Robot!

As minimal setup
- Wheel encoders (used for odometry)
- LIDAR

} are enough, more sensors can increase navigation precision

IF LIDAR available: read the data from it and publish on /scan topic (with message type sensor_msgs/msg/LaserScan

ADVICE:
use LIDAR ROS 2 compatible, meaning that code to read data and publish LaserScan is already available

FOR each available sensor, follow a similar procedure

ex: CAMERA
- create camera_link in URDF
- publish to /camera/image_raw the data
- use sensor_msgs/msg/Image type

similar for IMU, GPS ... If ROS2 compatible, there is already a package that connects to sensor and publish correct data on correct topic

- Nav STACK OUTPUT : HARDWARE CONTROLLER

⇓

the OUTPUT is a velocity command that has to be translated into something that motors can understand

usually published on /cmd_vel of type geometry_msgs/msg/Twist

contain both linear/angular velocity ←

**geometry_msgs/Twist Message**

File: geometry_msgs/Twist.msg

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
Vector3  angular
```

To setup the hardware controller, we have two options:

OPTION 1)
- create your own custom hardware controller
  - find a way to send correct velocity signal to motors, reading encoders velocity to create closed-loop control system to ensure correct velocity is actuated
  - + publish on top: current velocity and position of Robot, task of odometry

↑

make sure Robot follow cmd_vel and publish encoders data

the implementation of OPTION 1 strongly depends on used motors and encoders... Hardware characteristic

OPTION 2) • rely on ros2_control

ex: diff_drive_controller take care of close-loop control, publishing odometry and required TF for odom

+ write your code interface to communicate with motors/encoders

↓

all closed-loop control logic is handled by ros2_control

NOTICE: using ROS2 compatible HW to build your robot makes life easyer!

# RUN NAVIGATION WITH YOUR CUSTOM ROBOT (using slam_toolbox)

We Learn how to configure your Robot for Nav stack !

once properly configured, HOW TO START SLAM and Navigation?

FOR NOW we see how to do it from terminal, using existing Ros 2 packages, later we will organize it in a launch file

- SLAM, for map generation

with turtlebot3 we use cartographer.launch.py ....
NOW we want to start general SLAM for any Robot
↓
we install ros-<distro>-slam-toolbox

in fact, we use slam_toolbox to generate map
The requirements is that LASERSCAN Data has to be published on /scan topic

(TERMINAL 1)

- FIRST, start simulation environment, launching ROBOT stack
  you can control and move over the environment,
  This has nothing to do with Navigation stack, just properly set up robot

for now we use turtlebot3_gazebo simulation

turtlebot3 publish on /scan topic, a sensor_msgs/msg/LaserScan which is requirement from SLAM toolbox

- Then, start SLAM by launching
  - nav2_bringup navigation_launch.py
    IF working in simulation
    use_sim_time := True

↑
we need to start Nav stack (TERMINAL 2)

- and in TERMINAL 3: start the SLAM functionality with SLAM toolbox launching

```
slam_toolbox online_async_launch.py use_sim_time:=True
```

this start SLAM process to generate a map

- than, to visualize what is going on, TERMINAL 4: start Rviz2

and properly configure Rviz2 by Adding all the elements we want to visualize:

- > TF
- > Map specify the Topic as /map
- > LaserScan specify the /scan topic
- > Robot Model and select /robot_description as topic

recreate similar view as previously used mapping tools

- THEN, run the teleoperation node!
  when set-up your Robot, be sure to have a way to move robot

```
turtlebot3_teleop teleop_keyboard
```

by moving around, map will be generated (as before!)

- FINALLY, to save the map, keep all terminals open, and run in another terminal

