inMind Academy
Robotics Track

Final Project:

Navigation with NAV2 and Behavior Trees

Submitted by
Alexa Fahel
Gregory Sarrouf
Jack Dergham

Submitted on
27 - 3 - 2025

Submitted to
Mr. Ziad Saoud

Table of Contents

Table of figures

I. Introduction

The aim of this project is to develop a fully autonomous navigation system for a simulated robot using behavior trees, the ROS2 framework and the Nav2 stack. The robot will be equipped with the ability to map its environment using Simultaneous Localization and Mapping (SLAM), localize itself within that map using Adaptive Monte Carlo Localization (AMCL), and navigate through a series of predefined waypoints while continuously monitoring its battery level. This integration ensures smooth path planning and effective obstacle avoidance, making the robot's movement efficient and safe. When the battery is low, the robot will autonomously navigate to a designated charging station, remain stationary for a specified period to simulate recharging, and then resume its tasks. To manage the robot's decision-making process, we will utilize BehaviorTree.CPP, which allows for dynamic transitions between different states based on the battery status and task completion. By leveraging these technologies, we can create a sophisticated autonomous system capable of adapting to changing conditions and performing complex tasks independently. [1]

II. Behavior Trees

A. Theoretical background

Behavior trees are hierarchical structures traversed through “ticks” to model decision-making processes. They organize the behaviors into a tree made up of “nodes” which is traversed in a specific order to determine the system’s behavior. Each node performs a task and can either return success, failure, or running when ticked. Generally, these nodes are represented according to the symbols in figure 1 and can be divided into four categories:

1. **Control nodes** that have many children and can be
    - **Sequence nodes:** execute children in order and return success when they all succeed or failure when one of them fails
    - **Fallback nodes:** execute their children sequentially until one returns success or all children return failure
    - **Parallel nodes:** execute their children in parallel and can be configured to return success or failure depending on how many children succeed
2. **Decorator nodes** that modify a single child node according to a custom policy
3. **Condition nodes** that do not alter the system but merely check if a condition is satisfied
4. **Action nodes** that have no children and perform a specified action [2]

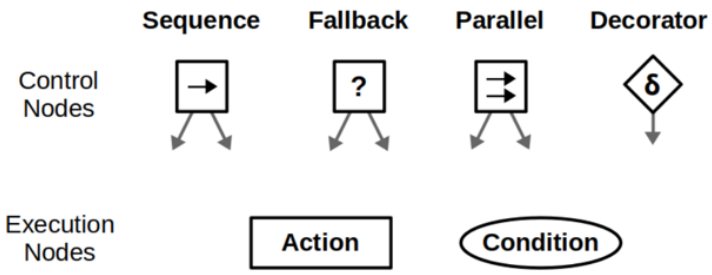


Figure 1: Behavior tree nodes

B. Practical Implementation

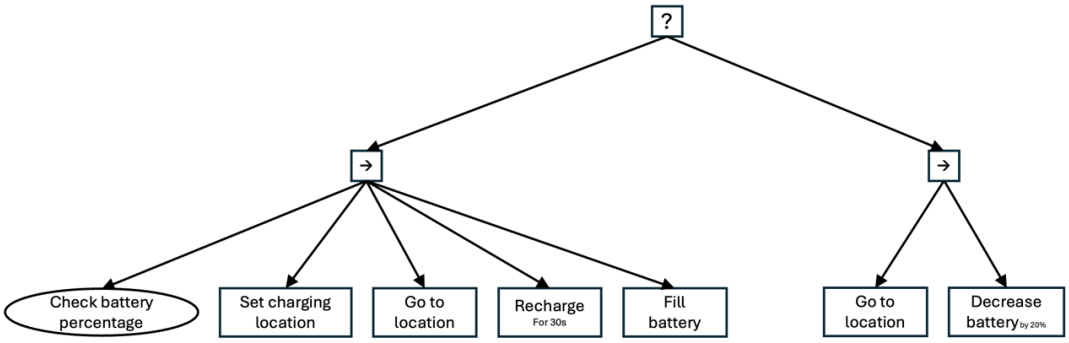


Figure 2: Behavior tree diagram

The behavior tree designed for this project can be seen in figure 2. The tree begins with a fallback root node that executes the two following nodes sequentially until one returns success. Next we have two sets of sequence nodes that go through their children in order and return success when all the children succeed. The first sequence node starts by checking the battery percentage and returns success if the battery is less than 25%. If the battery is indeed low, the robot is given the charging location and told to go there. Next the robot sits in the charging dock for 30 seconds and resets the battery to 100%. The second sequence node goes to the next specified location from the location array and once this action returns success, it decrements the battery variable by 20%.
In summary, the robot goes through the location array and decrements its battery every time it reaches a new location. However if the battery is less than 25%, the robot halts and goes back to the charging dock to fill its battery before resuming its activities.

