Autonomous Golf Cart Project

AMR EHAB

HAZEM ASHRAF

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Purpose and Objectives . . . 1
- 1.3 Key Features . . . 1

2 System Architecture **3**
- 2.1 Introduction . . . 3
- 2.2 Hardware Components . . . 3
- 2.3 Software Components . . . 3
- 2.4 Data Flow and Processes . . . 4
  - 2.4.1 GPS Data Sharing . . . 4
  - 2.4.2 Sensor Data Processing . . . 4
  - 2.4.3 LiDAR Data Processing . . . 4
  - 2.4.4 Navigation . . . 5
  - 2.4.5 Emergency Stop Mechanism . . . 5

3 Data Flow and Processes **6**
- 3.1 Introduction . . . 6
- 3.2 GPS Data Sharing . . . 6
- 3.3 Sensor Data Processing . . . 7
- 3.4 LiDAR Data Processing . . . 7
- 3.5 Navigation . . . 8
- 3.6 Emergency Stop Mechanism . . . 8

4 Software **10**
- 4.1 NAV2 . . . 10
  - 4.1.1 AMCL (Adaptive Monte Carlo Localization) . . . 10
  - 4.1.2 MPPI Controller . . . 12
  - 4.1.3 Hybrid A* Planner . . . 15
  - 4.1.4 Local and Global Costmaps . . . 18
  - 4.1.5 Behavior Trees in Nav2 . . . 23
  - 4.1.6 SLAM using SLAM Toolbox . . . 27
- 4.2 Python Scripts . . . 29
  - 4.2.1 Odometry.py . . . 29

4.2.2 Publish_Goal.py 31
4.2.3 Publish_goal_exact_xy.py 34
4.2.4 Publish_gps_flask.py 35
4.2.5 publish_gps_telegram.py 37
4.2.6 publish_initial_pose.py 38
4.2.7 simulate_scan.py 40
4.2.8 send_commands.py 42
4.2.9 stop_car_flask.py 46
4.2.10 stop_car_telegram.py 47
4.3 Arduino Sketches 49
4.3.1 sensor_data 49
4.3.2 car_control 52

Chapter 1

Introduction

1.1 Overview

The Autonomous Golf Cart project aims to develop a self-driving golf cart capable of navigating a university campus autonomously. Leveraging advanced sensors and sophisticated algorithms, the golf cart can efficiently reach specified destinations while avoiding obstacles, providing a reliable and robust navigation solution.

1.2 Purpose and Objectives

The primary goal of this project is to create a fully autonomous golf cart that can:

- Navigate a pre-mapped university campus using Simultaneous Localization and Mapping (SLAM) techniques.
- Receive target GPS locations via a Telegram bot and navigate to those destinations.
- Avoid obstacles and ensure safe and efficient travel across the campus.

1.3 Key Features

- **Arduino Integration**: Utilizes two Arduino boards to control the golf cart and manage sensor data.
  - One Arduino is responsible for sending velocity and steering angle commands to the golf cart.
  - The second Arduino handles sensor integration, including IMU and proximity sensors for odometry.

- **Odometry Calculation**: A Python script (odometry.py) processes sensor data to compute odometry, providing accurate position tracking.
- **LiDAR Processing**: Employs the Velodyne package to gather 3D point clouds, which are converted into 2D laser scans using the pointcloud_to_laserscan package.
- **SLAM for Mapping**: Creates a detailed map of the university campus using the slam_toolbox package, combining scan and odometry data.
- **Navigation System**: Integrates the nav2 stack with MPPI controller and Hybrid A* planner to facilitate precise navigation and path planning.
- **Initial Position Estimation**: Utilizes GPS and magnetometer data to estimate the initial position and heading, refined by the ICP algorithm for accuracy.
- **Goal Positioning**: Receives GPS coordinates via Telegram and converts them to predefined map coordinates, setting navigation goals accordingly.
- **Steering Control**: Translates cmd_vel messages from nav2 into steering and velocity commands for the Arduino.
- **Emergency Stop**: Incorporates an emergency stop mechanism that can be triggered via a phone to halt the golf cart immediately.
- **GPS Integration**: Enhances odometry with GPS data received from a phone, converting GPS coordinates to map coordinates for accurate positioning.

