Appendix (Detailed Report)

This appendix is intended to detail the steps taken to complete world creation and map generation. The following will provide a detailed description of each part of the world specification, startup files, maps, test scenarios, and how to achieve the generation of accurate maps, and include reflective sections on what worked, the problems encountered, and the lessons learned.

Contents

1. World design and human creation

Gazebo was used to build an indoor environment with multiple obstacles and paths to ensure sufficient space for robot navigation, saved as wrold_rbot.wrold. The files are in the world folder under the flat_exploration_robot package. Figure 1 shows the interior environment of the design, which is enclosed by four walls with two Windows, and two interior walls as partitions, each with a door for small robots to pass through. A coffee table, a chair and a bookshelf are also inserted into the environment, along with a sphere and cube as obstructions.

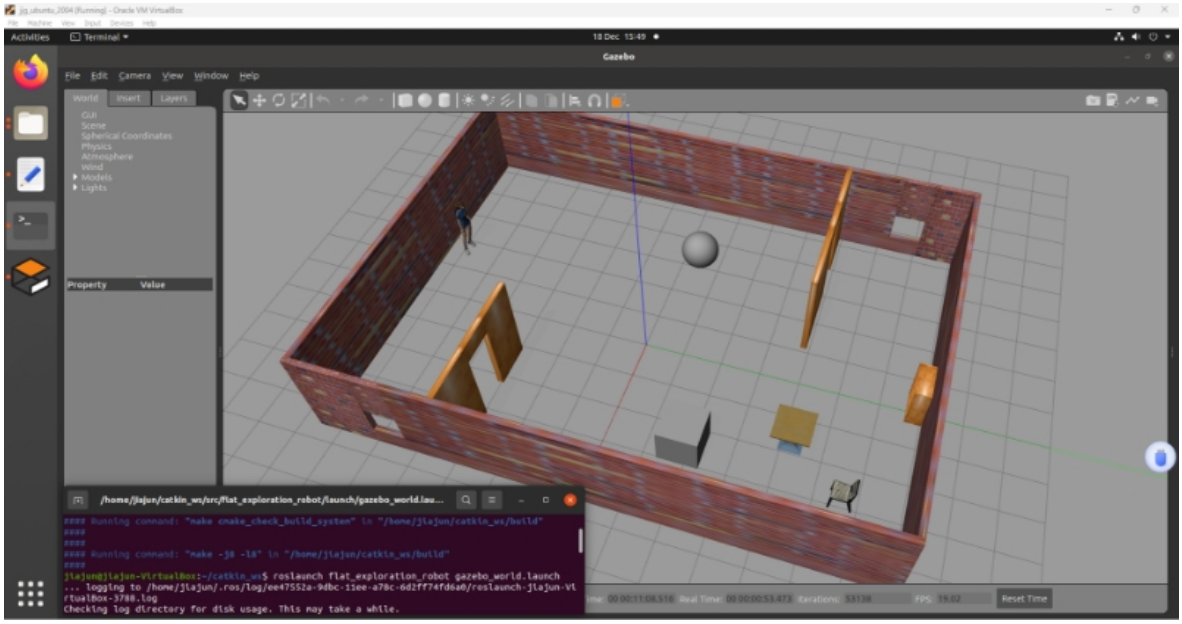

Figure 1

Regarding the creation of the human model, Figure 2 shows the human model I created in MakeHuman. It is important to ensure that Low-poly eyes are selected and that the scale units are selected as meters when exporting. Save and name her “gina” after creation, making sure to keep the names consistent in other human-related files such as config and sdf files. After testing it in Blender to see if it works, export a new dae file named “gina2” and change the “gina” in the previous sdf file to “gina2”, shown in Figure 3 and 4. Place the “gina” folder in "/ home/jiajun /.gazebo/models" path can be easily found in the Insert Gzaebo her.

