ROS 2 NAVIGATION – SLAM TOOLBOX

Agenda

- SLAM2 Toolbox and mapping
- Nav2 stack
- SLAM and navigation, Unity
- SLAM and navigation, Turtlebot3

SLAM TOOLBOX AND MAPPING

What is SLAM

- SLAM (Simultaneous Localization And Mapping)
- SLAM: building or updating a map of an unknown environment
- Different algorithms solve this problem in 3D or in 2D space
  - ORB SLAM2 (3D SLAM)
  - Gmapping (2D SLAM)
- SLAM can be solved using different sensors such as camera (usually 3D SLAM) or lidar (usually 2D SLAM)

SLAM toolbox in ROS2

- ROS2 includes a toolbox for SLAM
- In the toolbox different algorithms are implemented to solve 2D SLAM problem
  - Gmapping (default)
  - Cartographer
  - Hector
  - Karto
- Default SLAM algorithm in ROS2 is Gmapping and we will use it in our courses

SLAM toolbox Gmapping

- GMapping is based on particle filters to learn grid maps from laser range data
- SLAM toolbox has several parameters to specify
  - Type of solver
  - Topic name for sensors
  - …

```yaml
slam_toolbox:
  ros__parameters:

    # Plugin params
    solver_plugin: solver_plugins::CeresSolver
    ceres_linear_solver: SPARSE_NORMAL_CHOLESKY
    ceres_preconditioner: SCHUR_JACOBI
    ceres_trust_strategy: LEVENBERG_MARQUARDT
    ceres_dogleg_type: TRADITIONAL_DOGLEG
    ceres_loss_function: None

    # ROS Parameters
    odom_frame: odom
    map_frame: map
    base_frame: base_footprint
    scan_topic: /scan
    mode: mapping #localization

    debug_logging: false
    throttle_scans: 1
    transform_publish_period: 0.02 #if 0 never publishes odometry
    map_update_interval: 5.0
    resolution: 0.05
    max_laser_range: 20.0 #for rastering images
    minimum_time_interval: 0.5
    transform_timeout: 0.2
    tf_buffer_duration: 30.
    stack_size_to_use: 40000000
    enable_interactive_mode: true
```

Type of solver

Topic info

SLAM execution

- Steps to run SLAM in ROS2
  - Bringup the robot
  - Set correct topic name in SLAM configuration file
  - Run SLAM node with RVIZ2 to see the creation of the map
  - Run teleoperation node to move the robot in the map
  - Save the created map in a local file

Map format

- When you save the map two files will be generated
  - .pgm file with the map
  - .yaml file with map information
    - Path to .pgm image
    - Resolution of the map meters/pixel
    - Origin of the map (x, y, theta)
    - Occupancy Probability threshold: pixels with occupancy probability greater than this threshold are considered completely occupied (and viceversa)

Localization AMCL

Localization

- To localize a robot in the map, ROS2 uses AMCL (Adaptive Monte-Carlo Localization) technique based on a particle filter
- The global localization creates a frame called "odom" and defines the relation between map frame and odom frame
  - Odometry (odom frame) is the estimated robot position in the map based on sensors data
  - Odometry defines the relationship between the odom frame and robot base link
- Usually, the localization is based on the fusion of multiple sensors such as IMU, laser scanner, motor encoder.

Localization

- AMCL has several parameters
  - Map and robot information
  - Type of controller for the robot
    - Our case is differential drive
  - Scan Topic (scan)
- Usually, this parameters are in the navigation configuration file
  - src/turtlebot3_NavSLAM/config/NAV2.yaml

```yaml
amcl:
  ros__parameters:
    use_sim_time: True
    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: "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
    z_rand: 0.5
    z_short: 0.05
    scan_topic: scan
```

