5. gmapping map building algorithm

wiki：http://wiki.ros.org/gmapping/

ros2 gmapping：https://github.com/Project-MANAS/slam_gmapping

The operating environment and hardware and software reference configuration are as follows:

- Reference model: ROSMASTER R2
- Robot hardware configuration: Arm series main control, Silan A1 LiDAR, AstraPro Plus depth camera.
- Robot system: Ubuntu (version not required) + docker (version 20.10.21 and above)
- PC virtual machine: Ubuntu (20.04) + ROS2 (Foxy)
- Usage scenario: use on a relatively clean 2D plane

5.1. Introduction

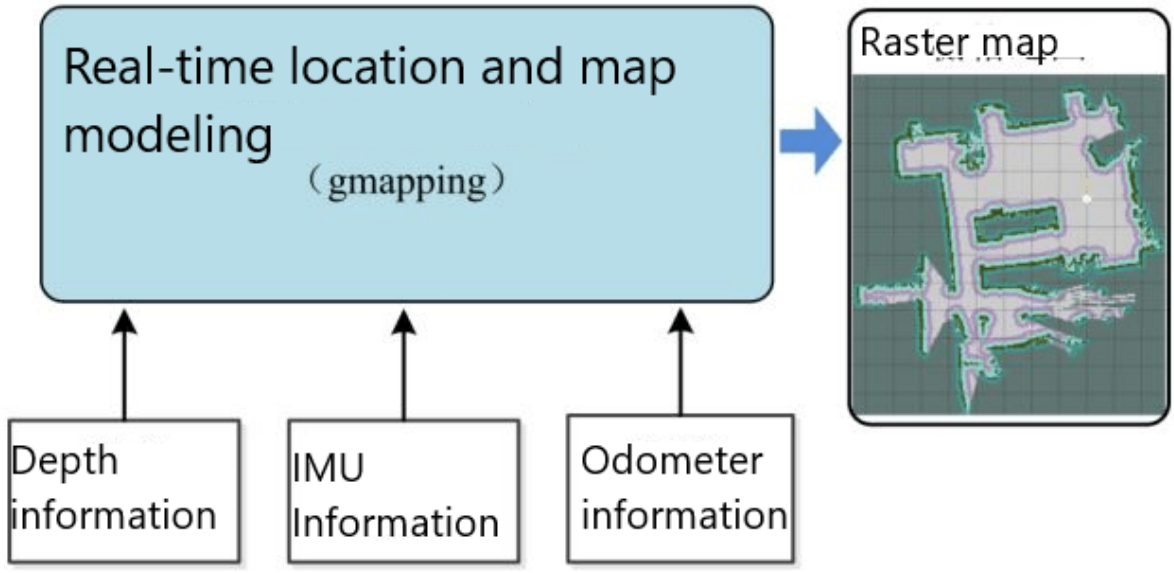
- gmapping only applies to single-frame 2D laser points less than 1440 points, if the single-frame laser points are greater than 1440, then the [[mapping-4] process has died] such a problem.
- Gmapping is a commonly used open source SLAM algorithm based on the filtered SLAM framework.
- Gmapping is based on the RBpf particle filtering algorithm, which instantly separates the localization and mapping processes, first localization and then mapping.
- Gmapping makes two main improvements on the RBpf algorithm: improved proposal distribution and selective resampling.

Advantages: Gmapping can build indoor maps in real time, and requires less computation and higher accuracy in building maps for small scenes.

Disadvantages: The number of particles required increases as the scene grows, and since each particle carries a map, the amount of memory and computation required to build large maps increases.

memory and computation will increase when constructing a large map. Therefore, it is not suitable for building large scene maps. Moreover, there is no loop detection, so the map may be misaligned when the loop is closed.

Although increasing the number of particles can make the map closed, it is at the cost of increasing the amount of computation and memory.



5.2 Usage

5.2.1 Pre-use Configuration

Note: Since ROSMASTER series robots are divided into multiple robots as well as multiple devices, the factory system has been configured with routines for multiple devices, but since it is not possible to automatically recognize the products, you need to manually set the machine type and lidar model.