```
nav2_map_server map_saver_cli -f <path/map_name>
```

As Recap: to perform SLAM

1) start Robot stack (simulation of your specific Robot)
2) start Nav2 stack
3) start SLAM toolbox
4) start Rviz 2 and configure → >File> Save config as to reuse this configuration
5) start teleoperation node
6) save the map

↑ this perform same process as cartographer.launch.py of turtlebot3, This STEPS can be implemented for any Robot! NOT specific to any Robot

⇓

at the end, we will have an available map.pgm, map.yaml and we are ready for navigation

- NAVIGATION, make any Robot navigate

- As before, FIRST: launch the Robot stack
  turtlebot3_gazebo

- start Nav2 stack, launching

nav2_bringup bringup_launch.py use_sim_time := True map:= <map>.yaml

(IN SLAM we used navigation_launch.py)

↓

- start Rviz2 for visualization rviz2 and configure by Adding
  > LaserScan /scan
  > TF
  > Robot Model
  > Map /map → change "Durability Policy" to Transient Local to see the map

(some errors remain until we define the 2D Pose estimate)
on TF due to localization

to add also costmaps > Map with Display name "Global costmap" and chose /global_costmap/costmap
change colorscheme to costmap
to visualize properly colors

> Map Display name "Local cost map"
topic /local_costmap/costmap
chosimy costmap as color scheme

↓

Now > File > Save config as

You are ready to send 2D Goal Pose

To visualize also global path planned, you need to configure Rviz2 furthermore

# SLAM and Navigation Steps for Any Robot (ROS2 Nav2 Course - Section 7)

Those commands are the general commands to run for SLAM and Navigation, when using any robot that is configured for the Nav2 stack.

## Steps - SLAM

You will need to install the slam_toolbox package:
*$ sudo apt install ros-humble-slam-toolbox*

### 1. Start your robot

This will be specific to your own robot.
Example with simulation:
*$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py*

### 2. Start a Navigation launch file

*$ ros2 launch nav2_bringup navigation_launch.py*
(add *use_sim_time:=True* if using Gazebo)

### 3. Start SLAM with slam_toolbox

*$ ros2 launch slam_toolbox online_async_launch.py*
(add *use_sim_time:=True* if using Gazebo)

### 4. Start Rviz

*$ ros2 run rviz2 rviz2*
(you will need to configure RViz, follow the instructions in the video)

### 5. Generate and save your map

Make the robot move in the environment (specific to your own robot).
Example with simulation:
*$ ros2 run turtlebot3_teleop teleop_keyboard*
Save the map:
*$ ros2 run nav2_map_server map_saver_cli -f ~/my_map*

# Steps - Navigation

## 1. Start your robot

This will be specific to your own robot.
Example with simulation:
*$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py*

## 2. Start the main Navigation2 launch file

*$ ros2 launch nav2_bringup bringup_launch.py map:=path/to/map.yaml*
(add *use_sim_time:=True* if using Gazebo)

## 3. Start RViz

*$ ros2 run rviz2 rviz2*
(you will need to configure RViz, follow the instructions in the video)

## 4. Send navigation commands

Use the “2D Pose Estimate” button to set the initial pose, and the “Nav2 Goal” button to send navigation goals.

Note: instead of using RViz to send commands, you can directly interact with the Nav2 interfaces in your own code, for example using the Simple Commander API (see Section 8 of the course)

## LAUNCH FILE + PARAMETER

We want to perform the steps previously defined for SLAM and navigation NOT one by one BUT from one launch file!

we don't need to write those launch from scratch:
Use turtlebot3 launch files and adapt for our custom Robot

here we will NOT create our own package and launch, instead we see where to find the template to use...

**SLAM launcher** Let's look at

turtlebot3_cartographer  cartographer.launch.py

in /opt/ros/humble/share/turtlebot3_cartographer/launch

look at that launch file contents... adapting it is NOT complex!