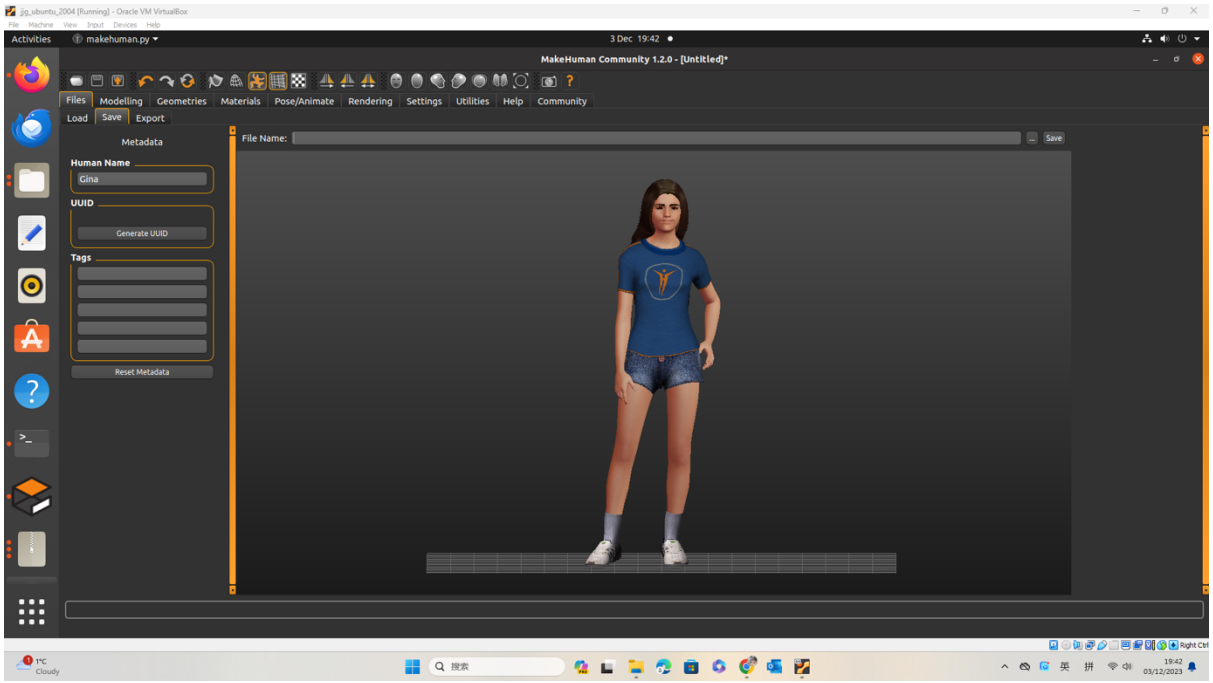

Figure 2

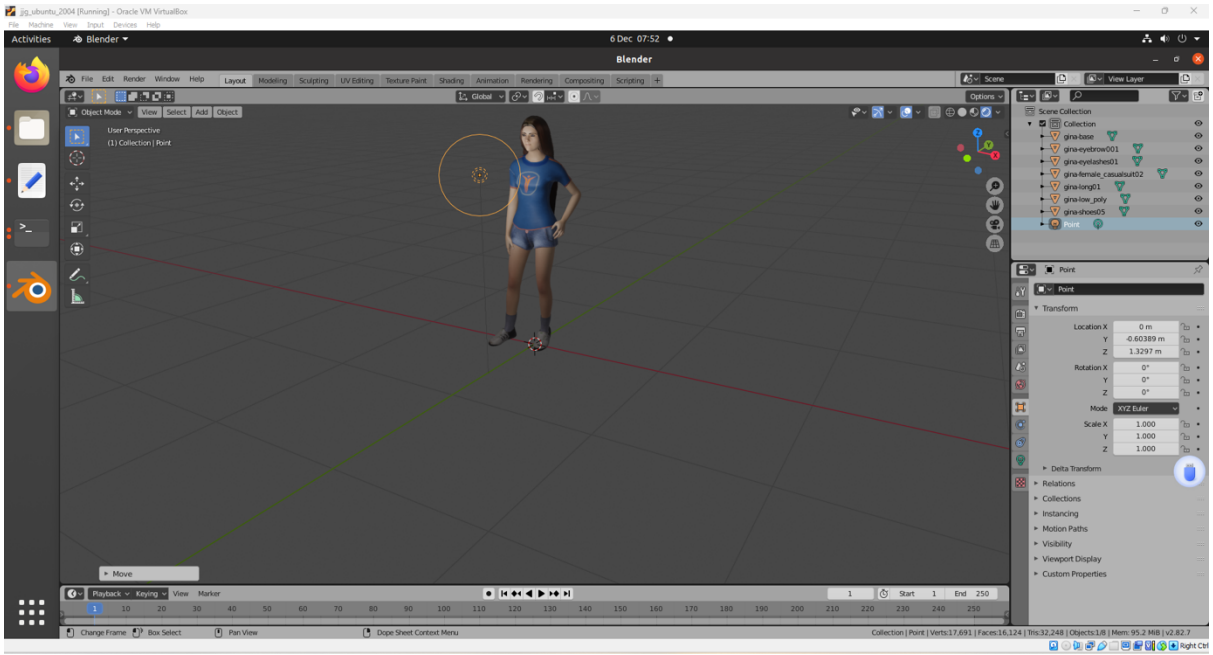

Figure 3

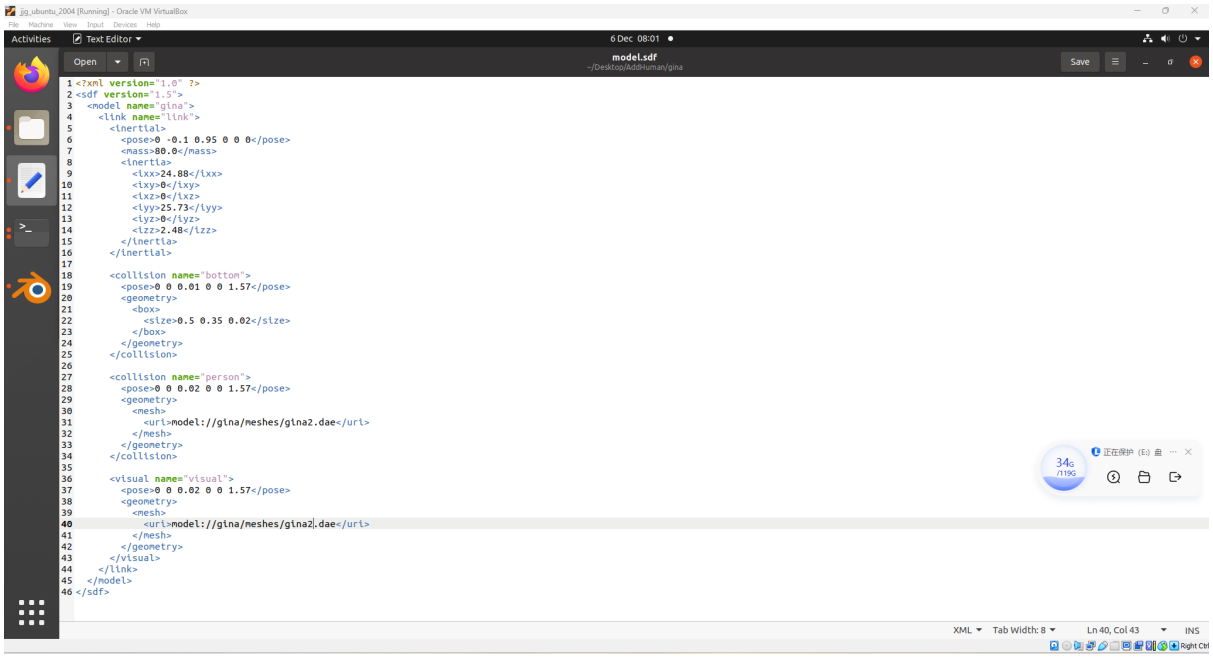

Figure 4

1.2 Launch file:

The gazebo_world.launch file shown in Figure 5 can be launched by running “roslaunch flat_exploration_robot gazebo_world.launch” on the terminal to load the indoor environment in Gazebo.