This project demonstrates the integration of various technologies to achieve autonomous navigation, showcasing the potential for self-driving vehicles in controlled environments like a university campus.

Chapter 2

System Architecture

2.1 Introduction

The Autonomous Golf Cart project is built upon a robust and modular system architecture, combining hardware and software components to achieve seamless autonomous navigation. The following sections provide an overview of the high-level system architecture, detailing the key components and their interactions.

2.2 Hardware Components

- **Arduino Mega**: Connects to various sensors including IMU and proximity sensors and is responsible for publishing odometry data.
- **Arduino Uno**: Controls the golf cart by sending velocity and steering angle commands.
- **Sensors**:
  - **IMU (Inertial Measurement Unit)**: Provides orientation and acceleration data.
  - **Proximity Sensors**: Act as wheel encoders to measure the distance traveled.
  - **LiDAR**: Gathers 3D point cloud data for environment perception and mapping.

2.3 Software Components

- **ROS2 Humble**: The primary robotics framework used for managing communication between nodes and handling the navigation stack.

- **Python Scripts**: Custom scripts developed for processing sensor data, estimating positions, and controlling the golf cart.
  - `odometry.py`
  - `publish_initial_pose.py`
  - `publish_goal.py`
  - `send_commands.py`
- **Arduino Sketches**: Code uploaded to Arduino boards to interface with sensors and control the golf cart's movement.

2.4 Data Flow and Processes

The system operates through a series of interconnected processes that facilitate data collection, processing, and action execution. The key processes are:

2.4.1 GPS Data Sharing

- `publish_gps_flask.py`: Publishes GPS data from a phone to the `/fix` topic.
- `gps_to_odom.py`: Converts GPS data to odometry and publishes it to the `/odometry/gps` topic.

2.4.2 Sensor Data Processing

- `odometry.py`: Gathers IMU and proximity sensor data, calculates odometry, and publishes it to the `/odom` topic.
- `publish_initial_pose.py`: Estimates the initial pose using GPS and magnetometer data and publishes the initial pose to the `/initialpose` topic.
- `ekf_node`: Fuses sensor data to improve odometry accuracy and publishes the result to the `/odometry` topic.

2.4.3 LiDAR Data Processing

- `pointcloud_to_laserscan`: Converts 3D point clouds to 2D laser scans and publishes them to the `/scan` topic.
- `slam_toolbox`: Utilizes the odometry and scan data to create a map of the environment, publishing the map to the `/map` topic.

2.4.4 Navigation

- `nav2`: Manages the navigation stack including localization (AMCL), path planning (Hybrid A*), and control (MPPI controller). Key topics include `/odom`, `/scan`, `/map`, and `/cmd_vel`.
- `publish_goal.py`: Receives target GPS coordinates via Telegram, converts them to map coordinates, and sets navigation goals by publishing to the `/goal` topic.
- `send_commands.py`: Translates navigation commands (`cmd_vel` messages) into velocity and steering commands for the Arduino.

2.4.5 Emergency Stop Mechanism

- Monitors the `/stop_car` topic and can be triggered via scripts (`stop_car_telegram.py`, `stop_car_flask.py`) to immediately halt the golf cart.

This architecture ensures a cohesive and efficient operation of the autonomous golf cart, leveraging the strengths of each component to achieve reliable navigation and obstacle avoidance.

Chapter 3

Data Flow and Processes

3.1 Introduction

The Autonomous Golf Cart project involves several interconnected processes that facilitate data collection, processing, and execution of navigation commands. The following subsections describe the key data flow and processes in detail

3.2 GPS Data Sharing

- `publish_gps_flask.py`
  - **Functionality**: Publishes GPS data from a phone to the ROS2 network.
  - **Topic**: `/fix`
  - **Description**: This script runs on a Flask server that receives GPS data from a phone and publishes it to the `/fix` topic. This data is crucial for estimating the initial position of the golf cart.
- `gps_to_odom.py`
  - **Functionality**: Converts GPS data to odometry.
  - **Topic**: `/odometry/gps`
  - **Description**: This script takes the GPS data published on the `/fix` topic and converts it into odometry data, publishing the result to the `/odometry/gps` topic. This conversion is necessary for integrating GPS data into the SLAM and navigation processes.