NAV 2 STACK

NAV 2

What is NAV2 stack

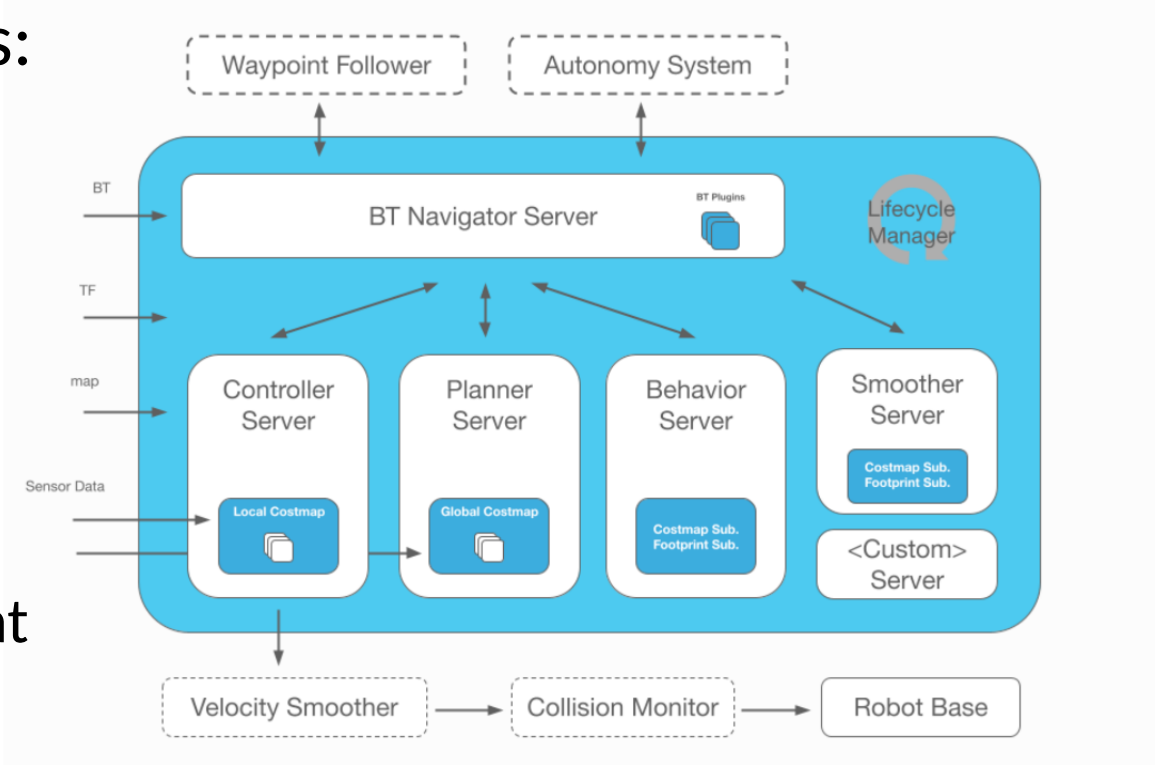
- Nav2 is an official navigation stack for ROS2
- Several toolboxes are included:
  - Map server: loads, manages and stores maps
  - AMCL: localizes the robot in the map
  - NAV2 Planner: plans a path from A to B
  - Nav2 Controller: controls the robot
  - Nav2 Costmap 2D: converts sensor data into a costmap representation of the world
  - …

Costmap

- All implemented algorithms in the navigation stack are based on the costmap concept
- Costmap is a local or global map generated by sensors (e.g laser)
- The costmap includes an occupancy grid that defines where the static or dynamic objects are.
- Global costmap includes all environment, useful for static obstacles (e.g wall, door, table)
- Local costmap includes only the environment detected by the sensors useful to include dynamic obstacles (e.g human, moving objects)

NAV2 stack

- Nav2 is composed by different modules:
  - Behavior tree server
  - Controller Server
  - Planner Server
  - Behavior server
  - Smother server
- Each module will have several parameters that can be set in a configuration file



Behavior tree

- State of the art for robotics tasks
- They are a tree structure of tasks to be completed
- Different concept compared to FSM
- Behavior tree uses primitives concept to execute an action
- A behavior tree node could have some sub-trees to execute a complex action
- The BT aims at calling the sub-module of the navigation stack in order to reach the goal

Controller server

- It is the local motion planner
- Its aim at following a globally computed path avoiding collisions
- The controller will have access to a local environment representation to compute feasible control actions
  - Sensor data
  - Map
- Different plug-in for controller
  - We will use DWBLocalPlanner

DWB Local Planner

- DWB a highly configurable DWA implementation with plugin interfaces
- The aim of local planner is to create a command given, a global plan and a local costmap
- The DWB algorithm simulates different commands and evaluates them on various metrics to select the one with the best score
- Knowing the robot pose, its velocity and its kinematics the algorithm will generate a trajectory with the "safe" waypoints through which the robot will have to pass