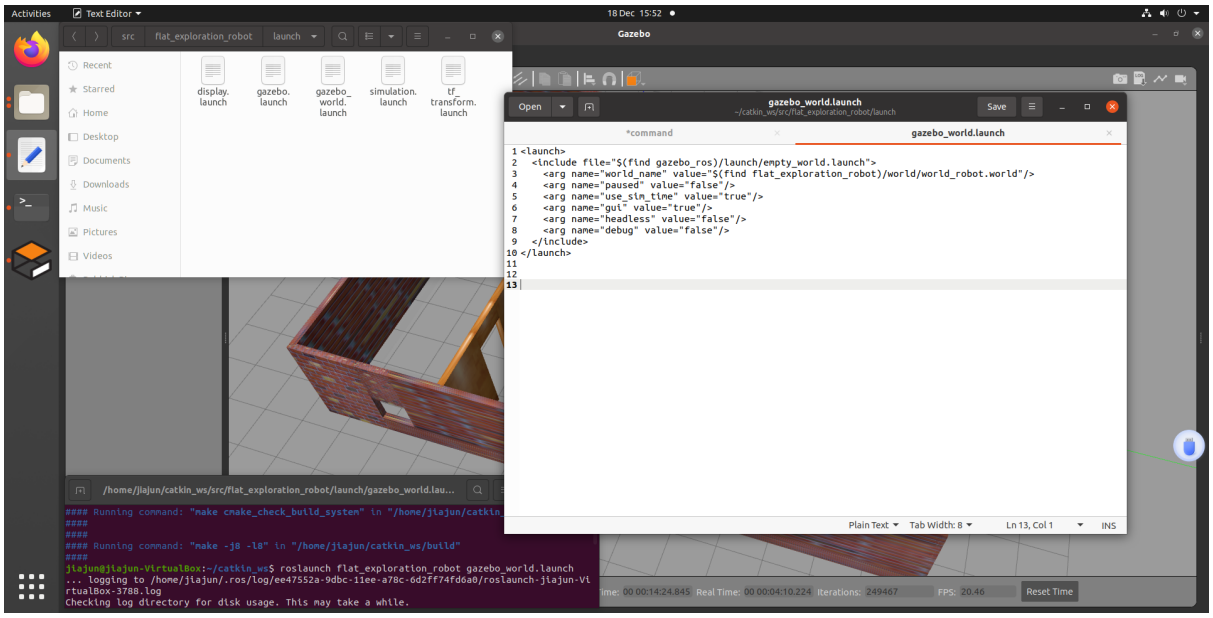

Figure 5

2. Sensor selection and robot transformation

Figure 5 shows the model of laser added in the flat_exploration_robot.xacro file, You can see that this code defines a fixed Lidar sensor that is mounted 10 cm above the base of the robot. The sensor does not move relative to the base. Figure 7 and 8 are part of the flat_exploration_robot.gazebo.xacro file, which defines the appearance, physical properties, and sensor configuration of the robot in the Gazebo simulation environment. It makes the robot model dynamic and interactive when running in the simulated world. The <plugin> element with the name "differential_drive_controller" and filename "libgazebo_ros_diff_drive.so" is used to control the robot's differential drive, which allows it to move and turn by varying the speeds of its left and right wheels. It contains parameters for the robot's motion, such as wheel separation, wheel diameter, and acceleration limits. <gazebo reference="base_laser_link"> configures a laser sensor attached to the robot. It specifies the visualization properties, update rate, scan properties, and noise model for the sensor. It also defines the plugin responsible for integrating the sensor with ROS, allowing it to publish scan data to a ROS topic. <gazebo reference="Camera_Link"> defines a camera sensor with its visualization settings, field of view, image size, and format. The nested <plugin> element with the name "camera_controller" and filename "libgazebo_ros_camera.so" allows the camera to interface with ROS, defining topics for image data and camera information. Figure 9 shows the tf tree, which clearly examines all the transformations and how they are connected.

