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

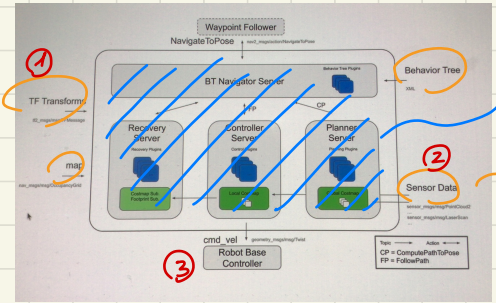
We have seen how Nav2 stack work with SLAM and Navigation + how to create a custom world for a Robot

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

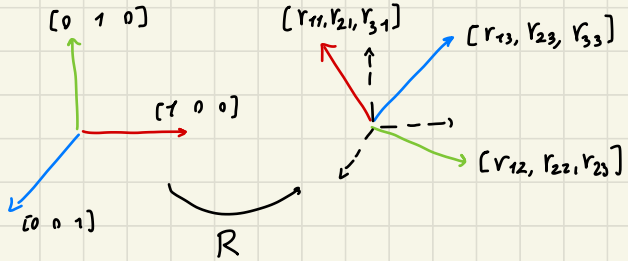
we will focus on

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 ?

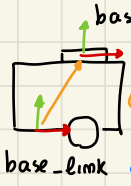
⇓

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



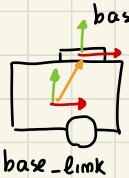
(where LIDAR is situated, where laser scan come from)
we are interested in the TF between
(attached to main Robot structure, chassis)
usually first frame you design

where is laser scan relative to Robot origin

⇓

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

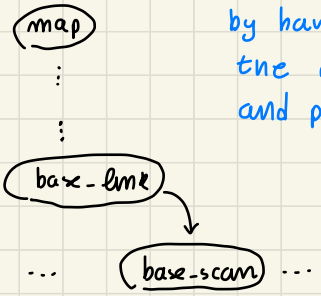
The frames can be at the bottom of the link or at the center



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!

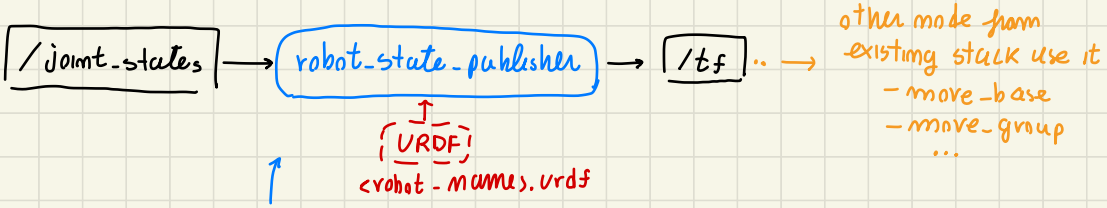
Im the TF tree



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

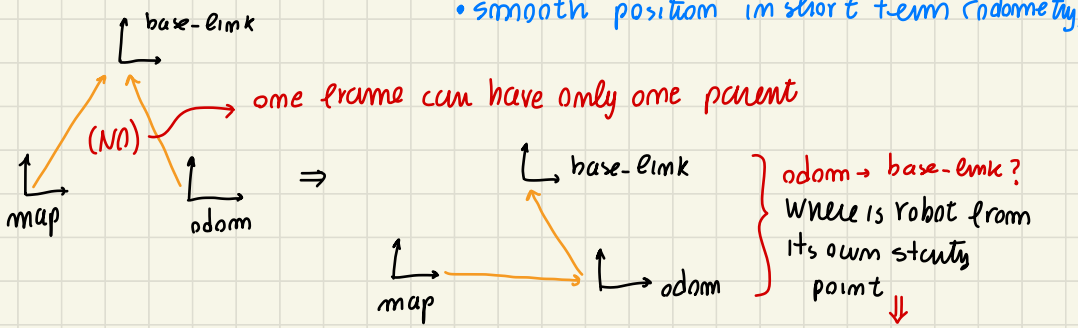
ex. walking an localizing yourself by counting steps... this introduce an error which cumulate with each step

↳ 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

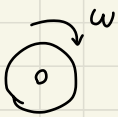
As seen, we will have both
- long-term precise location (map)
- smooth position in short term (odometry)



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

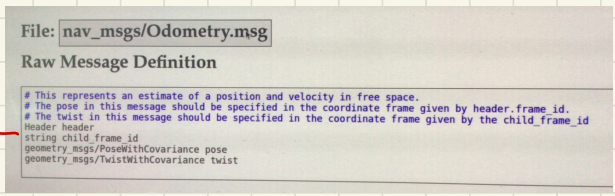


by doing simple integration, you can also compute position over the distance

⇓ 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

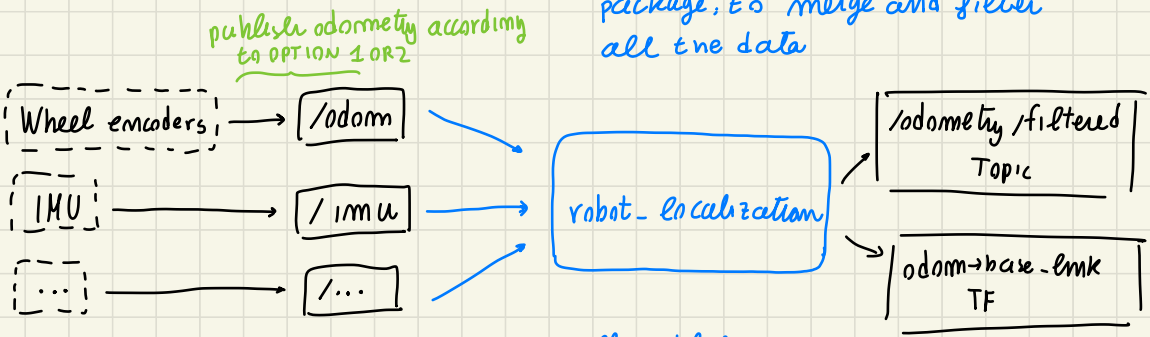
this framework allow you to bridge output of nav stack and your hardware...

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

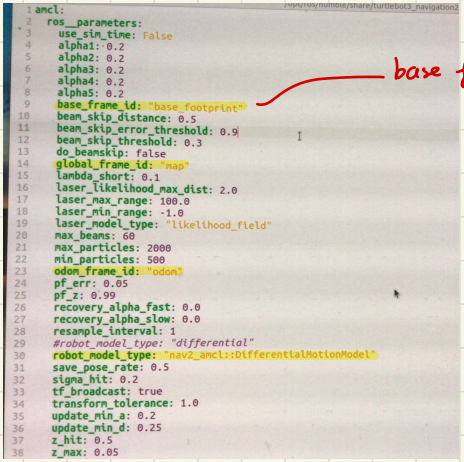
`opt/ros/humble/share/turtlebot3_navigation2/param/...`

⇐ `<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!