**Assignment 2 - ROS – Inverse Kinematics of a DDMWR- PRATHIBHA SOBHA**

Github link:<https://github.com/prathibhasobha/ECG711-Assignments/tree/main/Assignment2/ROS_inverse_kinematics>

Youtube: <https://youtu.be/evzx7DotMmw>

Steps taken:

* **Install ROS**: First, I installed ROS on your system.
* **Created a ROS Workspace**: Created a catkin workspace where you will organize your ROS packages.
* **Created a ROS Package**: Created a ROS package that will contain my C++ program (driver. cpp and simulator. cpp). It was done by using a package named the `catkin\_create\_pkg` command.
* **Create Your C++ Program**: Copied my C++ program within the package I just created. I have to place it in the `src/` directory of your package. Make sure to include the necessary ROS headers and libraries in your C++ program (#include <ros/ros.h>) and link against ‘roscpp’.
* **Edit CMakeLists.txt**: Open the `CMakeLists.txt` file located in your package's directory.

* **Build the Package**: Build the package using ‘catkin\_make’ in your catkin workspace:
* **Source the Workspace**: After building, make sure to source your workspace to access the newly built package and its executables.
* **Run the C++ ROS Node**: You can now run your C++ ROS node using ‘rosrun’.

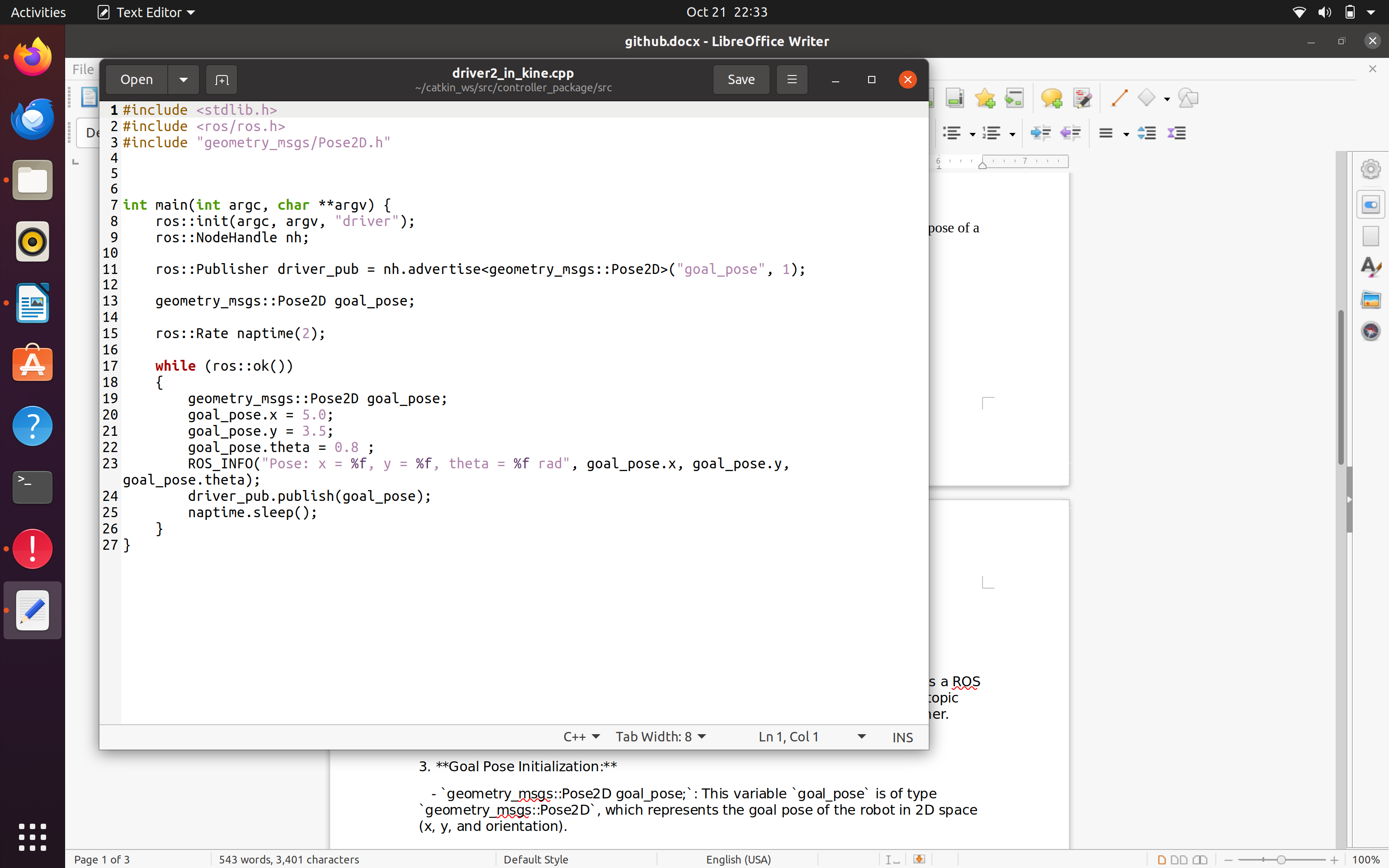
Now my C++ ROS program is all set to run the code.