After entering the container: according to the model of the trolley, the type of lidar and the type of camera make the following changes:

```
root@ubuntu:/# cd
root@ubuntu:~# vim .bashrc
```

```
# env
alias python=python3
export ROS_DOMAIN_ID=112

export ROBOT_TYPE=r2            # r2, x1, x3
export RPLIDAR_TYPE=a1          # a1, s2, 4ROS
export CAMERA_TYPE=astraplus    # astrapro, astraplus
echo "------------------------------------------------------"
echo -e "ROS_DOMAIN_ID: \033[32m$ROS_DOMAIN_ID\033[0m"
echo -e "my_robot_type: \033[32m$ROBOT_TYPE\033[0m | my_lidar: \033[32m$RPLIDAR_TYPE\033[0m | my_camera: \033[32m$CAMERA_TYPE\033[0m"
echo "------------------------------------------------------"

#colcon_cd
source /usr/share/colcon_cd/function/colcon_cd.sh
export _colcon_cd_root=/root/yahboomcar_ros2_ws/yahboomcar_ws
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash

#ros2
source /opt/ros/foxy/setup.bash
source /root/yahboomcar_ros2_ws/yahboomcar_ws/install/setup.bash
source /root/yahboomcar_ros2_ws/software/library_ws/install/setup.bash
```

When the changes are complete, save and exit vim, then execute:

```
root@ubuntu:~# source .bashrc
------------------------------------------------------------
ROS_DOMAIN_ID: 12
my_robot_type: r2 | my_lidar: a1 | my_camera: astraplus
------------------------------------------------------------
root@ubuntu:~#
```

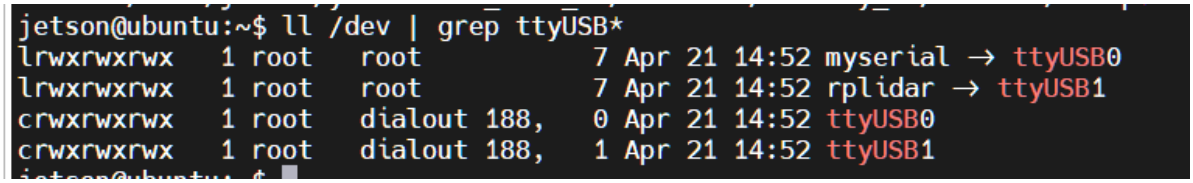
You can see the current model of the modified cart, the type of lidar and the type of camera.

5.2.2 Specific use

Note: When building a map, the slower the better the result (note that if the rotation speed should be slower), the speed is too fast, the effect will be very bad.

First of all, in the host [car] need to do port binding operation [see the port binding tutorial chapter], here mainly use the lidar and serial port two devices;

Then check whether the lidar and serial port devices are in the port binding state: in the host [cart] refer to the following command to view the successful binding is the following state:



If it shows that the lidar or serial device is not bound, you can plug and unplug the USB cable to check again.