The way this concept was implemented through code could be seen below:

```
<root BTCPP_format="4">
  <BehaviorTree ID="NavBT">
      <Fallback name="rootFallback">

        <Sequence name="BatterySequence">
          <isBatteryLow/>
          <setChargingLocation/>
          <goToLocation/>
          <recharge/>
          <fillBattery/>
        </Sequence>

        <Sequence name="NavigationSequence">
          <goToLocation/>
          <decreaseBattery/>
        </Sequence>

      </Fallback>
  </BehaviorTree>
</root>
```

Figure 3: bt_tree.xml

A bt_tree.xml file was written to define the structure of the behavior tree. What this code snippet does is translate figure 2 into code. Next a C++ file was written to specify the function of each of the above behavior tree nodes.
The code begins by including iostream, vector, and chrono, which allows us to use input/output operations, create vectors, and set waiting times. Action_node.h and bt_factory.h are also included to be able to create some action node later in the code and define the functionality of the tree nodes.

```cpp
#include <iostream>
#include <chrono>
#include <vector>
#include "behaviortree_cpp/action_node.h"
#include "behaviortree_cpp/bt_factory.h"

using namespace std::chrono_literals;

int battery_level = 100;
int locIdx = 0;

const std::vector<std::vector<float>> locationCoords = {
  { 0.0f, -2.0f },    // Point A (Right)
  { 2.0f, 0.0f },     // Point B (Up)
  { 0.0f, 2.0f },     // Point C (Left)
  { -2.0f, 0.0f },    // Point D (Down)
  { -0.55f, 0.55f }   // Charging Station
};

const std::vector<std::string> locationNames = {
  "Point A",
  "Point B",
  "Point C",
  "Point D",
  "Charging Station"
};
```

The variable battery_level keeps track of the robot's battery percentage and the locIdx variable keeps track of which location the robot should navigate to next. It points to the coordinates and name of the location which are stored in the two vectors locationCoords and location Names.
The next task ahead is defining the functionality of each tree node. To do that, functions were written for the nodes with a simple behavior that just needed to return success or failure without keeping track of any internal variables, and classes were used when the behavior involved maintaining an internal state or required slightly more complex execution logic.

```cpp
BT::NodeStatus isBatteryLow() {
  if (battery_level < 25){
    return  BT::NodeStatus::SUCCESS;
  }
  else{
    return BT::NodeStatus::FAILURE;
  }
}
```

```
BT::NodeStatus setChargingLocation() {
  locIdx = 4;
  return BT::NodeStatus::SUCCESS;
}

BT::NodeStatus fillBattery() {
  battery_level = 100;
  return BT::NodeStatus::SUCCESS;
}

BT::NodeStatus decreaseBattery() {
  battery_level -= 20;
  std::cout << "Battery is now " << battery_level << "%" << std::endl;
  return BT::NodeStatus::SUCCESS;
}
```

isBatteryLow returns success if the battery is belowe 25% and returns failure otherwise. setChargingLocation specifies that the next location the robot has to go to is the charging station. fillBattery sets the battery level to 100. decreaseBattery decrement the battery_level by 20% and prints the current battery percentage before returning success.

```
class GoToLocation : public BT::SyncActionNode {
public:
  GoToLocation(const std::string& name) : BT::SyncActionNode(name, {}) {}

  BT::NodeStatus tick() override {
    std::string scriptCommand =
      "../scripts/nav_to_location.py " +
      std::to_string(locationCoords[locIdx][0]) +
      " " +
      std::to_string(locationCoords[locIdx][1]);

    std::cout << "Navigating to: " << locationNames[locIdx] << std::endl;
    locIdx = (locIdx == 4) ? 0 : locIdx = (locIdx + 1) % 4;

    system(scriptCommand.c_str());
    return BT::NodeStatus::SUCCESS;  // Simulate success for now
  }
};

class Recharge : public BT::SyncActionNode {
public:
  Recharge(const std::string& name) : BT::SyncActionNode(name, {}) {}

  BT::NodeStatus tick() override {
    std::cout << "Recharging for 30s..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(30));
    return BT::NodeStatus::SUCCESS;
  }};
```