3.3 Sensor Data Processing

- `odometry.py`
  - **Functionality**: Calculates odometry from IMU and proximity sensor data.
  - **Topic**: `/odom`
  - **Description**: This script processes data from the IMU and proximity sensors to calculate odometry, publishing the resulting odometry data to the `/odom` topic. Accurate odometry is essential for reliable localization and mapping.
- `publish_initial_pose.py`
  - **Functionality**: Estimates and publishes the initial pose of the golf cart.
  - **Topic**: `/initialpose`
  - **Description**: This script uses GPS and magnetometer data to estimate the initial position and heading of the golf cart, publishing the initial pose to the `/initialpose` topic. This information is used to initialize the SLAM and navigation systems.
- `ekf_node`
  - **Functionality**: Fuses sensor data for improved odometry.
  - **Topic**: `/odometry`
  - **Description**: This node implements an Extended Kalman Filter (EKF) to fuse data from various sensors, providing a more accurate estimate of the golf cart's odometry. The fused odometry data is published to the `/odometry` topic.

3.4 LiDAR Data Processing

- `pointcloud_to_laserscan`
  - **Functionality**: Converts 3D point clouds to 2D laser scans.
  - **Topic**: `/scan`
  - **Description**: This package takes the 3D point cloud data from the LiDAR and converts it into 2D laser scan data, publishing the scans to the `/scan` topic. This data is used for both mapping and obstacle detection.
- `slam_toolbox`
  - **Functionality**: Creates and publishes a map of the environment.
  - **Topic**: `/map`

  - **Description**: This package combines odometry and laser scan data to create a map of the environment, publishing the map to the `/map` topic. The map is used for navigation and obstacle avoidance.

3.5 Navigation

- `nav2`
  - **Functionality**: Manages the navigation stack.
  - **Topics**: `/odom`, `/scan`, `/map`, `/cmd_vel`
  - **Description**: This package includes the Adaptive Monte Carlo Localization (AMCL) for localization, Hybrid A* planner for path planning, and MPPI controller for control. It uses odometry, laser scan, and map data to navigate the golf cart to the target destinations, publishing velocity commands to the `/cmd_vel` topic.
- `publish_goal.py`
  - **Functionality**: Receives and publishes navigation goals.
  - **Topic**: `/goal`
  - **Description**: This script receives GPS coordinates via Telegram, converts them to map coordinates, and sets navigation goals by publishing to the `/goal` topic. This enables the golf cart to navigate to specified locations autonomously.
- `send_commands.py`
  - **Functionality**: Translates and sends navigation commands to Arduino.
  - **Topic**:`/cmd_vel`
  - **Description**: This script takes velocity commands `cmd_vel` messages from the navigation stack and translates them into velocity and steering angle commands for the Arduino. This ensures that the golf cart follows the planned path accurately.

3.6 Emergency Stop Mechanism

- **Functionality**: Provides an emergency stop feature to halt the golf cart immediately.
- **Topic**:`/stop_car`

- **Description**: : The emergency stop mechanism listens to the `/stop_car` topic. If a stop signal is received, either from the `stop_car_telegram.py`or `stop_car_flask.py` script, the golf cart will halt immediately, ensuring safety in case of unexpected obstacles or emergencies.

This detailed breakdown of the data flow and processes provides a comprehensive understanding of how the Autonomous Golf Cart system operates, ensuring reliable navigation and obstacle avoidance.

Chapter 4

Software

4.1 NAV2

4.1.1 AMCL (Adaptive Monte Carlo Localization)

Overview

Adaptive Monte Carlo Localization (AMCL) is a probabilistic algorithm used for estimating the pose of a robot in a known map. It utilizes a particle filter to maintain and update a set of hypotheses (particles) about the robot's position and orientation.