Controller server parameter

- The most important parameters for controller server are:
  - Kinematics constraints
  - Goal tolerance
  - Plug-in definition for goal and waypoints checker for DWB algorithm

```yaml
controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    failure_tolerance: 0.3
    progress_checker_plugin: "progress_checker"
    goal_checker_plugins: ["general_goal_checker"] # "precise_goal_checker"
    controller_plugins: ["FollowPath"]
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    general_goal_checker:
      stateful: True
      plugin: "nav2_controller::SimpleGoalChecker"
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
    # DWB parameters
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: True
      min_vel_x: 0.0
      min_vel_y: 0.0
      max_vel_x: 0.26
      max_vel_y: 0.0
      max_vel_theta: 1.0
      min_speed_xy: 0.0
      max_speed_xy: 0.26
      min_speed_theta: 0.0
      acc_lim_x: 2.5
      acc_lim_y: 0.0
      acc_lim_theta: 3.2
      decel_lim_x: -2.5
      decel_lim_y: 0.0
      decel_lim_theta: -3.2
      vx_samples: 20
      vy_samples: 5
      vtheta_samples: 20
      sim_time: 1.7
      linear_granularity: 0.05
      angular_granularity: 0.025
      transform_tolerance: 0.2
      xy_goal_tolerance: 0.25
      trans_stopped_velocity: 0.25
      short_circuit_trajectory_evaluation: True
      stateful: True
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle", "GoalAlign", "PathAlign", "PathDist", "GoalDist"]
      BaseObstacle.scale: 0.02
      PathAlign.scale: 32.0
      PathAlign.forward_point_distance: 0.1
      GoalAlign.scale: 24.0
      GoalAlign.forward_point_distance: 0.1
      PathDist.scale: 32.0
      GoalDist.scale: 24.0
      RotateToGoal.scale: 32.0
      RotateToGoal.slowing_factor: 5.0
      RotateToGoal.lookahead_time: -1.0
```

Plug-in definition

Kinematics

Tollerance

Planner server

- It is the global planner
- Its aims is to compute a valid, and potentially optimal, path from the current pose to a goal pose
- The planner have access to a global environment (map) and sensor data
- Different plug-ins are available
- We will use the Dijkstra planner that implements Dijkstra algorithm

Planner server parameters

- The parameters for planner server are:
  - Planner plug-in type (in this case a grid)
  - Tolerance (safe distance from obstacles)
  - A* solver for particular cases (our case is false)

```yaml
planner_server:
  ros__parameters:
    expected_planner_frequency: 20.0
    use_sim_time: True
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: false
      allow_unknown: true
```

Behavior server

- It is a recovery system to provide a fault tolerant system
- Its goal is to deal with unknown or failure conditions of the system and autonomously handle them
- Examples may include faults in perception resulting in the environmental representation being full of fake obstacles. The clear costmap recovery would then be triggered to allow the robot to move.

Smoother server

- Its purpose is to smooth the trajectories generated by the planner
- The global or local planner often generates a path with several non smooth transitions that could generates abrupt rotations of the robot
- The smoother allows to refine the trajectory near obstacles or high-cost areas in order to find a best way to reach goal
- Use of a separate smoother over one that is included as a part of a planner is advantageous when combining different planners with different smoothers

NAV2 execution

- Steps to run NAV2 in ROS2
  - Bringup the robot for sensors data and odometry
  - Correct parameters in NAV2 configuration file
  - Set up NAV2 nodes to run the stack navigation
  - Initialize the robot position in the map for localization
  - Define the goal for the robot

Exercises

Exercises

1. First exercise will be in Unity
   a) Create a map using SLAM algorithm and save it
   b) Load a saved map and localize the robot
   c) Load a saved map, run the navigation and set the goal for the robot
2. Second exercise will be on real turtlebot3
   a) Set up an arena
   b) Execute the simulated exercise on real robot

Exercise 1

- Check if Unity and ROS2 communicate
  - If not check the first lab lesson slides
- Install xacro (XML Macros)
  - *sudo apt install ros-foxy-xacro*
- Clone the git repository where you want (NavSLAM package added)
  - https://gitlab.com/TrottiFrancesco/mobile_robotics_lab.git
  - Copy the content of "ROS2Package" folder in "*colcon_ws/src*"
  - Move to colcon_ws folder and build workspace (*colcon build*)
  - Source the env (. *instal/setup.bash*)

Exercise 1

- In the "turtlebot3_NavSLAM" package you will find some folders
  - Launch
    - turtlebot3_SLAM launch file to run SLAM node
    - turtlebot3_AMCL launch file to run AMCL localization
    - turtlebot3_NAV2 launch file to run NAV2 stack
  - Config
    - SLAM_param.yaml
    - NAV2_param.yaml
  - Map
    - Where the SLAM map is saved