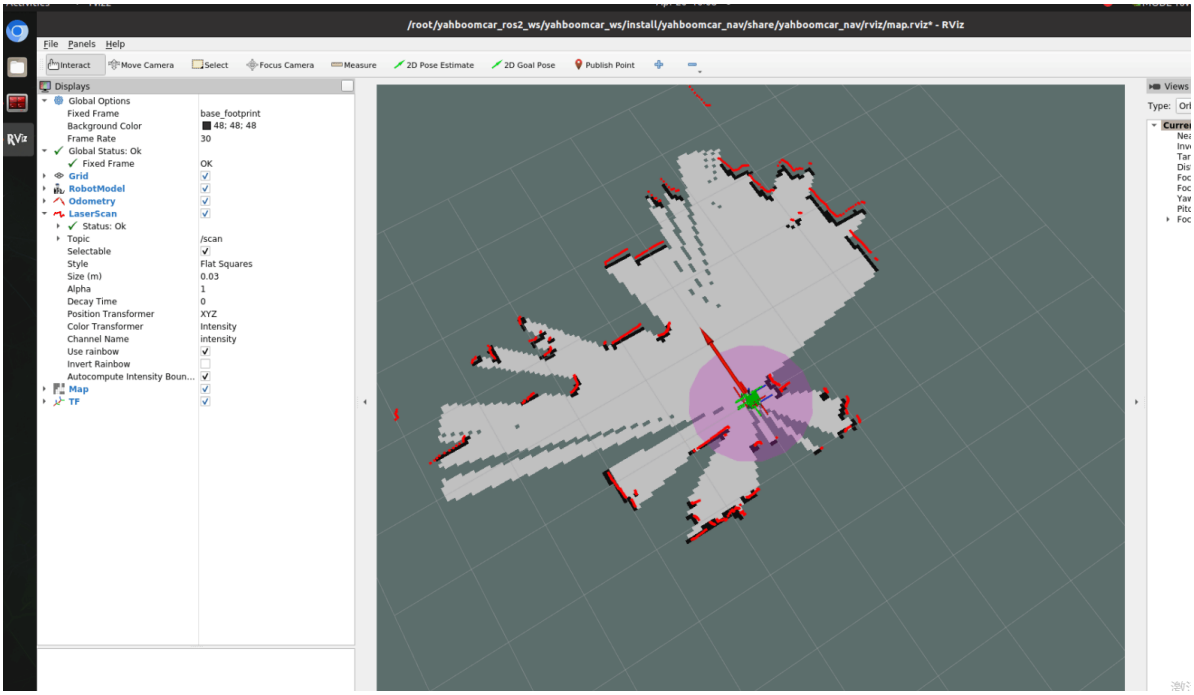
Enter the docker container (for steps, see [docker course chapter ----- 5. Enter the docker container of the robot]), and execute the following LAUNCH file in a sub-terminal:

1, start build map

```
ros2 launch yahboomcar_nav map_gmapping_launch.py
```

2, start rviz to display the map, this step is recommended to perform in the virtual machine, the virtual machine needs to be configured to multi-machine communication

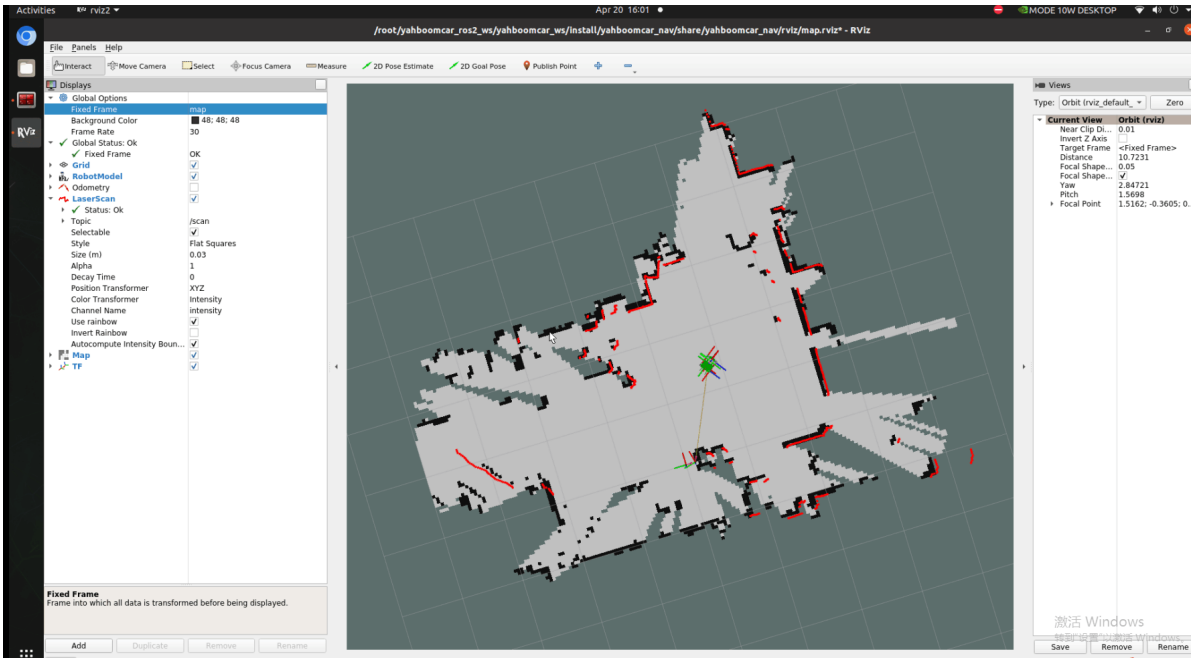
```
ros2 launch yahboomcar_nav display_map_launch.py
```