How AMCL Works

1. **Initialization:**
    - The algorithm starts by initializing a set of particles. Each particle represents a possible pose of the robot (x, y, yaw). The initial distribution of particles can be uniform or based on an initial guess if available.
2. **Motion Update (Prediction):**
    - When the robot moves, the particles are updated based on the robot's motion model. The motion model predicts the new state of each particle by adding random noise to simulate real-world uncertainties. This step is often referred to as the prediction step.
    - The parameters α_1 to α_5 control the noise added during this step, representing different types of errors such as rotational and translational errors.
3. **Sensor Update (Correction):**

- After moving, the robot uses sensor data to update the particles. Each particle is weighted based on how well the predicted pose matches the observed sensor data.
- For example, if the robot uses a laser scanner, the laser readings are compared against the map to determine the likelihood of each particle. Particles that better match the sensor data receive higher weights.
- Parameters like `laser_model_type`, `max_beams`, `laser_likelihood_max_dist`, and others influence how sensor data is interpreted and matched to the map.

4. **Resampling:**
   - To focus computational resources on more likely hypotheses, a resampling step is performed. Particles with higher weights are more likely to be selected, and particles with low weights are discarded. This step helps in concentrating the particles around the probable locations.
   - The `resample_interval` parameter controls how frequently resampling occurs, and the `min_particles` and `max_particles` parameters define the range of particles used.
5. **Beam Skipping (Optional):**
   - Beam skipping is a technique to improve efficiency by skipping beams in laser scans that are likely to be outliers or less informative. Parameters like `do_beamskip`, `beam_skip_distance`, `beam_skip_error_threshold`, and `beam_skip_threshol` control this feature.
6. **Convergence and Pose Estimation:**
   - Over time, the particles converge around the true pose of the robot. The pose with the highest weight or the mean of the particles is used as the estimated pose.
   - Parameters like `pf_err` and `pf_z` influence the convergence and the distribution of particles.

Key Parameters and Their Roles

- α_1 to α_5: Control noise added to particles during motion updates.
- `base_frame_id`, `odom_frame_id`, `global_frame_id`: Frame IDs used for transformations.
- `laser_model_type`: Determines the model used for laser-based sensor updates.
- `max_beams`: Maximum number of beams to consider from each laser scan.
- `max_particles`, `min_particles`: Limits the number of particles used.

- `resample_interval`: Interval at which resampling is performed.
- `laser_likelihood_max_dist`: Maximum distance for considering laser likelihood.
- `transform_tolerance`: Tolerance for looking up transforms.
- `update_min_a`, `update_min_d`: Minimum angle and distance changes required to trigger an update.
- `beam_skip_*`: Parameters for beam skipping to enhance performance.

Benefits and Applications

AMCL is widely used in robotics for its ability to provide accurate localization in dynamic and complex environments. Its probabilistic nature allows it to handle uncertainties and provide robust pose estimation, making it suitable for various applications, including autonomous navigation, mapping, and mobile robot operations.

Conclusion

AMCL is a critical component in the navigation stack, enabling robots to localize themselves accurately within a known map. By leveraging a particle filter and incorporating motion and sensor updates, AMCL ensures reliable and precise pose estimation, essential for effective autonomous navigation.

For more detailed information, you can refer to the Nav2 AMCL documentation

4.1.2 MPPI Controller

Overview

The Model Predictive Path Integral (MPPI) controller is a sophisticated control strategy designed for dynamic path tracking and obstacle avoidance. It evaluates multiple candidate trajectories and selects the optimal one based on a cost function, taking into account various dynamic and kinematic constraints.

Key Concepts and Parameters

1. **Trajectory Sampling:**
    - **Time Steps (`time_steps`):** Defines the prediction horizon of the controller. For example, a setting of 56 means the controller predicts the robot's behavior over 56 discrete time intervals.