```
def generate_launch_description():
    use_sim_time = LaunchConfiguration('use_sim_time', default='false')
    turtlebot3_cartographer_prefix = get_package_share_directory('turtlebot3_cartographer')
    cartographer_config_dir = LaunchConfiguration('cartographer_config_dir', default=os.path.join(
                                                  turtlebot3_cartographer_prefix, 'config'))
    configuration_basename = LaunchConfiguration('configuration_basename',
                                                 default='turtlebot3_lds_2d.lua')

    resolution = LaunchConfiguration('resolution', default='0.05')
    publish_period_sec = LaunchConfiguration('publish_period_sec', default='1.0')

    rviz_config_dir = os.path.join(get_package_share_directory('turtlebot3_cartographer'),
                                   'rviz', 'tb3_cartographer.rviz')

    return LaunchDescription([
        DeclareLaunchArgument(
            'cartographer_config_dir',
            default_value=cartographer_config_dir,
            description='Full path to config file to load'),
        DeclareLaunchArgument(
            'configuration_basename',
            default_value=configuration_basename,
            description='Name of lua file for cartographer'),
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use simulation (Gazebo) clock if true'),

        Node(
            package='cartographer_ros',
            executable='cartographer_node',
            name='cartographer_node',
            output='screen',
            parameters=[{'use_sim_time': use_sim_time}],
            arguments=['-configuration_directory', cartographer_config_dir,
                       '-configuration_basename', configuration_basename]),
```

some configurations as parameters ecc

] where is your custom rviz configuration

specific to cartographer, if using slam_toolbox you don't need this

cartographer node, we don't need it

```
        DeclareLaunchArgument(
            'resolution',
            default_value=resolution,
            description='Resolution of a grid cell in the published occupancy grid'),

        DeclareLaunchArgument(
            'publish_period_sec',
            default_value=publish_period_sec,
            description='OccupancyGrid publishing period'),

        IncludeLaunchDescription(
            PythonLaunchDescriptionSource([ThisLaunchFileDir(), '/occupancy_grid.launch.py']),
            launch_arguments={'use_sim_time': use_sim_time, 'resolution': resolution,
                              'publish_period_sec': publish_period_sec}.items(),
        ),

        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            arguments=['-d', rviz_config_dir],
            parameters=[{'use_sim_time': use_sim_time}],
            output='screen'),
    ])
```

start a new launch file using cartographer specific

start rviz as node + config

we can use this template to start nav2 and slam_toolbox launcher

Once you modify that template, you can then launch it in your custom package

## Navigation Launcher

with turtlebot3, we used to launch `turtlebot3_navigation2` `navigation2.launch.py`

again it is located in `/opt/ros/humble/share/turtlebot3_navigation2/launch/...`

we can open that one:

```
def generate_launch_description():
    use_sim_time = LaunchConfiguration('use_sim_time', default='false')
    map_dir = LaunchConfiguration(
        'map',
        default=os.path.join(
            get_package_share_directory('turtlebot3_navigation2'),
            'map',
            'map.yaml'))

    param_file_name = TURTLEBOT3_MODEL + '.yaml'
    param_dir = LaunchConfiguration(
        'params_file',
        default=os.path.join(
            get_package_share_directory('turtlebot3_navigation2'),
            'param',
            param_file_name))

    nav2_launch_file_dir = os.path.join(get_package_share_directory('nav2_bringup'), 'launch')

    rviz_config_dir = os.path.join(
        get_package_share_directory('nav2_bringup'),
        'rviz',
        'nav2_default_view.rviz')

    return LaunchDescription([
        DeclareLaunchArgument(
            'map',
            default_value=map_dir,
            description='Full path to map file to load'),

        DeclareLaunchArgument(
            'params_file',
            default_value=param_dir,
            description='Full path to param file to load'),

        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use simulation (Gazebo) clock if true'),

        IncludeLaunchDescription(
            PythonLaunchDescriptionSource([nav2_launch_file_dir, '/bringup_launch.py']),
            launch_arguments={
                'map': map_dir,
                'use_sim_time': use_sim_time,
                'params_file': param_dir}.items(),
        ),