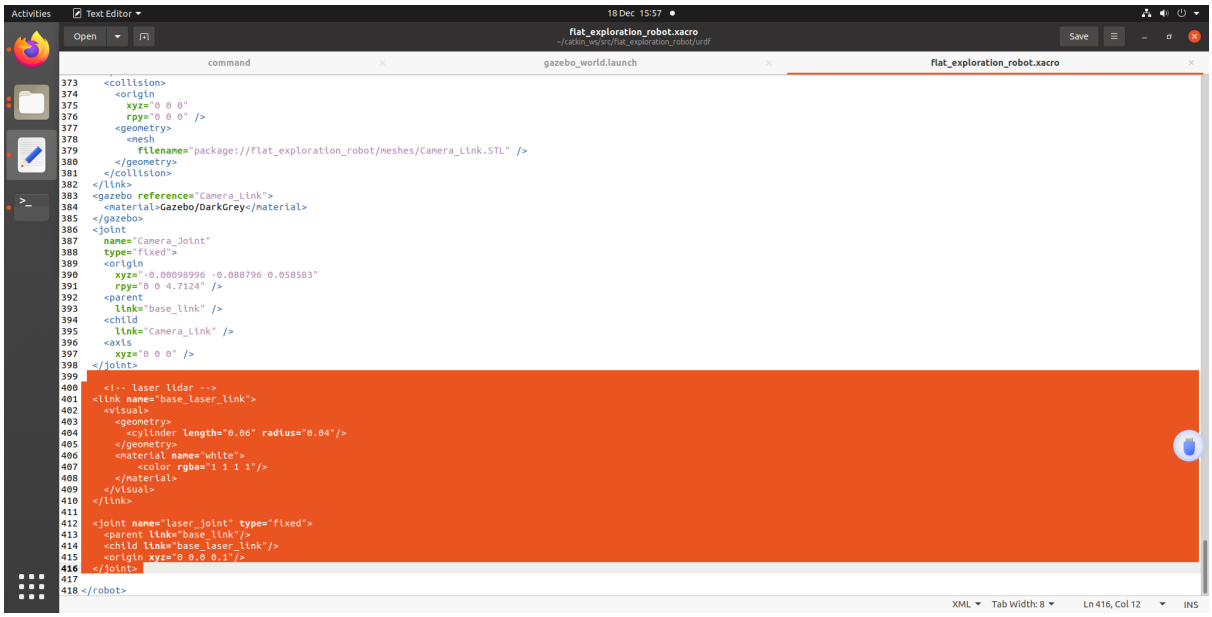

Figure 6

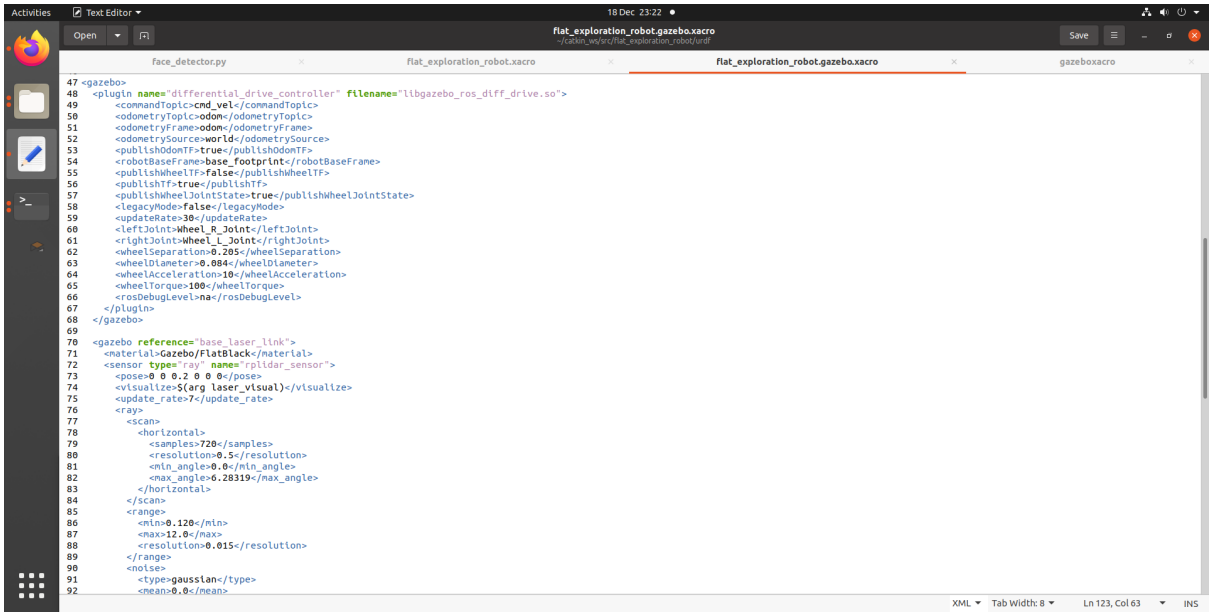

Figure 7

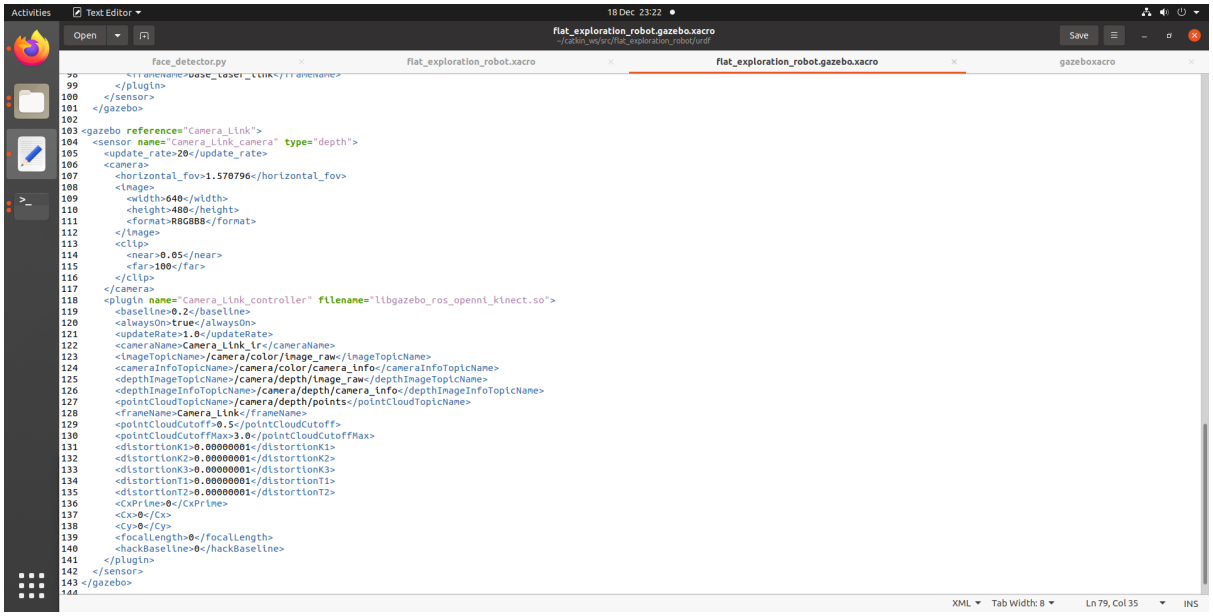

Figure 8

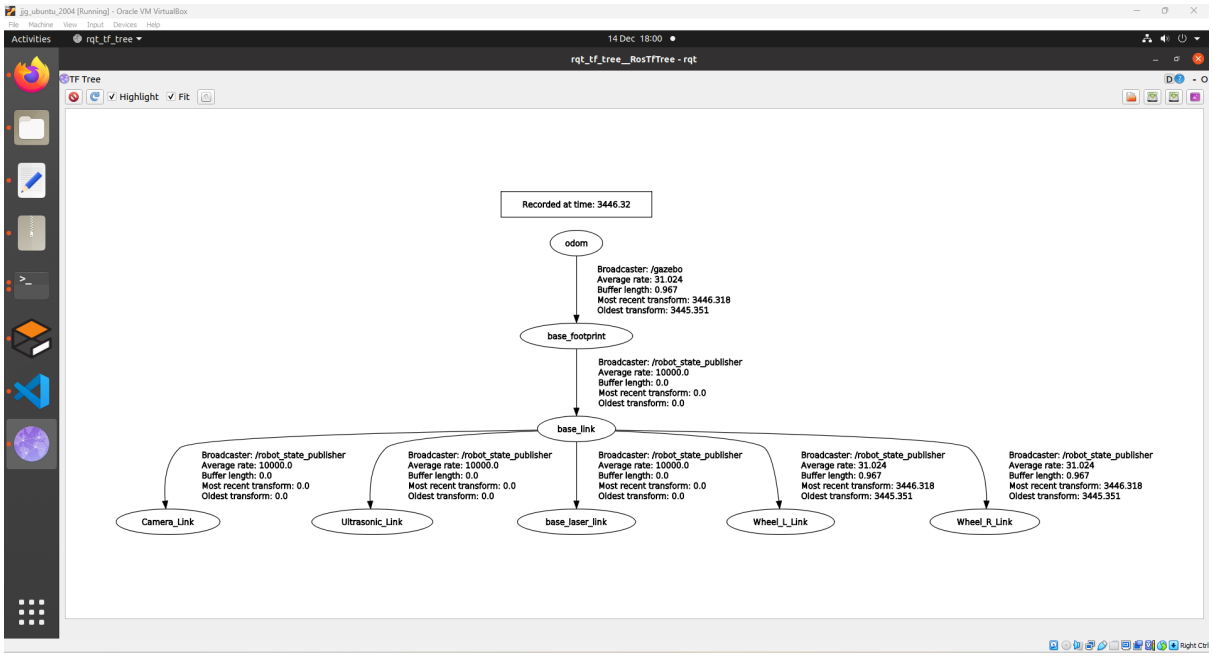

Figure 9

2.2 Launch file:

By running “roslaunch flat_exploration_robot gazebo_world.launch” on the terminal, the robot model can be loaded in Gazebo. Figure 10 shows simulation.launch file. This launch file is used to set up the simulation environment gazebo_world.world with flat_exploration_robot model. It places the robot at a defined position, loads the robot's description from an xacro file, and ensures that the robot's state is published for other nodes to use. Run “roslaunch flat_exploration_robot simulation.launch” on the terminal, as shown in Figure 11, you can load both the world and the robot in Gazebo, and the robot will be in the center of the world.