For the last two behavior tree nodes, classes were created that inherit from the SyncActionNode class which is defined in action_node.h included in the beginning of the code. SyncActionNode is a type of action node in BehaviorTree.CPP that executes its tick() function synchronously and was used in this project since we don't require our nodes to run asynchronoulsy in the background. Both classes include a constructor that passes the node's name as a parameter to the parent class's constructor and override the tick() function of their whose job it is to define the behavior of each node when it is ticked during execution.
The GoToLocation node constructs a command string to call the nav_to_location.py script which directs the robot to navigate to a specific location. It passes the coordinates of the next location as arguments to the script and executes the command. The locIdx variable is then incremented by 1 to move to the next location and set to 0 if it was previously 4 (the charging station). The recharge node waits for 30 seconds to simulate the robot charging before returning success.

```
int main()
{
  BT::BehaviorTreeFactory factory;

  factory.registerNodeType<GoToLocation>("goToLocation");
  factory.registerNodeType<Recharge>("recharge");

  factory.registerSimpleCondition("isBatteryLow", std::bind(isBatteryLow));
  factory.registerSimpleAction("setChargingLocation", std::bind(setChargingLocation));
  factory.registerSimpleAction("fillBattery", std::bind(fillBattery));
  factory.registerSimpleAction("decreaseBattery", std::bind(decreaseBattery));

  // Create Tree
  auto tree = factory.createTreeFromFile("./../bt_tree.xml");

  // execute the tree
  while (true){
    tree.tickWhileRunning();
    std::cout << "Iteration complete\n\n";
  }

  return 0;
}
```

Figure 4: bt_nav.cpp

The main function initializes a BehaviorTreeFactory object which is used to register and create our behavior tree. registerNodeType is used to register the GoToLocation and Recharge classes as behavior tree nodes, and registerSimpleCondition is used to register the four functions described previoously as tree nodes. createTreeFromFile reads the tree structure from the xml file and creates the executable behavior tree, and the while true loop continuously ticks the behavior tree to execute its nodes.

III. Simultaneous Localization and Mapping (SLAM)

A. Theoretical background

Simultaneous Localization and Mapping (SLAM) is a technique used in robotics to enable a robot to construct a map of an unknown environment while simultaneously determining its own position within that map. SLAM algorithms combine sensor data from sources like LiDAR, cameras, or sonar with motion estimation techniques to update a dynamic representation of the surroundings. By continuously refining the map and correcting for localization errors using probabilistic methods such as Kalman filters or particle filters, SLAM allows robots to navigate accurately without relying on pre-existing maps. This capability is essential for autonomous robots operating in complex or GPS-denied environments.

In the following figure, red lines indicate the obstacles currently being detected by the robot's LiDAR cameras, and the black lines indicate mapped obstacles. [3]