Description of code:

Here,I created two separate C++ files: one for the driver and another for the simulator.

* Driver (driver2\_in\_kine.cpp):

The provided driver code is for a ROS node responsible for publishing the goal pose of a mobile robot.

**1. Node Initialization**:

Initializes the ROS node, specifying the node name as "driver."

2. NodeHandle:

NodeHandle is a handle for this ROS node. It provides a way to create publishers, subscribers, and access other ROS features.

3.Publisher Initialization:

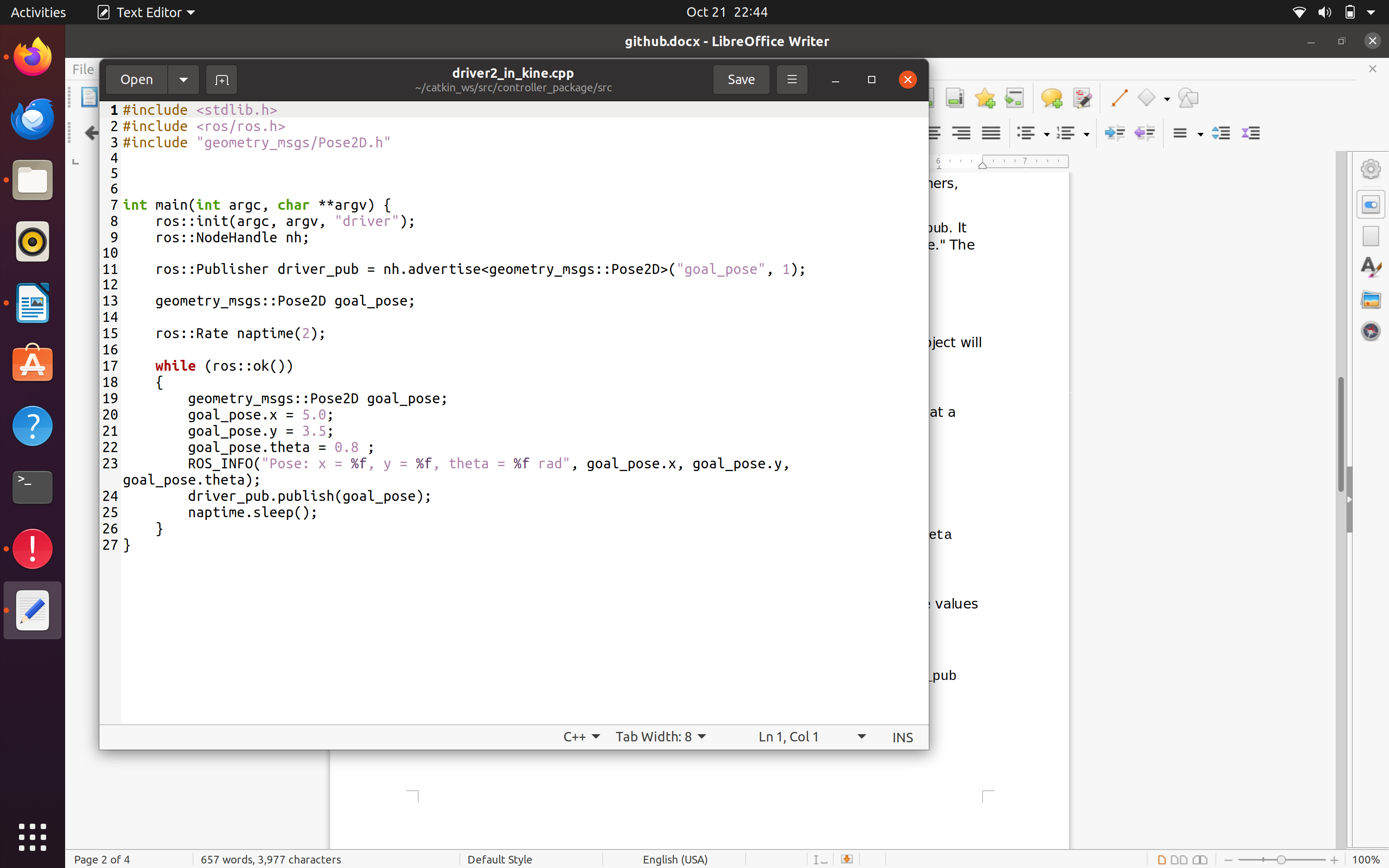
This initializes a ROS publisher named driver\_pub. It publishes messages of type geometry\_msgs::Pose2D on the topic "goal\_pose." The number 1 specifies the message queue size.