Exercise 1

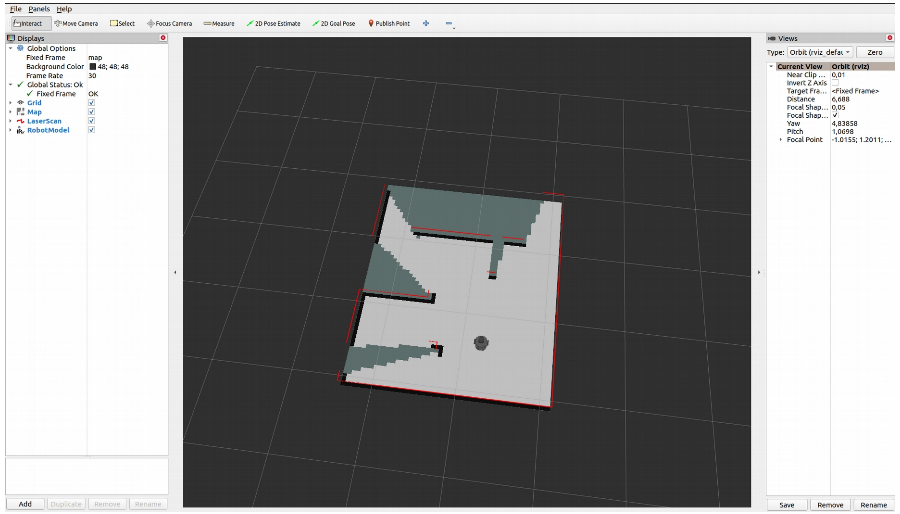
- In the "turtlebot3_visualizer" package you will find some folders
  - Launch
    - turtlebot3_visualizer launch file to load RVIZ2 config
  - RVIZ
    - RVIZ2 config file
- In the "turtlebot3_description" package you will find the .urdf robot description with links and joints descriptions
  - The .urdf file is loaded from *turtlebot3_visualizer* package

Exercise 1

- Update unity project with the newly cloned version
  - Two ways to do this
    - Delete old project and import the new project from Unity Hub
    - In old project directory replace all sub-folders with the sub-folders of the new project (copy and paste)
- This update enables the autonomous navigation

Exercise 1.a - SLAM

- run the SLAM algorithm in order to map all the environment
- Move robot in teleoperation
- After you have mapped all the environment you have to save the map
- The result will be something like the image but hopefully better!



Exercise 1.a - SLAM

- In "turtlebot3_SLAM.launch.py"
  - Parameters declaration (currently they are already set):
    - use_sim_time: false
    - Slam_param_file: path for the slam config file
    - rviz_config: path for rviz settings
  - Nodes
    - robot_state_publisher: define the robot state
    - SLAM algorithm with the config file defined before

```python
def generate_launch_description():
    use_sim_time = LaunchConfiguration('use_sim_time')
    slam_params_file = LaunchConfiguration('slam_params_file')
    rviz_config_file = LaunchConfiguration('rviz_config')

    path_to_urdf = get_package_share_path('turtlebot3_description') / 'urdf' / 'turtlebot3_burger.urdf'

    declare_use_sim_time_argument = DeclareLaunchArgument(
        'use_sim_time',
        default_value='false',
        description='Use simulation/Gazebo clock')

    declare_slam_params_file_cmd = DeclareLaunchArgument(
        'slam_params_file',
        default_value=os.path.join(get_package_share_directory("turtlebot3_NavSLAM"),
                                   'config', 'SLAM_param.yaml'),
        description='Full path to the ROS2 parameters file to use for the slam_toolbox node')

    declare_rviz_config_file_cmd = DeclareLaunchArgument(
        'rviz_config',
        default_value=os.path.join(get_package_share_directory("turtlebot3_NavSLAM"), 'rviz', 'SLAM_rviz.rviz'),
        description='Full path to the RVIZ config file to use')

    start_rviz_cmd = Node(
        package='rviz2',
        executable='rviz2',
        arguments=['-d', rviz_config_file],
        output='screen')

    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        parameters=[{
            'robot_description': ParameterValue(
                Command(['xacro ', str(path_to_urdf)]), value_type=str
            )
        }]
    )

    start_async_slam_toolbox_node = Node(
        package='slam_toolbox',
        executable='async_slam_toolbox_node',
        name='slam_toolbox',
        #output='screen',
        parameters=[
          slam_params_file,
          {'use_sim_time': use_sim_time}
        ],
    )
```