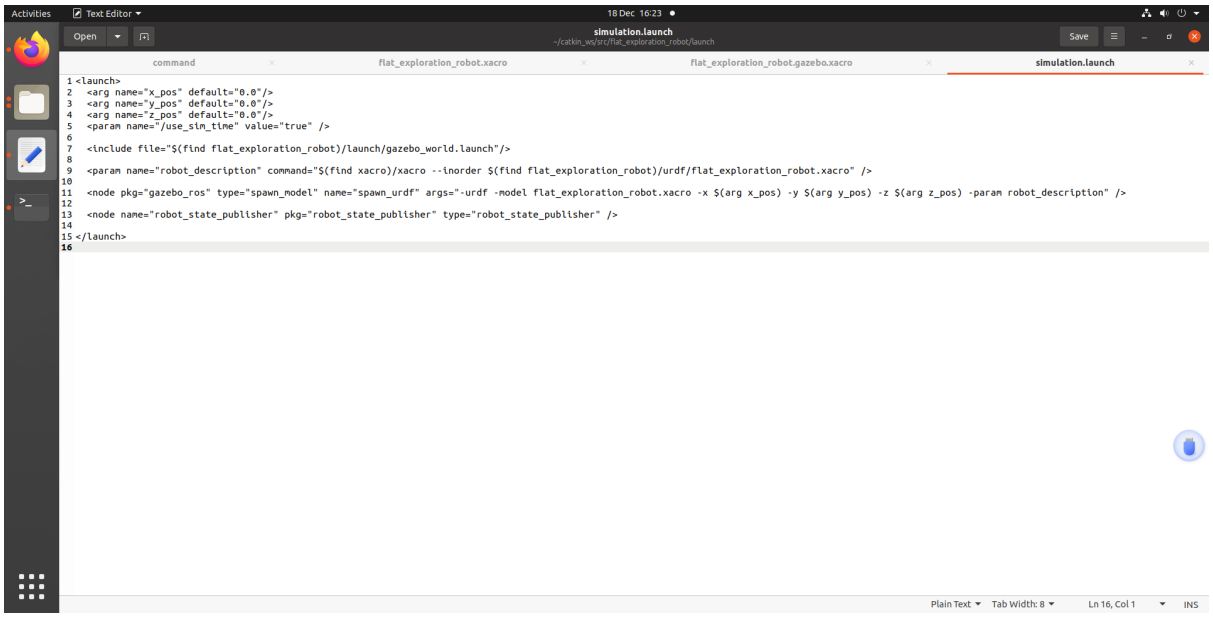

Figure 10

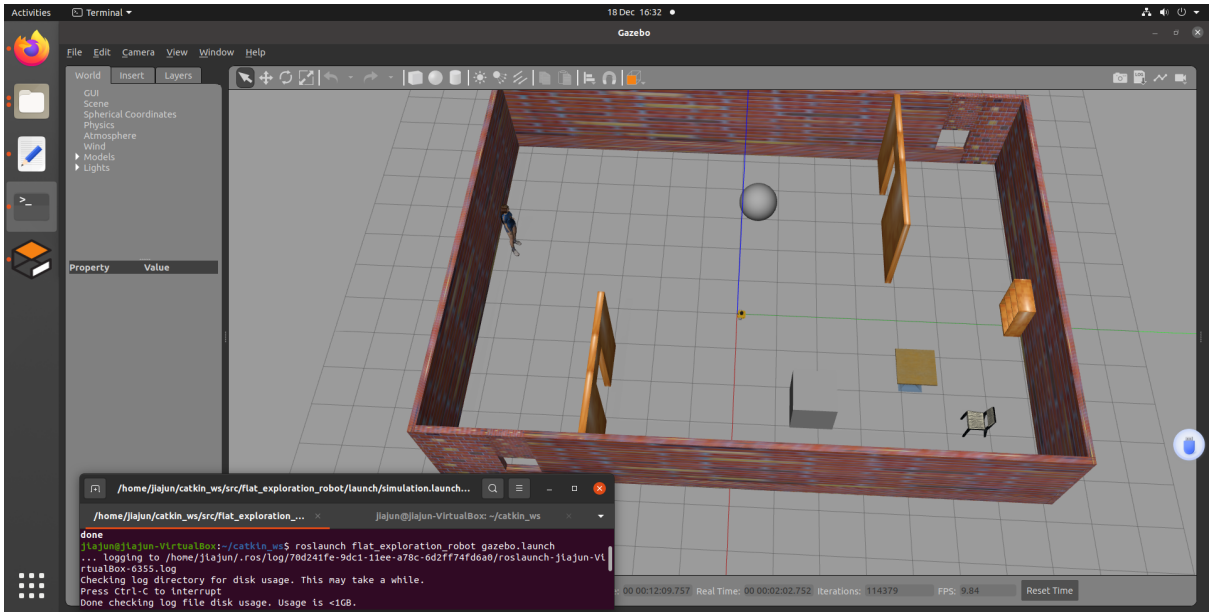

Figure 11

3. Autonomous navigation

First create a new package named "flat_exploration_robot_nav", then configure the required files and folders. Start SLAM using gmapping by running the command "roslaunch flat_exploration_robot_nav gmapping.launch simulation:=true", since the simulation environment is used, simulation:=true is specified to ensure a consistent timeline. Then, input "roslaunch flat_exploration_robot_nav slam_rviz.launch" in the terminal to launch the interface of Rviz. As can be seen from Figure 12, the starting position of the robot is in the

center of the map, and the radar carries out a 360-degree scan. Open a new terminal window and type "rosrun teleop_twist_keyboard teleop_twist_keyboard.py" Use keyboard control car to scan the whole world, after that use" /catkin_ws/src/flat_exploration_robot_nav/maps" and "rosrun map_server map_saver -f map saves the map in the "maps" folder in the "flat_exploration_robot_nav" package, as shown in Figure 13.

And then the navigation stack test, launch the file, " navigation_stack.launch", with the command "roslaunch flat_exploration_robot_nav navigation_stack.launch simulation:=true". At the same time, "roslaunch flat_exploration_robot_nav navigation_rviz.launch" can be launched to more clearly view the status and position of the robot in the map. If there is some deviation in the position of the robot, 2D Pose Estimate can be used to adjust the initial posture of the car to match the map. After giving the robot a target, it can see its planned path, and it can quickly adjust and plan a new path when it encounters obstacles. Figure 14 shows that after specifying a target in Rviz, the robot plans a green path.

