**Autonomous Pipeline Navigation with SLAM**

**Initial approach:**

Using 4 ultrasonic sensors with an esp-32 and use a simple localisation algorithm to maintain optimal distance from walls but I got suggestions from <https://www.solinas.in> a startup company in IITM who suggested me to go with a more advanced approach. They gave me a look into their patented bot (<https://www.youtube.com/watch?v=q-nZmhI51ck&t=1s>) I have included everything out of the NDA with them.

**Simultaneous Localization and Mapping (SLAM)**

SLAM is a computational technique that allows a robot to construct or update a map of an unknown environment while simultaneously tracking its own location within that map. It uses sensors such as LIDAR, cameras, or sonar to gather environmental data, which is then processed to create an occupancy grid map. This map represents traversable and non-traversable areas in a grid format.

**Occupancy Grid Map**

An occupancy grid map is a 2D representation of the environment where each cell has a value indicating whether it is occupied, free, or unknown. It is used by the robot to navigate and plan paths while avoiding obstacles.

**A\* Path Planning Algorithm**

A\* is a popular graph-based pathfinding algorithm that calculates the shortest path from a start point to a goal point. It uses the following principles:

1. **Nodes and Edges**: The environment is represented as a graph, where nodes are grid cells and edges connect adjacent cells.
2. **Cost Function**:
   * **G-Cost**: The cost of moving from the start node to the current node.
   * **H-Cost**: A heuristic estimate of the cost from the current node to the goal. The Euclidean distance is often used as the heuristic.
   * **F-Cost**: The sum of G-Cost and H-Cost, guiding the search toward the goal.
3. **Priority Queue**: A\* uses a priority queue to explore nodes with the lowest F-Cost first.
4. **Reconstruction**: Once the goal is reached, the algorithm backtracks to reconstruct the optimal path.

**Robot Control**

Robot control involves calculating the appropriate linear and angular velocities to navigate the robot toward a target while maintaining stability and avoiding collisions.

**Proportional Control (P-Control)**

Proportional control adjusts the robot’s movement based on the difference (error) between the current and desired states:

1. **Linear Velocity**: Proportional to the distance between the robot and the target.
2. **Angular Velocity**: Proportional to the angular error between the robot’s orientation and the direction to the target.

**ROS (Robot Operating System)**

ROS provides a framework for developing robotic systems, with key components including:

1. **Nodes**: Independent processes performing specific tasks.
2. **Topics**: Channels for communication between nodes using a publish/subscribe mechanism.
3. **Messages**: Standardized data formats used in communication (e.g., OccupancyGrid, Odometry, and Twist).
4. **Rate**: Controls the frequency of the main loop, ensuring consistent execution.

**Sensor Integration**

**Odometry**

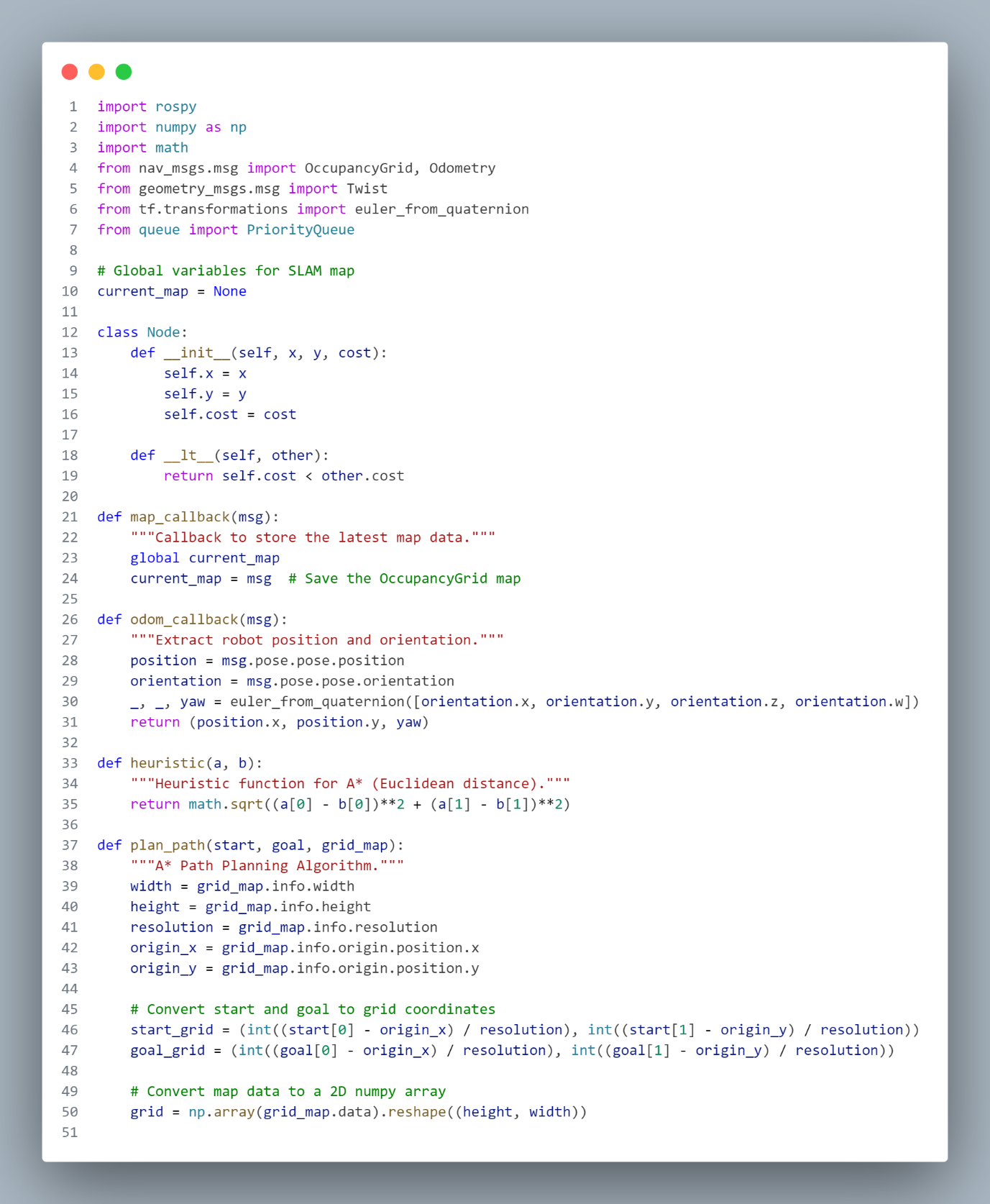
Odometry data provides the robot’s position and orientation in real-time, calculated using wheel encoders or inertial sensors.