Exercise 1.a - SLAM

- Modify the visualizer launch file
  - Check in launch file if the config param is correct
  - For this exercise it should be "SLAM.rviz"
- Build and source environment
  - *colcon build && . install/setup.bash*
- Launch the visualizer launch file
  - *ros2 launch turtlebot3_visualizer turtlebot3_visualizer.launch.py*
- Launch the SLAM launch file
  - *ros2 launch turtlebot3_NavSLAM turtlebot3_SLAM.launch.py*

```python
def generate_launch_description():
    package_name = 'turtlebot3_visualizer'

    package_dir = get_package_share_directory(package_name)

    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        output='screen',
        arguments=['-d', os.path.join(package_dir, 'rviz', 'SLAM.rviz')],
```

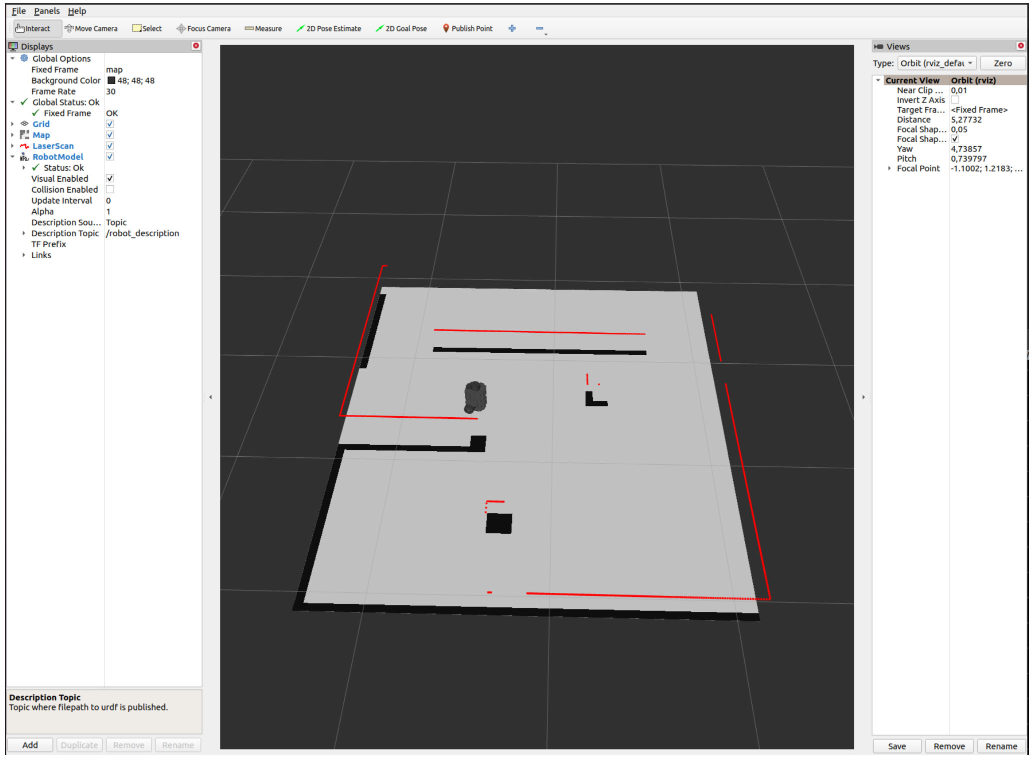
Config file

Exercise 1.a - SLAM

- Launch the teleoperation node to move the robot
  - *ros2 run turtlebot3_teleop teleop_keyboard*
- Move the robot around the map in order to map the whole environment
- In a terminal move to "*src/turtlebot3_NavSLAM/map*"
- Run map saver to save the map
  - *ros2 run nav2_map_server map_saver_cli -f <map_name> --ros-args -p save_map_timeout:=10000*

Exercise 1.b - AMCL

- run the AMCL algorithm in order to load a created map and localize the robot in the environment
- The result will be something like the image in RVIZ2 you will see the robot in the map localized within the environment