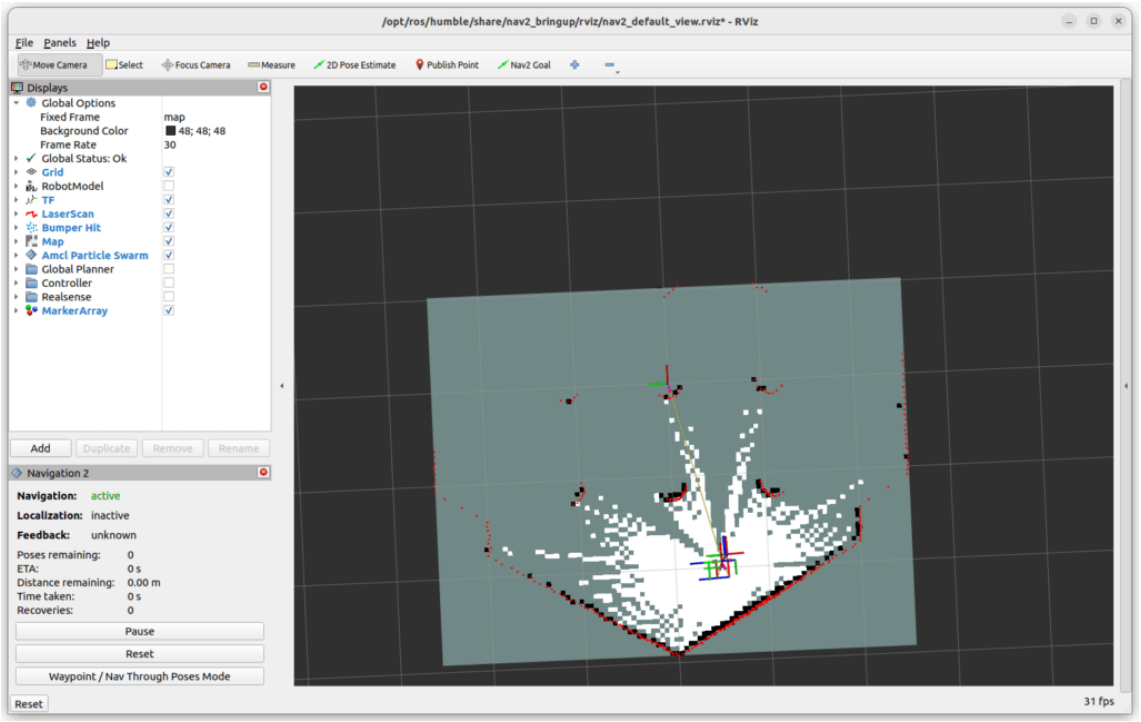


Figure 5: SLAM map

B. Practical implementation

In this project, the area chosen for mapping was the turtlebot3_world map due to its moderate size and ease of mapping. After setting up the Gazebo simulation as well as RViz2 for visualization, slam_toolbox was the main tool used for mapping purposes. In order to fully explore the area and correctly map all available obstacles, the teleop navigator tool was also used for manual control over the robot. This allowed the LiDAR cameras to slowly and accurately capture the surroundings, resulting in a .pgm file containing all available obstacle and free ground data. The following figure is that aforementioned file:

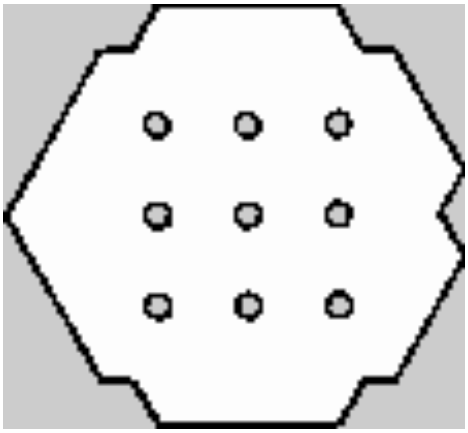


Figure 6: world_map_save.pgm

IV. Adaptive Monte Carlo Localization (AMCL) and Navigation

A. Theoretical background

Adaptive Monte Carlo Localization is a probabilistic localization technique used in robotics to estimate a robot's position within a known map by continuously refining its belief using particle filtering. AMCL represents the robot's possible locations as a set of weighted particles, which are updated based on sensor readings and motion data. The algorithm adjusts the number of particles dynamically—using more particles in uncertain situations and fewer when the robot's position is well-known—optimizing computational efficiency. By resampling particles based on their likelihood of representing the true position, AMCL enables robust and adaptive localization in dynamic environments. [4]