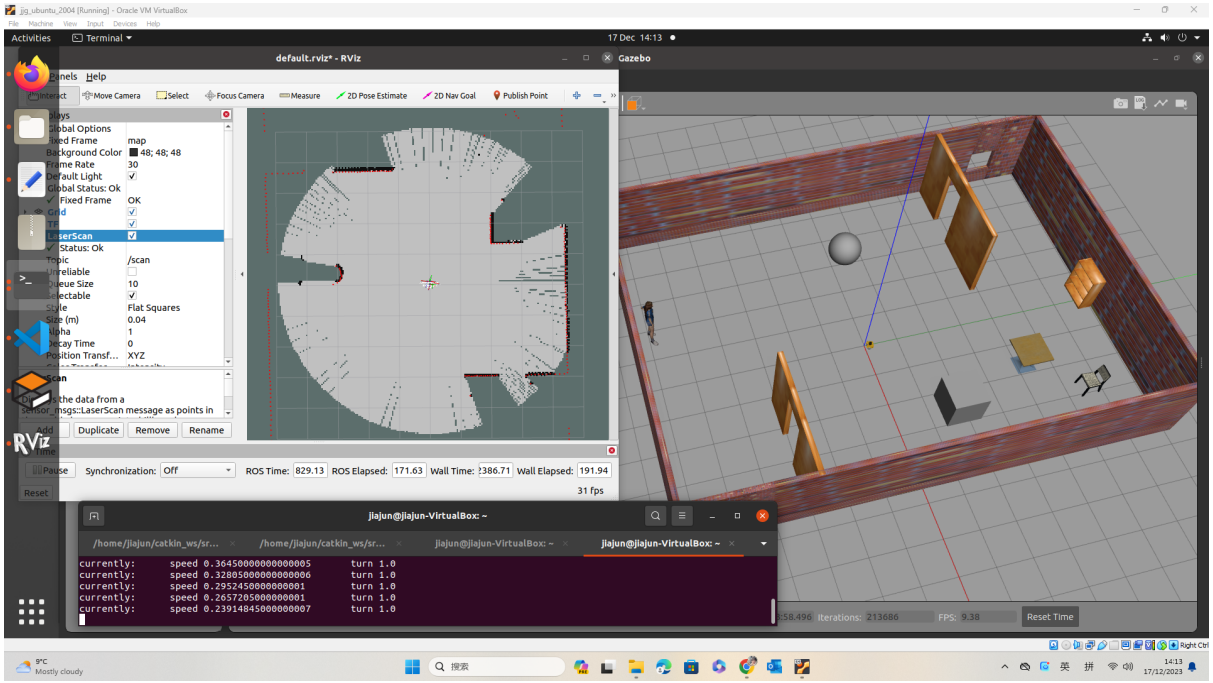

Figure 12

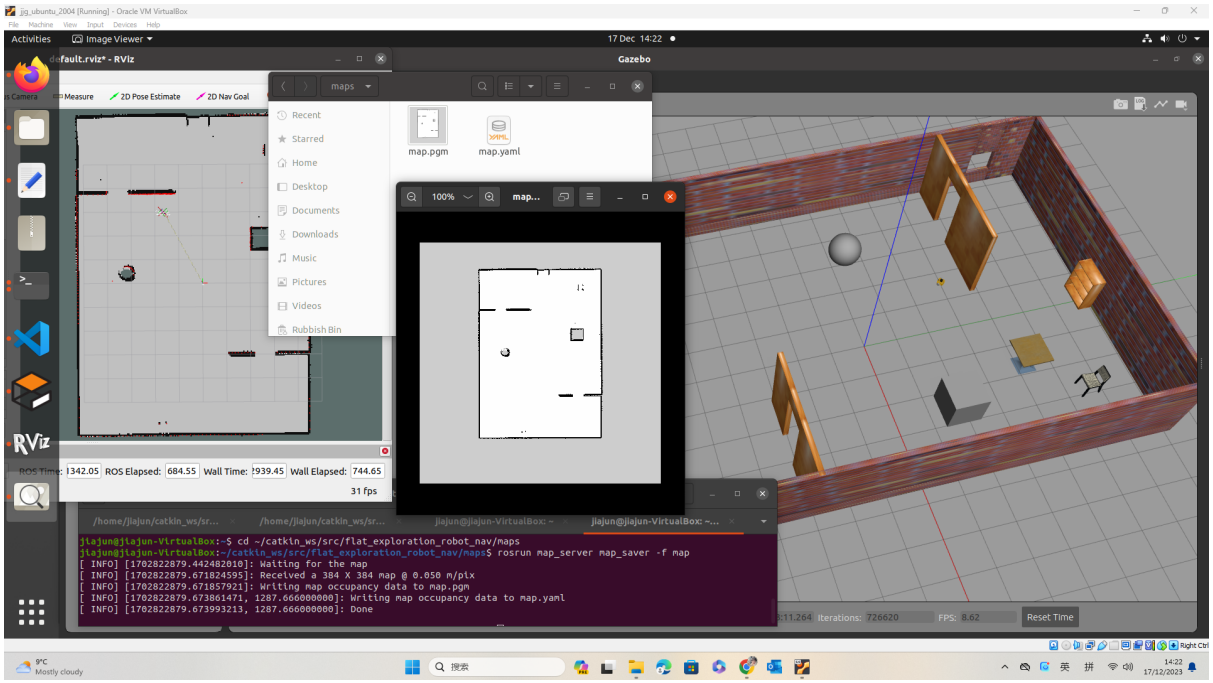

Figure 13

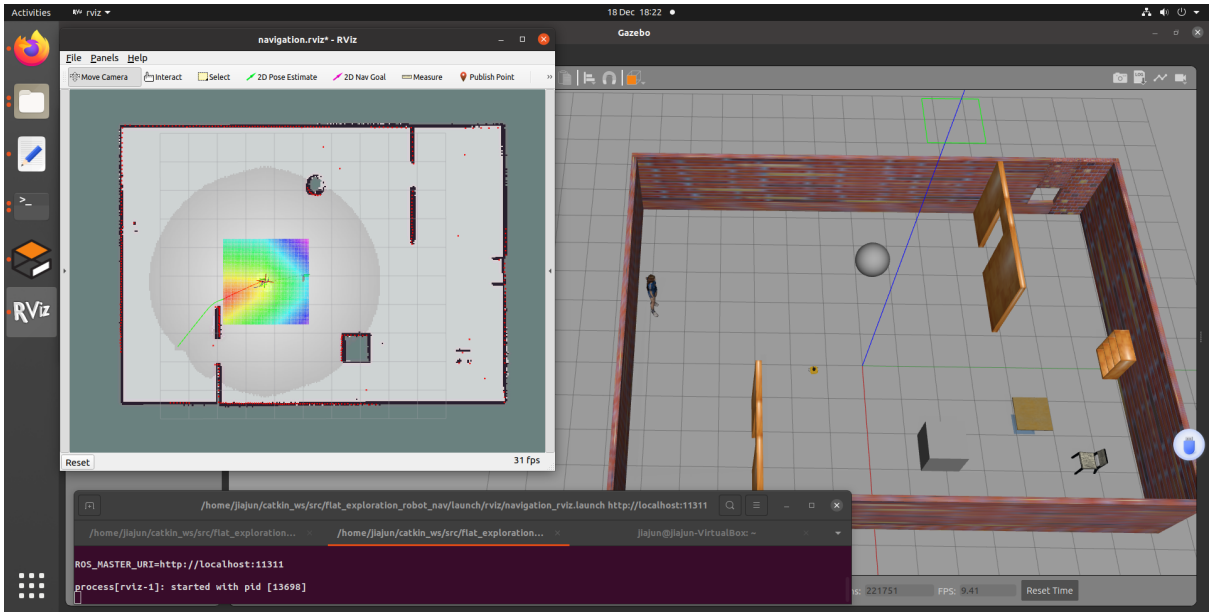

Figure 14

3.2 Launch file:

The "navigation_stack.launch" file in Figure 15 configures and runs the map service, localization, and path planning nodes and parameters required for autonomous navigation of the robot. It allows for the customization of various aspects of navigation by using parameters, which can be tailored to the specific needs of the robot or its operating environment. The "map_file" refers to the path of the map file used by the navigation stack, with the "map.yaml"