- **Model Time Delta (`model_dt`):** Specifies the duration of each time step, set to 0.1 seconds in your configuration. This influences the granularity of the predictions.
- **Batch Size (`batch_size`):** The number of trajectories sampled per iteration, set to 2000. Larger batch sizes can improve robustness but increase computational load.

2. **Path Handling:**

   - **Max Robot Pose Search Distance (`max_robot_pose_search_dist`):** This parameter defines how far ahead of the robot the controller searches for the nearest path point. This is critical for maintaining path accuracy.
   - **Prune Distance (`prune_distance`):** Set to 3.0 meters, it specifies the distance ahead of the nearest path point to prune the path, removing unnecessary path segments the robot has already passed.
   - **Transform Tolerance (`transform_tolerance`):** With a value of 1.0 second, this parameter sets the allowable time difference for coordinate frame transformations.

3. **Critic Functions:** Critic functions are essential to the MPPI controller as they evaluate trajectories based on various criteria. Your configuration includes several specific critics:

   - **Constraint Critic:**
     - **Cost Weight (`cost_weight`):** Set to 4.0, determining the influence of this critic on the overall cost.
     - **Cost Power (`cost_power`):** Set to 1, specifying the exponent applied to the cost.
   - **Goal Critic:**
     - **Cost Weight (`cost_weight`):** Set to 5.0, emphasizing the importance of reaching the goal.
     - **Threshold to Consider (`threshold_to_consider`):** At 1.4 meters, this sets the distance from the goal where the cost starts being significant.
   - **Goal Angle Critic:**
     - **Cost Weight:** 3.0, influencing the penalty for deviating from the goal direction.
     - **Threshold to Consider (`threshold_to_consider`):** 0.5 meters.
   - **Prefer Forward Critic:**
     - **Cost Weight (`cost_weight`):** 5.0, favoring forward motion.
     - **Threshold to Consider (`threshold_to_consider`):** 0.5 meters.

- **Obstacles Critic:**
  - **Consider Footprint (`consider_footprint`):** Set to true, this considers the robot's footprint for obstacle costs.
- **Cost Critic:**
  - **Collision Cost (`collision_cost`):** Set to 1000000.0, imposing a high penalty for collisions.
- **Path Align Critic:**
  - **Cost Weight (`cost_weight`):** 14.0, penalizing misalignment with the path.
- **Path Follow Critic:**
  - **Cost Weight (`cost_weight`):** 5.0, ensuring the robot follows the path accurately.
- **Path Angle Critic:**
  - **Cost Weight (`cost_weight`):** 2.0, assessing angular alignment with the path.

4. **Motion Model :**

- **Ackermann Motion Model:**
  - **Minimum Turning Radius (`min_turning_r`):** Set to 5.0 meters, this defines the smallest radius for the vehicle's turns, crucial for Ackermann steering.

5. **Velocity Constraints:**

- **Maximum Velocity (`vx_max`):** Set to 2.5 m/s, this is the upper limit of the robot's speed.
- **Maximum Angular Velocity (`wz_max`):** Set to 1.9 rad/s, controlling maximum rotation speed.
- **Minimum Velocity (`vx_min`):** Set to -0.35 m/s, allowing for backward motion when needed.

Practical Implications

- Granularity and Accuracy: Higher time_steps and smaller model_dt provide finer trajectory predictions, enhancing navigation accuracy but increasing computational demand.
- Obstacle Avoidance: : The Obstacles Critic and Cost Critic ensure effective obstacle avoidance by imposing high costs on collision paths.

- Path Following: Critics like Path Align Critic and Path Follow Critic ensure accurate path tracking, improving navigation reliability.
- Adaptive Motion: The Prefer Forward Critic and configured velocity constraints promote natural driving behavior, favoring forward motion.

These configurations enable the MPPI controller to balance precision in trajectory planning with computational efficiency, suitable for navigating complex and dynamic environments effectively.

For more detailed information, you can refer to Nav2 MPPI Controller Documentation (GitHub).