        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            arguments=['-d', rviz_config_dir],
            parameters=[{'use_sim_time': use_sim_time}],
            output='screen'),
    ])
```

map configuration parameter

← parameter file for navigation

find nav2_bringup and launched + rviz2 config

start bringup.launch as we need for custom navigation

↓ to use this for a custom Robot, there is NOT a lot to modify! It is possible to easily adapt for your Robot

an important part to consider is the parameter file

⇓

by looking at it in

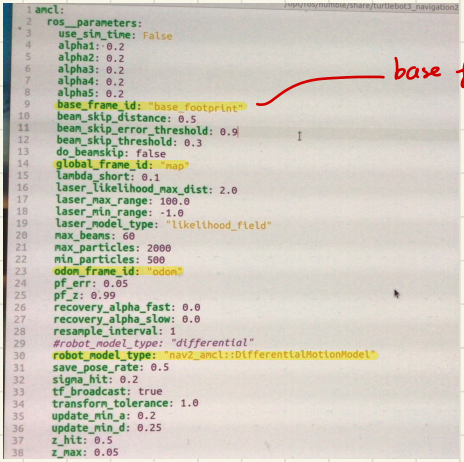
`opt/ros/humble/share/turtlebot3_navigation2/param/...`

⇐ `<model>.yame`

this is the parameter file you need for your Robot with a LOT of parameters, you will create by copying an existing template and adapt

the most important parameters to understand on this config file are:

in amcl:



```
amcl:
  ros__parameters:
    use_sim_time: False
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    alpha5: 0.2
    base_frame_id: "base_footprint"
    beam_skip_distance: 0.5
    beam_skip_error_threshold: 0.9
    beam_skip_threshold: 0.3
    do_beamskip: false
    global_frame_id: "map"
    lambda_short: 0.1
    laser_likelihood_max_dist: 2.0
    laser_max_range: 100.0
    laser_min_range: -1.0
    laser_model_type: "likelihood_field"
    max_beams: 60
    max_particles: 2000
    min_particles: 500
    odom_frame_id: "odom"
    pf_err: 0.05
    pf_z: 0.99
    recovery_alpha_fast: 0.0
    recovery_alpha_slow: 0.0
    resample_interval: 1
    #robot_model_type: "differential"
    robot_model_type: "nav2_amcl::DifferentialMotionModel"
    save_pose_rate: 0.5
    sigma_hit: 0.2
    tf_broadcast: true
    transform_tolerance: 1.0
    update_min_a: 0.2
    update_min_d: 0.25
    z_hit: 0.5
    z_max: 0.05
```

base frame of your Robot, if you are using as base the base_link, you use it. It should be the FIRST FRAME of your URDF

... in controller_server: there are lot of parameters, some interesting one are

max_vel_x
max_vel_y
max_vel_theta
acc_lim_x
....

} to define a speed limit

... in local_costmap: most important are

robot_radius used to compute collision free path
resolution

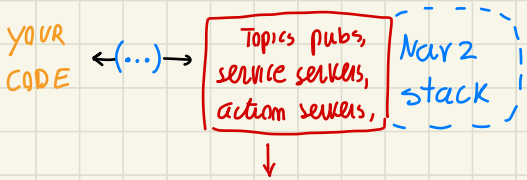
and the same for global_costmap

.... experiment with those parameters and understand which one you need to modify for your own Robot!

# 8) INTERACT PROGRAMMATICALLY WITH THE NAVIGATION STACK

So far, we use Rviz 2 to select initial pose and send Nav2 Goal
this is fine for testing → NEXT step is to automate this!

The objective is to interact programmatically with Nav2, to integrate Nav2 functionalities directly in your code and inside your ROS2 node → to use Nav stack directly from your custom ROS2 application



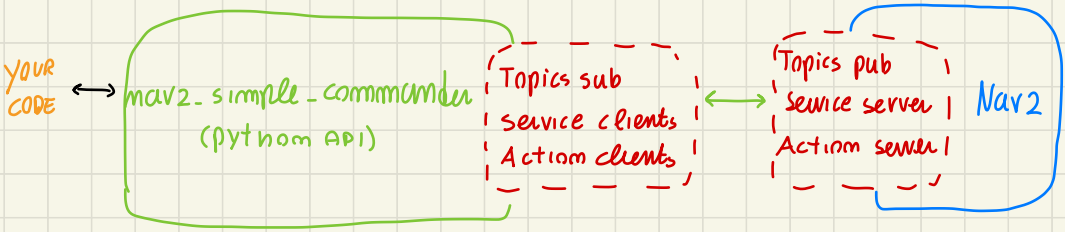

this is possible, in fact Navigation stack works by using topics, services, actions and so on...

- When you set 2D Pose Estimate in Rviz, behind the scene it has been used a topic
- When Nav2 Goal is sent, we use an action

We already have an INTERFACE to Nav2 stack, we need just to see which topic, services, actions are used and interact with it from your ROS2 code... → there is an easy to way to do it, by using an API developed for this:

nav2_simple_commander (python API)

let's you communicate with Nav2 stack with few lines of basic python code → all you need to do is to use this API in your own code to set initial pose, send nav2 Goal from your code




In this lecture we will:

- see what are those topics/actions used by Nav2
- learn how to use simple_commander API to interact programmatically with Nav2 stack

## DISCOVER WHAT TOPICS and ACTIONS ARE USED

FIRST, let's understand what are the ROS2 interfaces used when 2D Pose Estimate is set or Nav2 Goal is sent.

↓

this give you a global understanding, to know what happens behind the scene and use smarter the nav2 API

With turtlebot3_gazebo and nav2 running, we can visualize the network communication topics:

There are a lot of topics,

- for 2D Pose Estimate, we are interested on /initial_pose topic

↓

in which you publish the Initial Robot pose of type

geometry_msgs/msg/PoseWithCovarianceStamped

↑ defined as Pose (position + orientation) stamped with time

So, to set initial pose without using Rviz2, you need a publisher on this topic

by listening on this /initial_pose topic, when sending 2D Pose Estimate we can see how this message is formatted

this will have:

```
ed@pc:~$ ros2 topic info /initialpose
Type: geometry_msgs/msg/PoseWithCovarianceStamped
Publisher count: 1
Subscription count: 1
ed@pc:~$ ros2 topic echo /initialpose

header:
  stamp:
    sec: 139
    nanosec: 778000000
  frame_id: map
pose:
  pose:
    position:
      x: -0.009042748250067234
      y: 0.011425078846514225
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.012668822547163956
      w: 0.9999197472473821
  covariance:
  - 0.25
```

shape of initial pose

it is given as QUATERNION orientation

.... (covariance data)

⇒ quaternions are used instead of RPY angles because simpler to compute and manipulate

(it is possible to easily convert an euler orientation into quaternions)

- for Nav2 Goal, we need to refer to available actions,

especially /navigate_to_pose used when single Nav2 Goal Pose given

/follow_waypoints used when waypoint Follower is called

by looking at the characteristic of those actions

```
ed@pc:~$ ros2 action info /navigate_to_pose
Action: /navigate_to_pose
Action clients: 4
    /rviz2
    /rviz2
    /bt_navigator
    /waypoint_follower
Action servers: 1
    /bt_navigator
```

→ it is behavior tree navigator, which is inside Nav2 stack. This will call different global/local planner

Sending this messages, actions from terminal, it is complex due to the amount of fields we need to fill!

- OPTION 1

this can be done from a custom node to publish on topic, create an action client

- OPTION 2

use a simplified python API nav2_simple_commander

# SIMPLE COMMANDER API - INSTALL AND SET THE INITIAL POSE

Now, we know the different ROS2 interfaces used to communicate with Nav2 stack, we want to use Python code to send Navigation commands! => 1ST step: set-up API and use it to send initial Robot pose...

install `ros-<distro>-nav2-simple-commander`

Then, create a new python scipt `nav2_test.py` wherever you want

remember to make it as executable: `chmod +x nav2_test.py`
and open with your text editor
(for example using VSCode)

Here, we will run the code from this python scipt, it is easy to create it as a py package, including this code in your own node...

`nav2_test.py`

FIRST we need to initialize ROS communication (before using any ROS library) by `import rclpy`