4.Create a **geometry\_msgs::Pose2D** object:

This line creates a geometry\_msgs::Pose2D object called goal\_pose. This object will hold the 2D pose data.

5.Set Rate:

This line sets the publishing rate to 2 Hz. It means that the loop will execute at a frequency of 2 times per second.



6.Main Loop:

This loop will run as long as ROS is running .

7.Set the Goal Pose:

In each iteration of the loop, a new goal\_pose is created, and its x, y, and theta values are set.

8.Log the Pose:

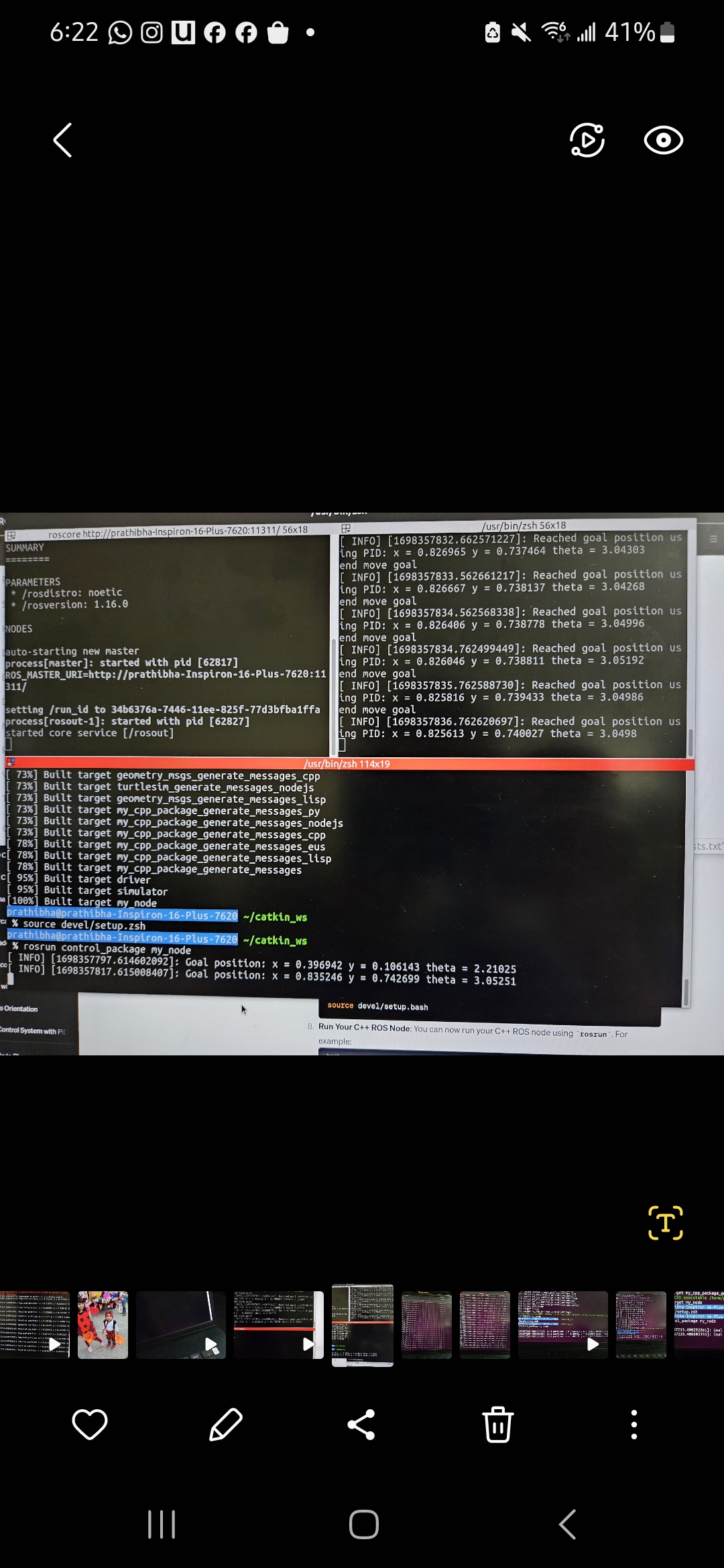
This line logs the current pose to the console using ROS\_INFO. It will print the values of x, y, and theta in the message.

9.Publish the Pose:

This line publishes the goal\_pose to the "goal\_pose" topic using the driver\_pub publisher.

The code essentially creates a ROS node that repeatedly publishes a 2D pose message with predefined values on the "goal\_pose" topic at a rate of 2 Hz. This can be useful for testing or simulating a robot's movement or control system within the ROS ecosystem.

Result:



* Controller (controller2\_in\_kine.cpp)

This code is a C++ program that controls a robot's movement using a PID (Proportional-Integral-Derivative) controller in the context of the Robot Operating System (ROS).

1. **Callback Function**:

This function is a callback that gets called when new data is received on the "goal\_pose" topic. It updates the `gpose` variable with the new goal pose. It assumes that there is a global variable `gpose` defined elsewhere in the code.

1. **Main Function**:

The `main` function initializes the ROS node with the name "PID\_Control", and sets up a subscriber to the "goal\_pose" topic for receiving goal pose information. It also creates a publisher for sending velocity commands to the robot via the "cmd\_vel" topic.

**3. Distance and Angle Calculation Functions**:

These functions, “getDistance” and “getAngle”, are used to calculate the distance and angle between the current robot pose and the goal pose. “getDistance” calculates the Euclidean distance, and “getAngle” calculates the angle.