3, start the keyboard control node, this step is recommended to perform in the virtual machine, the virtual machine needs to be configured for multi-machine communication

Or use the remote control [slowly moving cart] to start building the map, until a complete map is built!

```
ros2 run yahboomcar_ctrl yahboom_keyboard
```

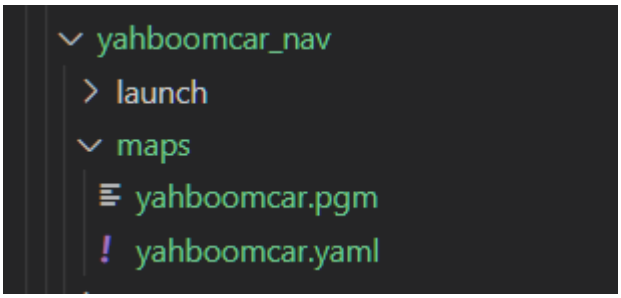


4. The map is saved with the following path:

```
ros2 launch yahboomcar_nav save_map_launch.py
```

The save path is as follows:

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/
```



One pgm image, one yaml file yahboomcar.yaml

```
image:
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/yahboomcar.pgm
mode: trinary
resolution: 0.05
origin: [-10, -10, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

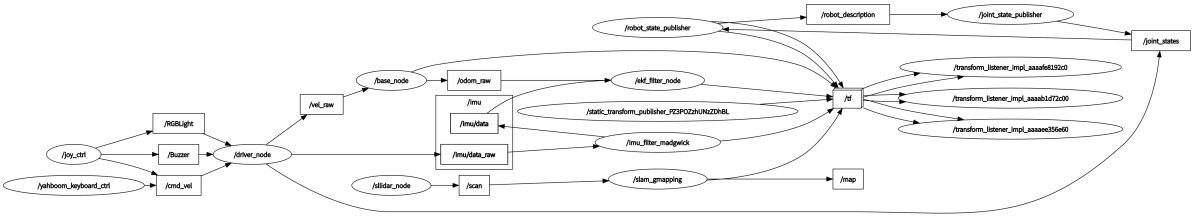
Parameter parsing:

- image: path of the map file, can be absolute or relative.
- mode: the attribute can be one of trinary, scale or raw, depends on the selected mode, trinary mode is the default.
- resolution: the resolution of the map, in meters/pixels
- origin: the 2D position (x,y,yaw) of the bottom left corner of the map, where yaw is rotated counterclockwise (yaw=0 means no rotation). Currently many parts of the system ignore the yaw value.
- negate: whether to reverse the meaning of white/black, free/occupied (the interpretation of the thresholds is not affected)
- occupied_thresh: pixels with occupancy probability greater than this threshold are considered fully occupied.
- free_thresh: pixels with occupancy probability less than this threshold are considered completely free.

5.3 Node resolution

5.3.1. Displaying the computation graphs

```
rqt_graph
```



5.3.2. gmapping node details

```
rr@rr-pc:~$ ros2 node info /slam_gmapping
/slam_gmapping
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /scan: sensor_msgs/msg/LaserScan
  Publishers:
    /entropy: std_msgs/msg/Float64
    /map: nav_msgs/msg/OccupancyGrid
    /map_metadata: nav_msgs/msg/MapMetaData
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /tf: tf2_msgs/msg/TFMessage
  Service Servers:
    /slam_gmapping/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /slam_gmapping/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /slam_gmapping/get_parameters: rcl_interfaces/srv/GetParameters
    /slam_gmapping/list_parameters: rcl_interfaces/srv/ListParameters
    /slam_gmapping/set_parameters: rcl_interfaces/srv/SetParameters
    /slam_gmapping/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
```

5.3.3 TF transformations

```
ros2 run tf2_tools view_frames.py
```