Exercise 1.b - AMCL

- In "turtlebot3_AMCL.launch.py"
  - Parameters declaration:
    - map: map path
    - configured_params: config file for map server and amcl
  - Nodes
    - nav2_map_server: map and map info publisher
    - nav2_amcl: localization algoritm

```python
declare_map_yaml_cmd = DeclareLaunchArgument(
    'map',
    default_value=os.path.join(tb3_dir, 'map', '<map_name.yaml>'),
    description='Full path to map yaml file to load')

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

declare_params_file_cmd = DeclareLaunchArgument(
    'params_file',
    default_value=os.path.join(tb3_dir, 'params', 'nav2_params.yaml'),
    description='Full path to the ROS2 parameters file to use for all launched nodes')

declare_autostart_cmd = DeclareLaunchArgument(
    'autostart', default_value='true',
    description='Automatically startup the nav2 stack')

declare_use_composition_cmd = DeclareLaunchArgument(
    'use_composition', default_value='False',
    description='Use composed bringup if True')

declare_container_name_cmd = DeclareLaunchArgument(
    'container_name', default_value='nav2_container',
    description='the name of conatiner that nodes will load in if use composition')

declare_use_respawn_cmd = DeclareLaunchArgument(
    'use_respawn', default_value='False',
    description='Whether to respawn if a node crashes. Applied when composition is disabled.')

load_nodes = GroupAction(
    condition=IfCondition(PythonExpression(['not ', use_composition])),
    actions=[
        Node(
            package='nav2_map_server',
            executable='map_server',
            name='map_server',
            output='screen',
            respawn=use_respawn,
            respawn_delay=2.0,
            parameters=[configured_params],
            remappings=remappings),
        Node(
            package='nav2_amcl',
            executable='amcl',
            name='amcl',
            output='screen',
            respawn=use_respawn,
            respawn_delay=2.0,
            parameters=[configured_params],
            remappings=remappings),
        Node(
```

Exercise 1.b - AMCL

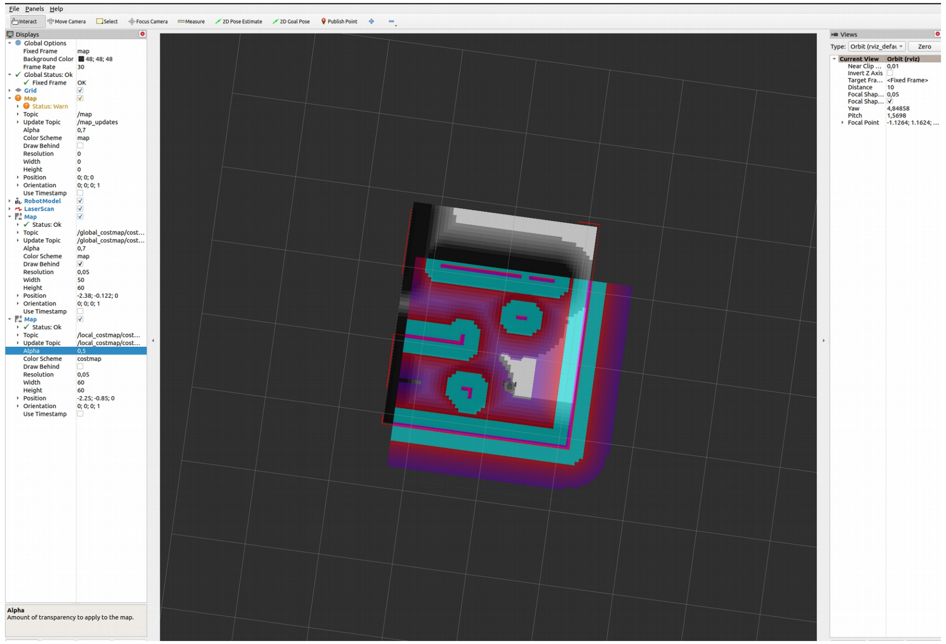
- Modify the AMCL launch file with your .yaml map name in <map_name.yaml> placeholder
- Modify the visualizer launch file
  - Check in launch file if the config param is correct
  - For this exercise it should be "AMCL.rviz"
- Build and source environment
- Launch the visualizer launch file
  - *ros2 launch turtlebot3_visualizer turtlebot3_visualizer.launch.py*

Exercise 1.b - AMCL

- Launch the AMCL launch file
  - *ros2 launch turtlebot3_NavSLAM turtlebot3_AMCL.launch.py*
- Launch the teleoperation node to move the robot
  - *ros2 run turtlebot3_teleop teleop_keyboard*
- When moving the robot, in RVIZ2 the robot should localte itself (not required the initial pose in simulator, ignore the command line messages)
- The position in the RVIZ2 map and Unity environment should be the same

Exercise 1.c - NAV2

- run the NAV2 stack in order to move the robot in autonomy until it reaches the defined goal
- The result will be the robot that reaches the goal position avoiding the obstacles.