```
...
def main():
    rclpy.init()
    ...
    rclpy.shutdown()
```

We also need to initialize Nav2 simple commander API by importing `BasicNavigator`, then create an object of that type. This class contain all we need to interact with Nav2

start the scipt with `#!/usr/bin/env python3` to set the interpreter and run it from terminal

```python
#!/usr/bin/env python3
import rclpy
from nav2_simple_commander.robot_navigator import BasicNavigator

def main():
    # --- Init
    rclpy.init()
    nav = BasicNavigator()

    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

~> ./nav2_test.py to check if library is properly installed and NO error occurs

IF you have include error, be sure the library is properly installed

once ROS2 communication is initialized and BasicNavigator() object exists:

We can set initial pose easily by using nav giving PoseStamped object to it!

calling setInitialPose() method with nav,

we can send as message a PoseStamped, inside the method, this is properly converted to PoseWithCovarianceStamped which is the one used by Nav2 stack

```
initial_pose = PoseStamped()
initial_pose.header.frame_id = 'map'
initial_pose.header.stamp = nav.get_clock().now().to_msg()
initial_pose.pose.position.x = 0.0
//                //    //  .y = 0.0
//                //    //  .z = 0.0
//                //  .orientation.x =
//                //    //    //  .y =
//                //    //    //  .z =
//                //    //    //  .w =
```

← because we want to give pose relative to the map frame

← to take the time

we will fix x,y,z real position later on

orientation is given as (x,y,z,w) as QUATERNION, it is easyer to give as EULER RPY and convert it using a proper library

to compute x,y,z,w just install

- ros-<distro>-tf-transformations
- python3-transforms3d

then import tf_transformations inside nav2_test.py

and use the method `tf_transformations.quaternion_from_euler()`
and assign it to
q_x, q_y, q_z, q_w
and use it to fill the orientation fields

{ quaternion (x,y,z,w) from euler (x,y,z) }

finally we are ready for: `nav.setInitialPose(initial_pose)`

And then we need Nav2 to be active, to be sure initial pose is set, once `nav2.waitUntilNav2Active()`
we are sure Initial Pose is set up correctly!

We are ready to run this, first of all let's start the Robot and Nav stack for example by using the usual turtlebot3 packages

Launching `turtlebot3_gazebo` and `turtlebot3_navigation2` packages
⇓
by looking in Rviz2 you can see what is going on when using the nav2 API from your code.

NOW you can run `nav2_test.py`
be sure ROBOT initial pose is sent as the correct one, in our case, pose.position is at map origin (0,0,0)
and pose.orientation is RPY = (0,0,0), NO ORIENTATION

This will set Nav2 Pose correctly!
So, we can interact with Nav2 without using Rviz2, in fact if we do the same using `nav2_bringup` package, without Rviz, by running our script we can set initial pose correctly!

NEXT STEP: Use the API to send Nav2 Goal to the Robot!

---

NOTE IF you have errors with import `tf_transformations`, try to reinstall and upgrade the library with

`pip install --upgrade transforms3d`

# SIMPLE COMMANDER API - SEND A NAV2 GOAL

Let's continue to write our script nav2_test.py, now to give nav2 Goal!
To test the API, for example we can define

Goal as { x = 3.5, y = 1.0, z = 0.0
R = 0.0, P = 0.0, Y = 1.57

move to (3.5, 1) in xy plane with 90° orientation on the left

⇓

to send a Goal we need to:

1) create a goal_pose as PoseStamped()
Initialize this as before, with proper header and pose
for example position.x = 3.5
position.y = 1.0
orientation.yaw = π/2 = 1.57


using right hand anti-clock wise convention
+ orientation positive

2) use method nav.goToPose(goal_pose)
This JUST send the pose, but we need to wait for navigation finish:
we can wait for navigation to goal by

⇓

3) While not nav.isTaskComplete():
this method return True if task given is completed

feedback = nav.getFeedback() ← we get current position as feedback just to check what is going on
print(feedback)

We can now launch the Robot stack (Gazebo) and nav2, and run the script in another terminal

BUT: if we try to re-run nav2_test.py... the initial pose will be again set up to (0,0,0) map, and it will mess up everything!