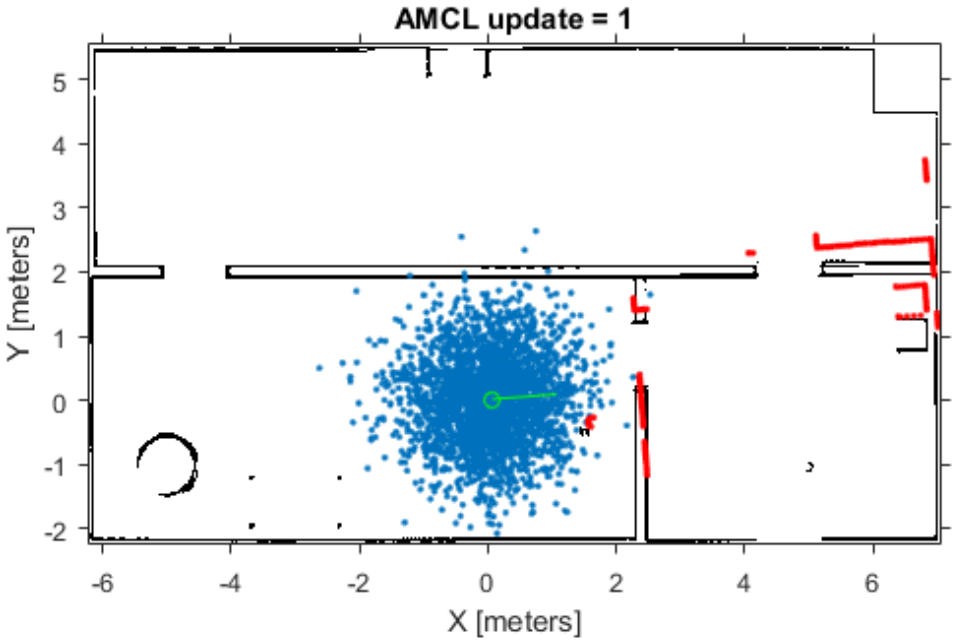


Figure 7: AMCL map

B. Practical implementation

After mapping the turtlebot3_world, it was then possible to start up the Gazebo simulation and RViz2 for visualization with the world_map_save.pgm file as the base. The next step was to specify the starting position of the robot through a particle estimation, from which the navigation2 algorithm could pinpoint the exact location of the robot based on its surroundings. In the following figure, the green dots are the estimation particles used.

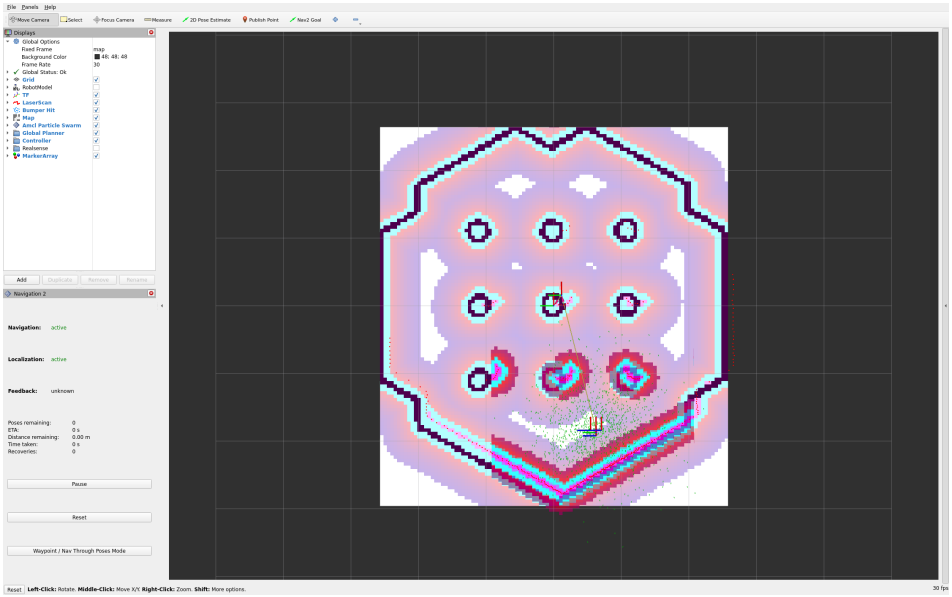


Figure 8: AMCL in RViz2

Having localized the robot with respect to the obstacles, the navigation algorithm can now receive the coordinates of a point to navigate to safely, all the while avoiding all blocking obstacles.