4.1.3 Hybrid A* Planner

The Hybrid A* planner, part of the Smac Planner suite in Nav2, is designed to generate feasible paths for robots with kinematic constraints, such as car-like robots. It combines grid-based search with continuous-space planning, ensuring the paths generated are both feasible and optimal for the robot's movement capabilities.

How the Hybrid A* Planner Works

1. **Graph Search:**
   - The Hybrid A* planner uses a templated A* implementation to search a graph of nodes. Each node represents a possible state of the robot, defined by its position (x, y) and orientation θ.
   - The planner expands nodes by simulating the robot's movement using motion models like Dubins (for forward-only paths) or Reeds-Shepp (for paths allowing reverse motion). These models generate feasible trajectories that the robot can follow.
2. **Motion Models:**
   - Dubins Model: Suitable for vehicles that can only move forward, generating smooth paths with a constrained turning radius.
   - Reeds-Shepp Model: Allows for reversing, providing more flexible paths that can handle tight spaces and complex maneuvers.
3. **Analytic Expansion:**
   - The planner attempts shortcuts during the path search to reduce overall path length. This involves dynamically generating graph nodes to find more efficient routes, improving both planning speed and path quality..

Key Parameters and Their Roles

1. **Tolerance (tolerance)**:
   - **Value**: 0.5 meters
   - **Description**: Sets the acceptable deviation from the exact goal pose. This flexibility allows the planner to consider a solution valid even if it slightly misses the target, reducing planning time.
2. **Downsampling Costmap (downsample_costmap)**:
   - **Value**: True
   - **Description**: Improves planning speed by reducing the resolution of the costmap. Downsampling helps in large maps or when rapid replanning is required by simplifying the environment representation.
3. **Allow Unknown Space (allow_unknown)**:
   - **Value**: False
   - **Description**: Determines if the planner can navigate through unknown areas. Setting this to false avoids unpredictable regions, ensuring the robot only navigates known and mapped areas.
4. **Maximum Iterations (max_iterations)**:
   - **Value**: 5000000
   - **Description**: Limits the number of iterations the search algorithm can perform. This ensures a thorough search while preventing excessive computation time, balancing between path quality and efficiency.
5. **Minimum Turning Radius (minimum_turning_radius)**:
   - **Value**: 4.25 meters
   - **Description**: Defines the smallest possible turn the vehicle can make. This parameter is crucial for planning realistic paths for vehicles with limited steering capabilities, ensuring the robot can physically follow the generated path.
6. Penalties
   - **Reverse Penalty (reverse_penalty)**:
     - **Value**: 0.9
     - **Description**: Applies a penalty for reversing, encouraging the planner to prefer forward motion unless reversing is necessary. This is more relevant for the Reeds-Shepp model.

- **Change Penalty (change_penalty)**:
  - **Value**: 5.0
  - **Description**: Applies a penalty for changing directions, such as switching from forward to backward. This encourages smoother paths with fewer direction changes.
- **Non-Straight Penalty (non_straight_penalty)**:
  - **Value**: 3.0
  - **Description**: Penalizes non-straight segments of the path, promoting straighter and smoother paths.
- **Cost Penalty (cost_penalty)**:
  - **Value**: 30.0
  - **Description**: Adds a penalty based on the costmap values, steering the robot away from high-cost areas.
- **Retrospective Penalty (retrospective_penalty)**:
  - **Value**: 0.025
  - **Description**: Prefers maneuvers later in the path, which can save search time and reduce unnecessary early maneuvers.
- **Rotation Penalty (rotation_penalty)**:
  - **Value**: 5.0
  - **Description**: Penalizes rotations in place, encouraging the planner to find paths that minimize such maneuvers.

7. Path Smoothing

- **Smoother Integration**: After generating the initial path, a smoothing process refines it further. The smoother ensures the final trajectory is smooth, feasible, and optimized for the robot's motion model.
- **Smoothing Parameters**:
  - **Maximum Iterations (max_iterations)**: Set to 1000, limiting the number of iterations the smoother can perform to ensure it completes in a reasonable time.
  - **Tolerance (tolerance)**: Defined as 1.0×10^{-10}, this sets the level of precision for the smoother, balancing path quality and computational load.