saved in the "maps" directory of the "flat_exploration_robot_nav" package. The "planner" specifies the path planning algorithm to use, such as the Dynamic Window Approach (DWA) or the Timed Elastic Band (TEB). The "map_server" node is launched to provide the map data to the rest of the navigation stack. This node reads the map from the "map_file" defined in the argument. The Adaptive Monte Carlo Localization (AMCL) node is started to estimate the robot's position based on laser scans and the provided map. It loads its configuration parameters from the YAML file specified by the environment variable BASE_TYPE, which should correspond to the type of base used by the robot. The parameters initial_pose_x, initial_pose_y, and initial_pose_a are set to 0.0, which positions the robot at the map's origin without any initial rotation. The "move_base" includes the "move_base" node, launched from the launch file in the "flat_exploration_robot_nav" package, which is responsible for moving the robot to the target location. It passes the previously defined arguments to the move_base configuration. These arguments affect how move_base plans paths and how it interacts with the rest of the navigation stack. It is worth noting that the environment variable "BASE_TYPE" is set to "NanoCar" by default. The related files for "NanoCar" are saved in the "param" directory of the "flat_exploration_robot_nav" package. You can set the "BASE_TYPE" to "NanoCar" by default locally using the command "export BASE_TYPE=NanoCar" and "source ~/.bashrc".

Figure 16 is the "move_base.launch" launch file, which is used to configure the move_base node for robot navigation using the dynamic window method (DWA) or Timing Elastic Band (TEB) planner. Setting parameters for the global and local cost graphs and specific planner parameters allows for flexible configuration of the navigation stack based on the needs of the robot and its navigation environment. Arguments define default values for the command velocity (cmd_vel_topic), odometry (odom_topic), planner type (planner), simulation flag, and path planning algorithm (use_dijkstra). There are two planner selections, one for each planner type. The file loads different parameters based on the planner selected (DWA or TEB). The configuration files for the costmaps are loaded, Global and Local Costmaps, these files define the parameters for obstacle information processing and cost distribution around obstacles. Global Planner Algorithm sets whether to use Dijkstra's or A*'s algorithm for global path planning.

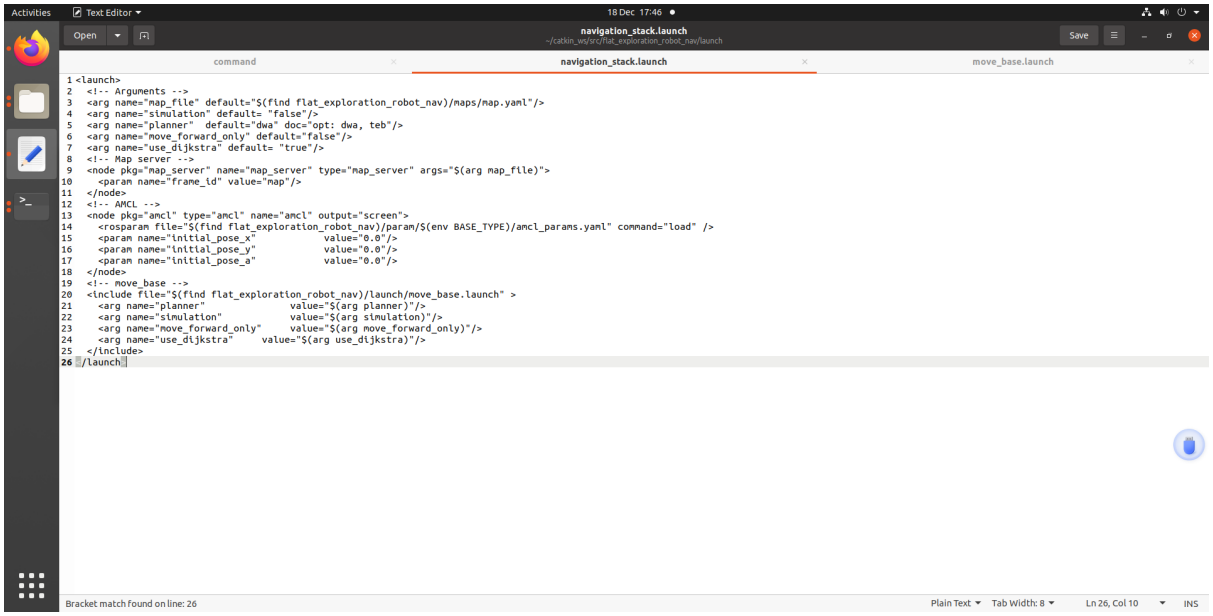

Figure 15

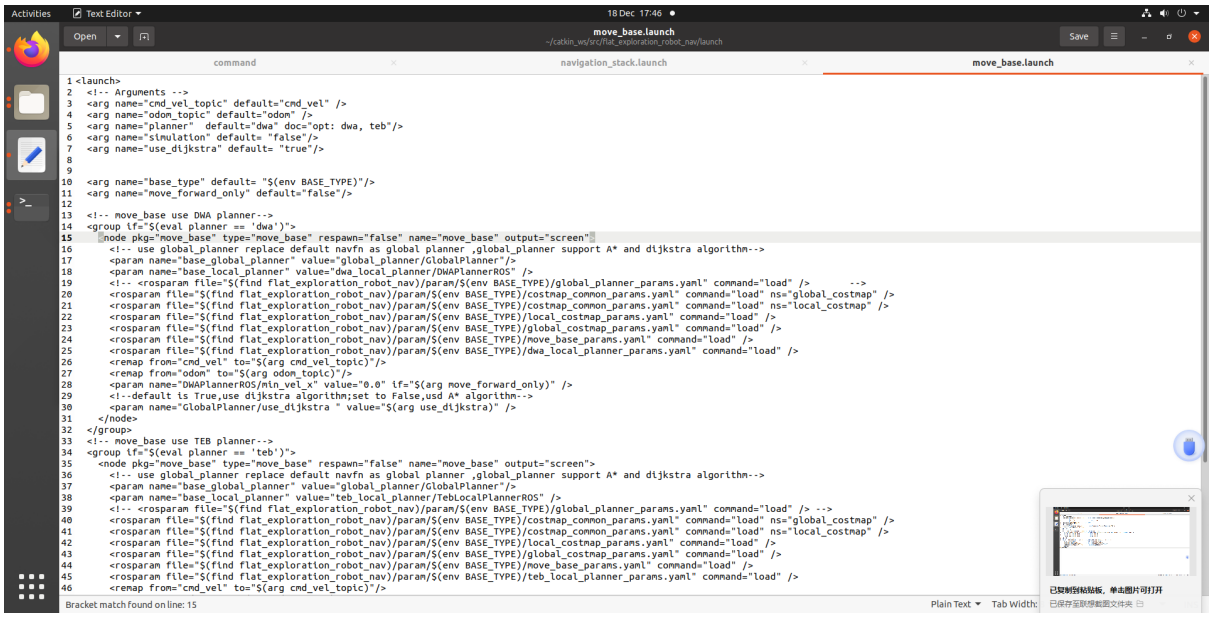

Figure 16

4. Face Detection

The face detection system utilizes camera sensors mounted on the robot, along with OpenCV-based algorithms, to recognize and capture human facial features. A ROS node is developed to subscribe to the camera topic and perform face detection using OpenCV. In the ROS package, a Python script is created, employing the Haar feature-based classifier provided by OpenCV for detecting faces in images. After making the script executable using a command, the script,