Exercise 1.c - NAV2

- In "turtlebot3_NAV2.launch.py"
  - Parameters declaration
    - map_yaml_file: map path
    - nav2_param_file: nav2 config file
  - Nodes
    - Map_server: load map
    - Localization launch file: AMCL
    - Behavior tree: for recovery
    - Nav2: controller and planner server

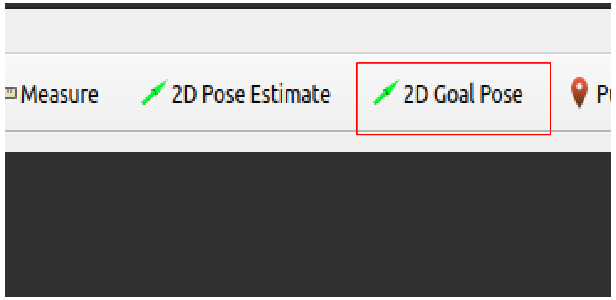
```python
declare_map_yaml_file = DeclareLaunchArgument(
    'map_yaml_file',
    default_value=os.path.join(package_dir,'maps', 'ice_map.yaml'),
    description='Full path to map file to load')

declare_nav2_params_file_cmd = DeclareLaunchArgument(
    'nav2_params_file',
    default_value=os.path.join(package_dir,'config', 'nav2_param.yaml'),
    description='Full path to the ROS2 parameters file to use for the navigation node')

declare_bt_params_file = DeclareLaunchArgument(
    'bt_params_file',
    default_value=os.path.join(package_dir,'config', 'bt_config.xml'),
    description='Full path to the ROS2 parameters file to use for the navigation node')

declare_autostart_cmd = DeclareLaunchArgument(
    'autostart', default_value='true',
    description='Automatically startup the nav2 stack')

with open(urdf, 'r') as infp:
    robot_desc = infp.read()

robot_description = [{'robot_description': robot_desc}]

rviz_node = Node(
    package='rviz2',
    executable='rviz2',
    output='screen',
    arguments=['-d', os.path.join(package_dir, 'rviz', 'nav2_rviz.rviz')],
)

robot_state_publisher_node = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher',
    output='screen',
    parameters=[{'robot_description': robot_desc, 'use_sim_time': use_sim_time}],
)

localization_launch = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(os.path.join(package_dir, 'launch', 'localization_launch.py')),
    launch_arguments={'namespace': namespace,
                      'map': map_yaml_file,
                      'use_sim_time': use_sim_time,
                      'autostart': autostart,
                      'params_file': nav2_params_file}.items()
)

navigation_launch = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        os.path.join(get_package_share_directory('nav2_bringup'), 'launch', 'navigation_launch.py')
    ),
    launch_arguments={
        'namespace': namespace,
        'use_sim_time': 'false',
        'autostart': autostart,
        'params_file': nav2_params_file,
        'use_lifecycle_mgr': 'false',
        'map_subscribe_transient_local': 'true',
        'default_bt_xml_filename': bt_params_file,
    }.items()
)
```