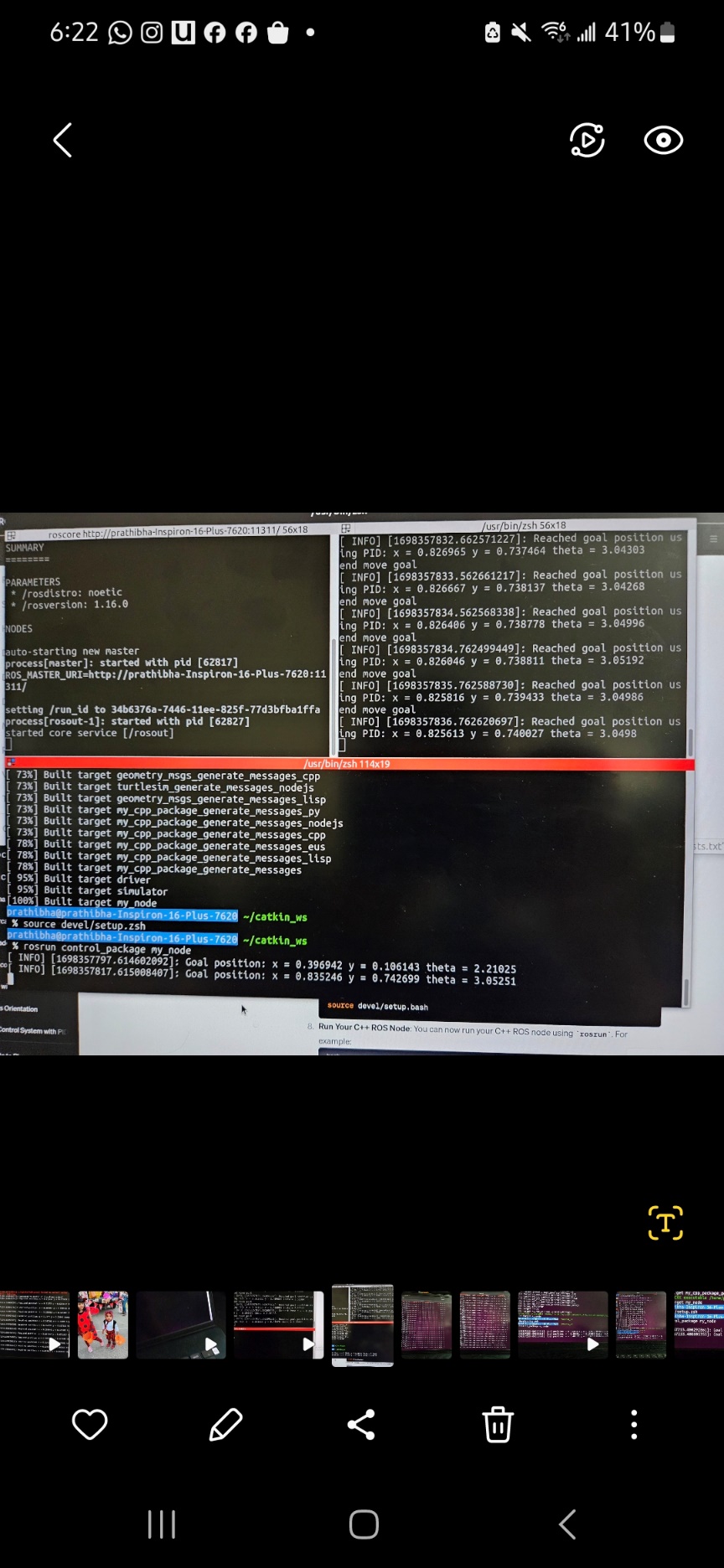
**4.PID Controller:**

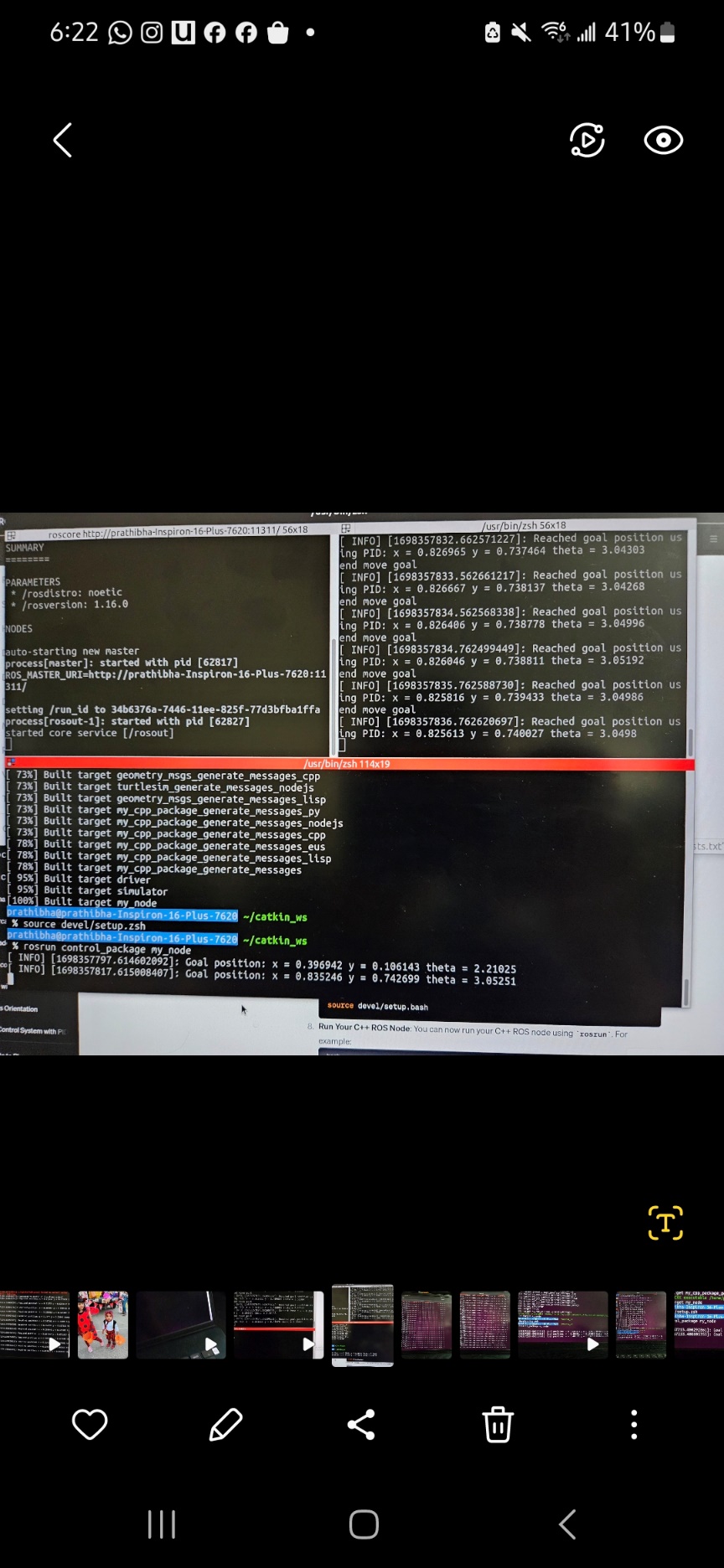
This is the core of the code, where the PID controller is implemented. It computes control commands to drive the robot from its current position to the goal pose while maintaining a specified distance tolerance.

Inside the “moveGoal” function, this function implements the PID controller for moving the robot to the desired position and orientation. It uses PID control to adjust the linear and angular velocities based on pose errors. The robot moves linearly towards the goal while also trying to align its orientation. The control loop continues until the robot reaches the goal within the specified tolerance. After reaching the goal, the code rotates the robot to the goal orientation.