when run, displays the image in an OpenCV window and draws a rectangular box at the corresponding position upon detecting a face. Change the permissions of the script file to make it executable before running it.

Firstly, the necessary packages are installed with the command "sudo apt-get install ros-noetic-cv-bridge ros-noetic-image-transport", where OpenCV is used for image processing and face detection, and ROS cv_bridge is for converting ROS image messages into OpenCV image formats. The published camera topic "/camera/color/image_raw" is utilized. Then, a new ROS package is created or an existing package is modified to add a Python script, as shown in Figure 17. This script uses the cv_bridge library to convert the image data received from the ROS topic (of type sensor_msgs/Image) into a format that OpenCV can process. The converted image is a standard OpenCV image encoded in BGR color. The Haar Cascade Classifier from OpenCV, located at '/usr/share/opencv4/haarcascades/haarcascade_frontalface_default.xml', is loaded. This classifier is a pre-trained model used to identify frontal faces in images. The BGR image is converted to a grayscale image as face detection typically occurs on grayscale images. The classifier detects faces on the grayscale image using the detectMultiScale function, which is designed to identify faces of varying sizes in the image. For each detected face, a blue rectangular box is drawn on the original BGR image to indicate the location of the face. The region of each detected face is cropped and saved as a separate image file (detected_face.jpg) for further processing or archiving.

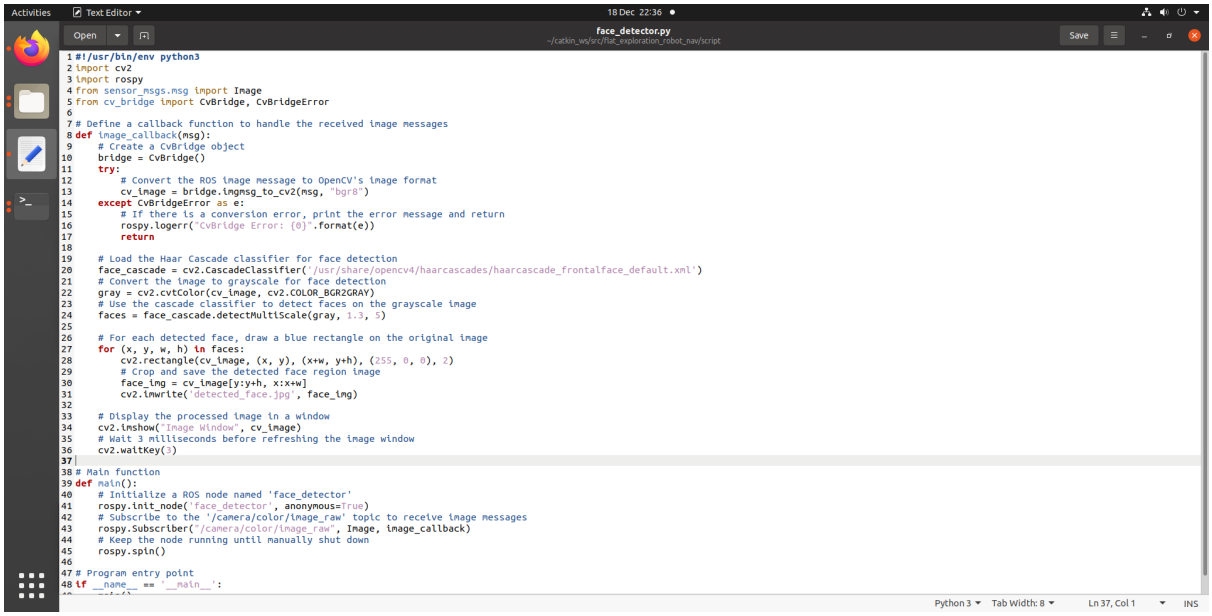

Figure 17

5. Discussion and reflection

5.1 World Design and Human Creation

Creating an indoor environment in Gazebo presented multiple challenges, especially in ensuring enough space for robot navigation while integrating obstacles and pathways that mimic a real indoor setting. The addition of various objects such as chairs, coffee tables, bookshelves, and human models increased environmental complexity. Adjustments in MakeHuman and subsequent exporting and testing in Blender was a meticulous process. It required a keen insight into details to ensure that the model was not only visually appealing but also functionally fit for simulation. This process highlighted the importance of consistency in naming and file management, which is critical for seamless integration into Gazebo.

5.2 Sensor Selection and Robot Modification

Integrating the LiDAR sensor into the robot's xacro file was straightforward, but ensuring its optimal placement for effective environmental scanning required repeated trials. Adjustments made in the gazebo.xacro file were vital for the robot's performance in the simulated world. The configuration of the TF tree was crucial for ensuring accurate and reliable navigation, emphasizing the importance of precise transformation relationships between various robot components.

5.3 Autonomous Navigation

The SLAM process with gmapping proved the effectiveness of the sensor setup and the robot's ability to map an unknown environment, with the map-saving process being highly efficient. The fidelity of the map greatly depended on the thoroughness of the initial scan. Navigation stack testing revealed the robot's ability to dynamically plan and adjust its path. However, occasionally manual adjustments were needed to align the robot's initial position in Rviz with the map, indicating the need for a more robust method of initial positioning.

5.4 Face Detection

Face detection using OpenCV and ROS highlighted the potential of integrating advanced computer vision technology with robotic technology. The process of converting ROS image messages into OpenCV format went smoothly, and the Haar Cascade Classifier proved to be

effective for face detection. However, the accuracy of detection depended on the quality of the input image and was also significantly influenced by lighting conditions and robot movement.

6. Conclusion

This assignment, from simulation and sensor integration to autonomous navigation and face detection, underscored the importance of detailed planning in environment creation, the effectiveness of integrating multiple sensors to achieve robust navigation, and the potential of computer vision to enhance robot perception. The challenges encountered mainly related to initial positioning for navigation and face recognition.