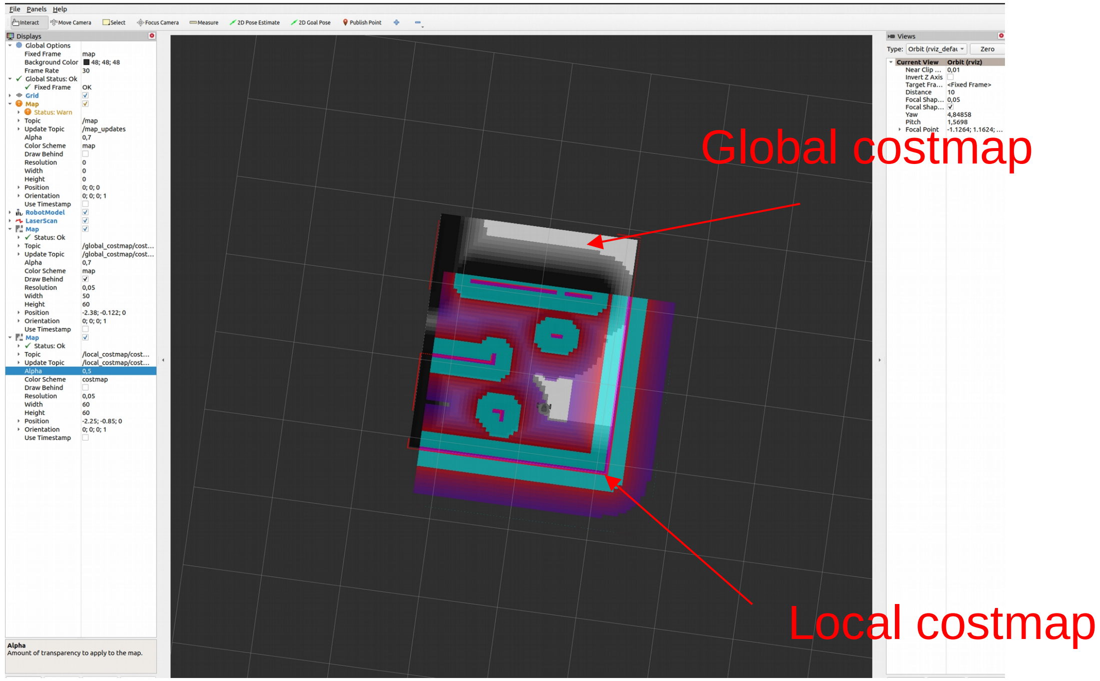
Exercise 1.c - NAV2

- Modify the NAV2 launch file with your .yaml map name in <map_name.yaml> placeholder
- Modify the visualizer launch file
  - Check in launch file if the config param is correct
  - For this exercise it should be "NAV2.rviz"
- Build and source environment
- Launch the visualizer
  - *ros2 launch turtlebot3_visualizer turtlebot3_visualizer.launch.py*

Exercise 1.c – NAV2

- Launch NAV2
  - *ros2 launch turtlebot3_NavSLAM turtlebot3_NAV2.launch.py*
- Map and robot will appear in RVZ2
- Set the goal from RVIZ2



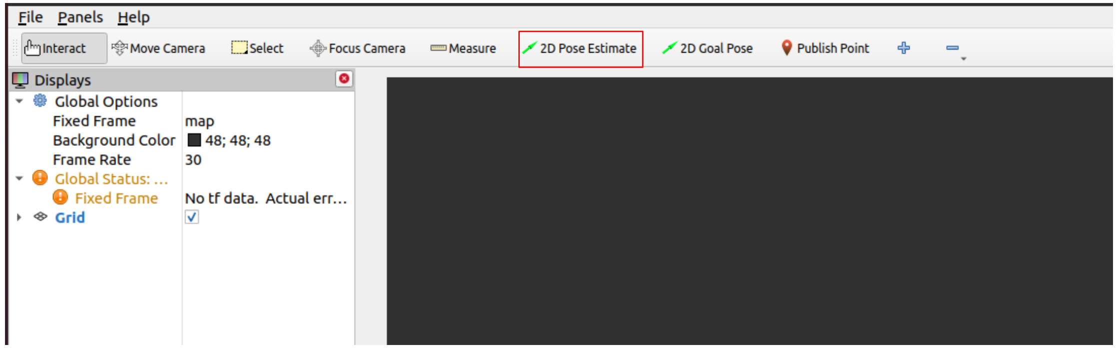


Exercise 2 - Turtlebot3

- Set up the arena for turtlebot3
- Turn on the robot
- Set your ROS_ID same to the correct robot
- Connect via ssh to the robot and run bringup
  - *ros2 launch turtlebot3_bringup robot.launch.py*
- Check if you see the turtlebot3 topics in your pc
  - *ros2 topic list*

Turtelbot3 – SLAM – AMCL – NAV2

- Run the same simulation tasks on the real robot
  - Map the arena and save it
  - Localize the robot in the map (initial pose is required)
    - When running the node in RVIZ2 you have to set the initial pose of the robot in the map
  - Send goal to NAV2 stack to move the robot



Turtelbot3 – SLAM – AMCL – NAV2

- All launch files (SLAM, AMCL, NAV2) must be launched on your PC
- Only the bringup file must run on the turtlebot3
- The map must be saved in the same folder as the simulation exercise (when you save the map choose a different name compared to the simulation exercise)
  - */src/turtlebot3_NavSLAM/map*

Troubleshoot

- If the RVIZ2 configuration does not load
  - Click on File –> Open config
  - Browse to map folder in "turtlebot3_NavSLAM" package
  - Select config file according to your exercise type

References

- NAV2 parameters
  - https://navigation.ros.org/concepts/index.html#global-positioning-localization-and-slam
  - https://navigation.ros.org/getting_started/index.html
  - https://navigation.ros.org/configuration/index.html
- AMCL parameters
  - https://navigation.ros.org/configuration/packages/configuring-amcl.html
- SLAM parameters
  - https://navigation.ros.org/tutorials/docs/navigation2_with_slam.html
  - https://github.com/SteveMacenski/slam_toolbox