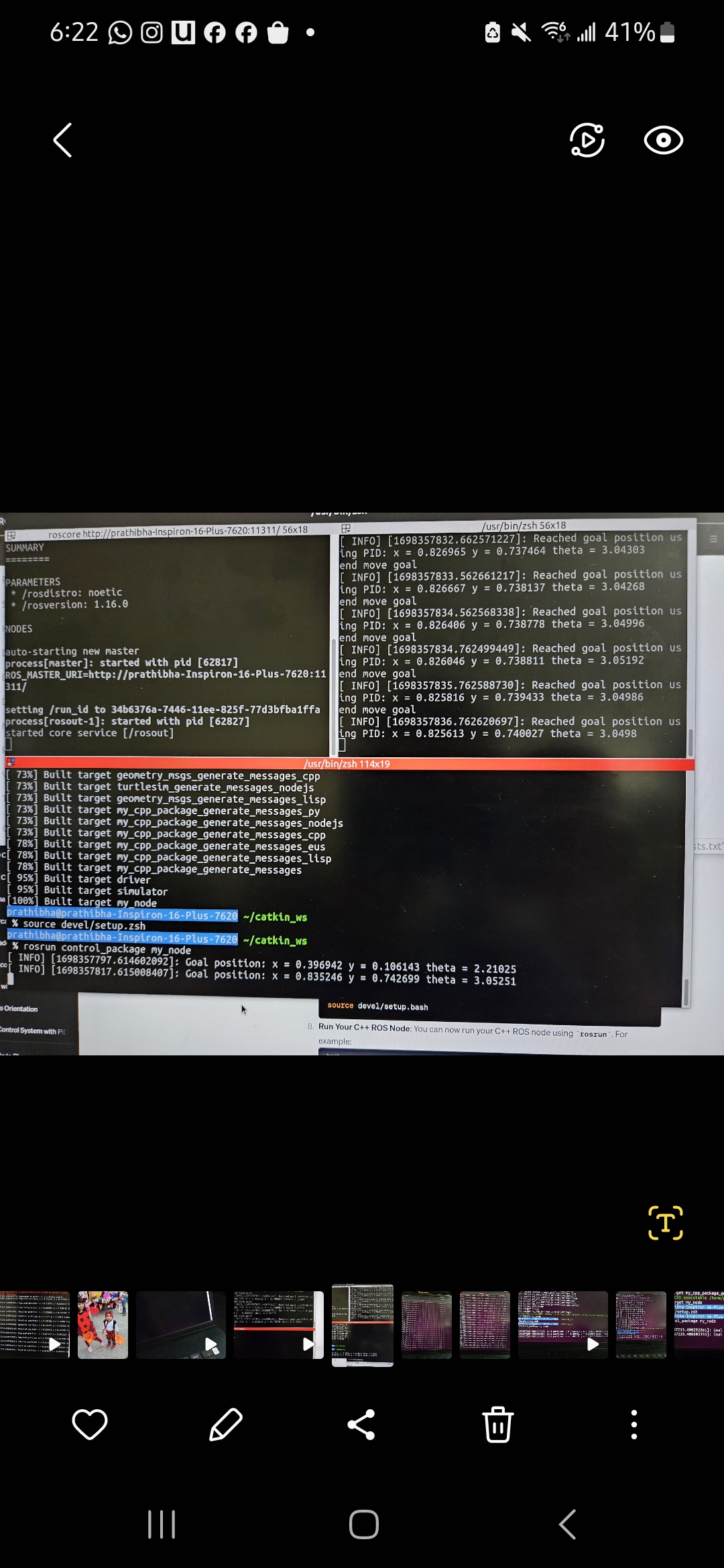
The program continuously publishes velocity commands to control the robot's motion and updates its current pose based on the calculated velocities. The PID controller uses proportional, integral, and derivative terms to minimize position and orientation errors, helping the robot reach the goal pose accurately.

Result:



Top of Form

Bottom of Form



This result shows that the PID controller effectively reached the goal position.