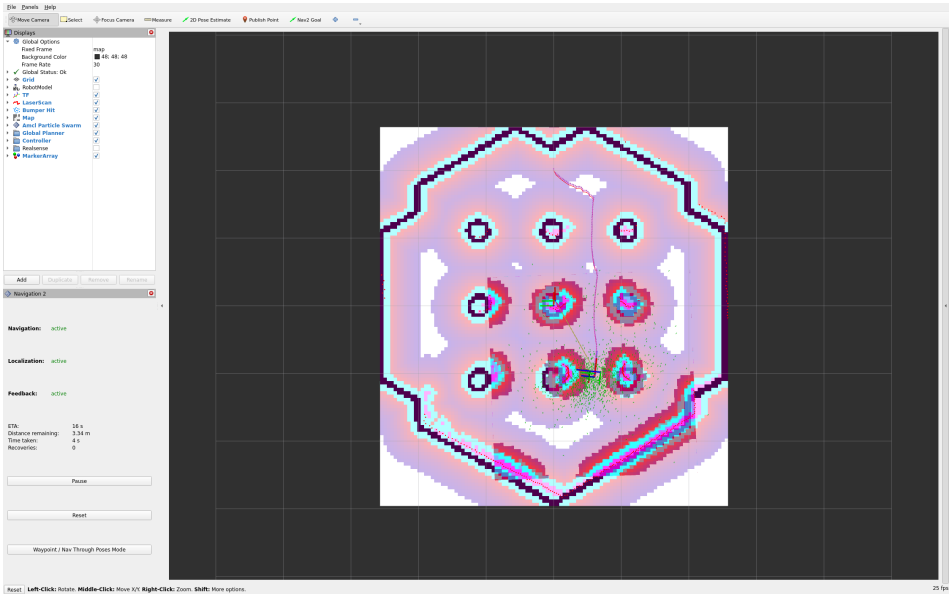
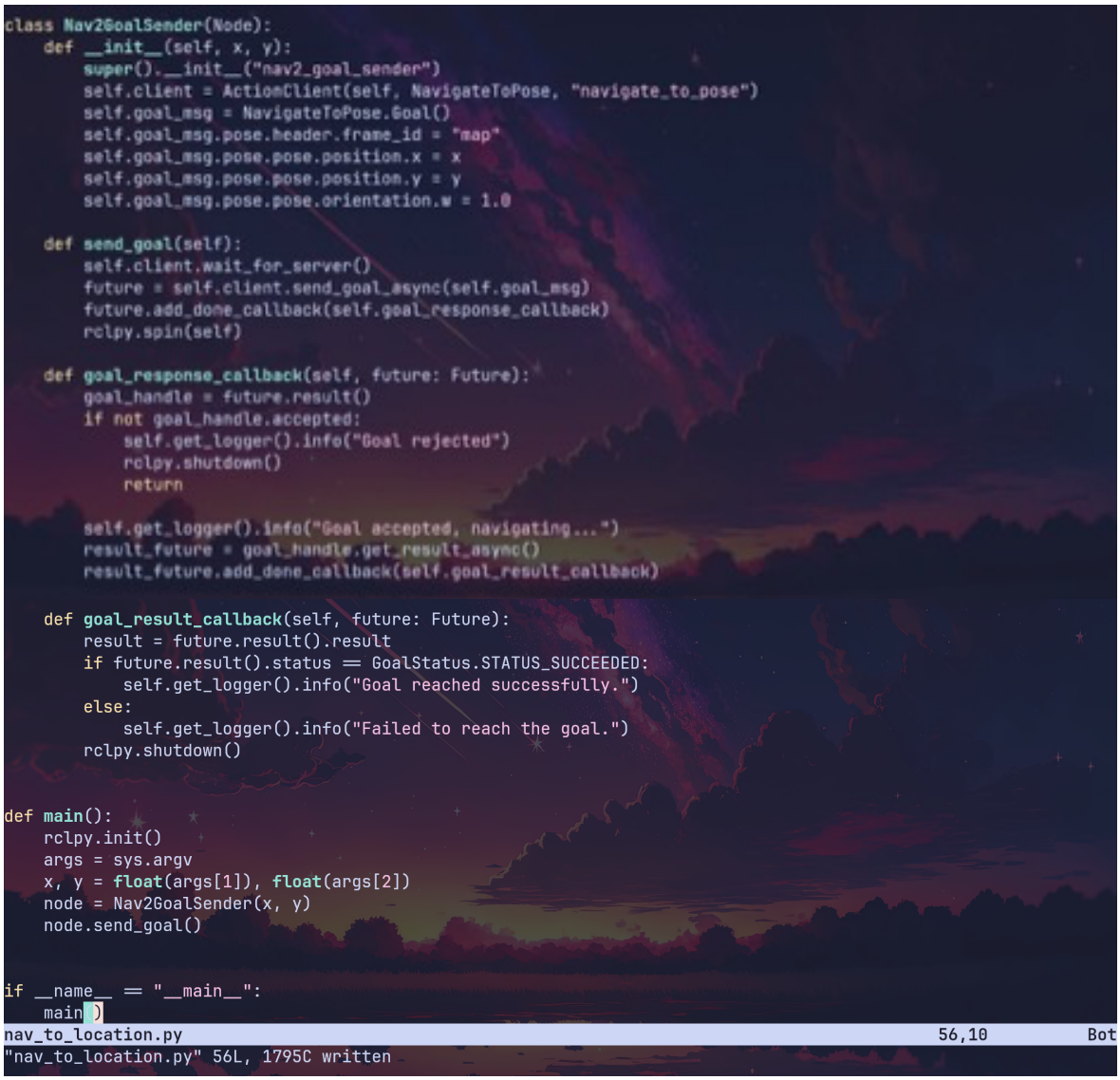