**LIDAR**

LIDAR sensors create a point cloud representation of the environment, which is converted into an occupancy grid map for SLAM.

These concepts work together to enable autonomous navigation in unknown environments. The robot creates a map using SLAM, plans a path using A\*, and navigates the path using proportional control while continuously updating its position and avoiding obstacles.

***CODE:***







**Code explanation:**

This script enables a robot to autonomously navigate a pipeline environment using SLAM (Simultaneous Localization and Mapping). The robot uses an A\* path-planning algorithm to determine the optimal path toward a target while avoiding obstacles detected in the SLAM-generated occupancy grid map. It computes control commands to drive the robot along the planned path.

**Key Components**

**Global Variables**

* current\_map: A global variable that stores the SLAM-generated occupancy grid map. This is updated whenever a new map message is received on the /map topic.

**Class: Node**

* Represents a grid cell in the A\* algorithm.
* **Attributes**:
  + x, y: Grid coordinates.
  + cost: Path cost from the start to this node.
* **Methods**:
  + \_\_lt\_\_: Compares nodes by their cost, enabling use in priority queues.

**Callbacks**

1. **map\_callback(msg)**
   * Stores the latest map data from the /map topic.
   * Input: An OccupancyGrid message containing the SLAM map.
   * Output: Updates the global current\_map.
2. **odom\_callback(msg)**
   * Extracts the robot's position and orientation from the /odom topic.
   * Input: An Odometry message.
   * Output: Robot's position (x, y) and orientation (yaw in radians).

**Helper Functions**

1. **heuristic(a, b)**
   * Calculates the Euclidean distance between two points. Used in the A\* algorithm to estimate the cost from the current node to the goal.
   * Input: Two tuples a and b, representing coordinates.
   * Output: Euclidean distance.
2. **plan\_path(start, goal, grid\_map)**
   * Implements the A\* algorithm for path planning.
   * Converts start and goal positions from world coordinates to grid coordinates.
   * Uses a priority queue (open\_list) to explore nodes with the lowest total cost (path cost + heuristic).
   * Checks for obstacles and reconstructs the path once the goal is reached.
   * Returns the next step in world coordinates.
   * Inputs:
     + start: Robot's current position (x, y) in world coordinates.
     + goal: Target position (x, y) in world coordinates.
     + grid\_map: SLAM map as an OccupancyGrid.
   * Outputs: Next position in world coordinates.
3. **compute\_control(robot\_position, target\_position)**
   * Computes the control commands (linear and angular velocities) to move the robot toward the target.
   * Uses proportional control:
     + Linear velocity is proportional to the distance to the target.
     + Angular velocity is proportional to the angular error (difference between current and target orientation).
   * Inputs:
     + robot\_position: Current robot position (x, y, yaw).
     + target\_position: Next position on the path (x, y).
   * Outputs:
     + Linear velocity (linear\_velocity).
     + Angular velocity (angular\_velocity).

**Main Function**

1. **Node Initialization**
   * Initializes the ROS node pipeline\_navigation\_slam.
2. **Publishers and Subscribers**
   * Subscribes to:
     + /map: Receives SLAM map updates.
     + /odom: Receives odometry data for robot position and orientation.
   * Publishes to:
     + /cmd\_vel: Sends velocity commands to the robot.
3. **Main Loop**
   * Runs at 10 Hz.
   * Waits for a valid SLAM map.
   * Gets the robot's current position via odom\_callback.
   * Plans the next step toward the target using plan\_path.
   * Computes control commands to move toward the next step using compute\_control.
   * Publishes the velocity commands.

**Key Algorithms**

***A Pathfinding*\***

* **Grid Conversion**:
  + Converts world coordinates to grid indices using the map's resolution and origin.
* **Node Exploration**:
  + Uses a priority queue to explore nodes with the lowest cost.
  + Costs are updated based on the current path and the heuristic to the goal.
* **Path Reconstruction**:
  + Backtracks from the goal to the start to reconstruct the optimal path.

**Proportional Control**

* Adjusts the robot's velocities to minimize the distance and angular error to the target.
* Ensures smooth navigation by limiting linear velocity when angular error is large.

**Execution Workflow**

1. Start the ROS node.
2. Wait for SLAM map updates.
3. Continuously:
   * Get the robot's current position from odometry.
   * Plan the optimal path to the target using the latest map.
   * Compute and publish control commands to follow the path.