↓

the initial pose is wrongly set-up

Be carefull! once initial pose is setted once, DON'T try to set it again

Next time you run it, don't use nav.setInitialPose(initial_pose)
We can manage it by adding main() arguments, to
set initial_pose true or false when we run from terminal
(by using argv, argc of main...)

We can also get the final result of the navigation, by calling
nav.getResult() after the wait!
You can see if task is SUCCEDED or ABORTED

# SIMPLE COMMANDER API - WAY POINT FOLLOWER

Let's see how to use the API to send some waypoints using waypoint follower mode directly from our script

⇓

OPTION 1: use goToPose method in eoop, for any desired goal

OPTION 2: use followWaypoints method, specific for this task

↓

this require a list of goal poses, each as PoseStamped() as before

↓

to make the code more effcient, let's define a function that return a pose stamped instead of rewriting all the code any time for any pose...

required to get the timestamp

def create_pose_stamped(navigator, position_x, position_y, orientation_z)

only pose info that can be specified in a 2D plane

here the code is the same used to define goal_pose before, using the arguments properly in the initialization, finish with a return pose

Then, we can use this function to make the code cleaner. Using it to create the initial_pose as x=0.0, y=0.0, yaw=0.0
And also to define a nav2 Goal with proper x, y, yaw arguments

↓

so, I can use create_pose_stamped(nav, x, y, yaw) to define multiple goal_pose i  i=1,2,3,...,N

store those poses in an array waypoints = [goal_pose1, goal_pose2, ...] and finally call

nav.followWaypoints(waypoints)

remember than to: while not nav.isTaskComplete() as before...

Then you can run the code and way point follower functionality will be started...

Always be carefull of avoidim reinitialize Pose in (0,0,0) after runnimg the first time...

NOTICE: I personally decided to customize the code to avoid commentimg out part of the scupt anytime.
I manage the functionality from the argoument given when runnimg the scupt in the terminal:

» ./nav2_test.py argoument
this will be argv[1]

- Initialize: only the first time to set initial Pose
- Goal: to send a simple Goal
- Waypoints: to send multiple Goal followed with waypoint follower

It is easy to further modify the scupt to give the goal pose [x y yaw] as argv arag ouments...

## • ASSIGNMENT 4: SIMPLE COMMANDER API - CREATE A ROBOT PATROL

Practice a bit with this API.
usimg turtlebot3_house as Robot stack, simulation environment
previously we map it and patrol araund usimg Rviz Nav2 Goal

Task: stant simulation + Nav2 stack (with / without Rviz)
then, with a custom pythom scupt set initial Robot pose and Patrol araund the house environment

**SOLUTION:** the solution is easy, launch turtlebot3_house.launch.py and navigation2.launch.py, choose some points (x, y, yaw) in each room (by looking at Rviz)
choose waypoints wisely to cross doors without "U-shape" turn that doesn't work very well with turtlebot3 navigation (by usimg additional points to go through). use the same nav2_test.py scupt, just Redefine the goal_pose i composimg waypoints array
(you can then stant to waypoint follow in loop)

# 9) CONCLUSION

This course provide good/intuitive idea of what is Nav2 stack and How to use it + how to integrate it with your ROS2 application

## WHAT'S NEXT ?

(IDEAS and RECCOMENDATIONS.)

- Apply this knowledge of Nav2 into your own project
  - Create your own gazebo world
  - Add Nav2 to an existing ROS2 Application
    (easy to do with nav2_simple_commander API)
    OR look deeper into interface with Nav2 to create your own API
  - Adapt your ROBOT for Nav2

- Learn more, reccomendation
  - ros2_control super usefull to bridge ROS2 Application with Robot Hardware
  - Robotic Arm with MoveIt2 framework to control Robot Arm
  - Advanced ROS2 concepts actions, lifecycle nodes, components