Figure 9: Turtlebot navigating to point (2.0, 0.0)

This navigation is achievable through a python script that publishes the coordinates of the desired point as an action. The coordinates are sent as an argument while calling the script.



```
class Nav2GoalSender(Node):
    def __init__(self, x, y):
        super().__init__("nav2_goal_sender")
        self.client = ActionClient(self, NavigateToPose, "navigate_to_pose")
        self.goal_msg = NavigateToPose.Goal()
        self.goal_msg.pose.header.frame_id = "map"
        self.goal_msg.pose.pose.position.x = x
        self.goal_msg.pose.pose.position.y = y
        self.goal_msg.pose.pose.orientation.w = 1.0

    def send_goal(self):
        self.client.wait_for_server()
        future = self.client.send_goal_async(self.goal_msg)
        future.add_done_callback(self.goal_response_callback)
        rclpy.spin(self)

    def goal_response_callback(self, future: Future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().info("Goal rejected")
            rclpy.shutdown()
            return

        self.get_logger().info("Goal accepted, navigating...")
        result_future = goal_handle.get_result_async()
        result_future.add_done_callback(self.goal_result_callback)

    def goal_result_callback(self, future: Future):
        result = future.result().result
        if future.result().status == GoalStatus.STATUS_SUCCEEDED:
            self.get_logger().info("Goal reached successfully.")
        else:
            self.get_logger().info("Failed to reach the goal.")
        rclpy.shutdown()


def main():
    rclpy.init()
    args = sys.argv
    x, y = float(args[1]), float(args[2])
    node = Nav2GoalSender(x, y)
    node.send_goal()


if __name__ == "__main__":
    main()
nav_to_location.py                                    56,10         Bot
"nav_to_location.py" 56L, 1795C written
```

Figure 10: Python navigation script

V. Building and running the project

To run this project, many dependencies need to be downloaded first. Since we use BehaviorTree.CPP to build our behavior tree, the following commands have to be executed in the project directory

```
git clone git@github.com:BehaviorTree/BehaviorTree.CPP.git
cd BehaviorTree.CPP
sudo apt-get install libzmq3-dev libboost-dev
```

```
mkdir build
cd build
cmake ..
make
sudo make install
```

```
cd ../..
mkdir build
cd build
cmake ..
make
```

To start Gazebo run:

```
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

and to start the navigation program with RViz2 run the following, while being sure to input your actual file path

```
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True
map:=/path/to/map/world_map_save.yaml
```

Then add the 2D point estimate on the Rviz2 map to approximate the location of the robot, and run the following commands from your project's directory

```
cd build
./bt_nav
```

Once done, the robot should start moving inside gazebo and executing the behavior tree.

VI. Conclusion

This project successfully demonstrates the integration of the Nav2 stack with BehaviorTree.CPP to create a fully autonomous robot navigation system. The robot efficiently maps its environment using SLAM, localizes itself with AMCL, and navigates between waypoints while avoiding obstacles, showcasing the robust capabilities of the Nav2 stack. The incorporation of battery monitoring and recharging capabilities adds a layer of realism and practicality to the simulation, making it more applicable to real-world scenarios. By dynamically transitioning between navigation, battery monitoring, and recharging states, the robot showcases intelligent decision-making capabilities, built using BehaviorTree.CPP. The successful implementation of this autonomous system underscores the capability of robots to perform complex tasks independently, paving the way for more advanced autonomous systems in the future.

VII. References

[1] Saoud, Z. (2025, March 24). *Navigation with NAV2 and Behavior Trees*. Notion. https://jeweled-blackbird-fad.notion.site/Navigation-with-NAV2-and-Behavior-Trees-1c00ecf9765c803f99cffdec44eff189

[2] Auryn Robotics. (n.d.). *Introduction to BTs*. BehaviorTree. https://www.behaviortree.dev/docs/learn-the-basics/bt_basics/

[3] Bermudez, L. (2024, April 17). Overview of SLAM - machinevision. *Medium*. https://medium.com/machinevision/overview-of-slam-50b7f49903b7

[4] Das, S. (2025, January 2). *Robot localization in a mapped environment using Adaptive Monte Carlo algorithm*. Arxiv. https://arxiv.org/html/2501.01153v1