8. Collision Checking

- **Footprint Consideration**: Uses the robot's footprint to check for collisions along the planned path. This ensures the robot can navigate without hitting obstacles, even in tight spaces.

Practical Implications

- **Efficiency**: The downsampling of the costmap and high iteration limits enable efficient and comprehensive path planning, suitable for dynamic and complex environments.
- **Accuracy**: The combination of grid-based search and continuous-space planning ensures that the generated paths are precise and feasible, accommodating the robot's kinematic constraints.
- **Adaptability**: The planner's ability to handle different motion models and integrate costmap data makes it adaptable to various robotic platforms and environmental conditions.

These configurations enable the Hybrid A* planner to generate accurate, feasible, and smooth paths, making it a powerful tool for navigating complex and dynamic environments.

For more detailed information, you can refer to Nav2 smac planner Documentation (GitHub).

4.1.4 Local and Global Costmaps

Costmaps are essential components in the Nav2 framework, providing a grid-based representation of the environment around the robot. They integrate sensor data to identify obstacles and free space, crucial for safe navigation. There are two primary types of costmaps: local and global.

Local Costmap

The local costmap represents the immediate surroundings of the robot, updating frequently based on real-time sensor data. It is used for short-term planning and obstacle avoidance.

1. Key Parameters and Their Roles
    - **Update Frequency (update_frequency)**:
        - **Value**: 10.0 Hz
        - **Description**: Determines how often the local costmap is updated. Higher frequencies ensure the costmap remains up-to-date with the latest sensor data, crucial for real-time obstacle avoidance.

- **Publish Frequency (publish_frequency)**:
  - **Value**: 10.0 Hz
  - **Description**: Sets how often the costmap is published. This is important for other nodes that rely on the costmap for planning and control.
- **Global Frame (global_frame)**:
  - **Value**: odom
  - **Description**: The coordinate frame in which the costmap is maintained. The "odom" frame is typically used for local costmaps to represent the robot's immediate surroundings accurately.
- **Robot Base Frame (robot_base_frame)**:
  - **Value**: base_link
  - **Description**: The frame attached to the robot's base. It is the reference point for the costmap.
- **Rolling Window (rolling_window)**:
  - **Value**: True
  - **Description**: Enables a rolling window approach, where the costmap moves with the robot, always representing the area around it.
- **Dimensions (width and height)**:
  - **Value**: 25 meters (width), 10 meters (height)
  - **Description**: Define the size of the local costmap. A larger costmap covers more area but requires more computational resources.
- **Resolution (resolution)**:
  - **Value**: 0.05 meters
  - **Description**: Specifies the size of each cell in the costmap. Higher resolution provides more detail but increases computational load.
- **Footprint (footprint)**:
  - **Value**:Custom polygon
  - **Description**: Defines the shape and size of the robot in the costmap for collision checking.
- **Plugins**:
  - Obstacle Layer (obstacle_layer): Detects obstacles using sensor data (e.g., from a laser scanner).
  - Inflation Layer (inflation_layer): Inflates obstacles to account for the robot's size and safety buffer.
  - Keepout Filter (keepout_filter): Defines areas where the robot is not allowed to navigate.
- **Inflation Layer Parameters**:

  - Cost Scaling Factor (cost_scaling_factor):
    * Value: 1.0
    * Description: Determines how quickly the cost values decrease with distance from an obstacle.
  - Inflation Radius (inflation_radius)
    * Value: 2.2 meters
    * Description: The radius around each obstacle that is inflated to ensure the robot maintains a safe distance.
- **Obstacle Layer Parameters:**:
  - Observation Sources (observation_sources): Specifies the sensors used for obstacle detection
  - Obstacle Max Range (obstacle_max_range):
    * Value: 25.0 meters
    * Description: The maximum range at which obstacles are considered. Obstacles beyond this range are ignored to optimize performance.
  - Raytrace Max Range (raytrace_max_range):
    * Value: 100.0 meters
    * Description: The maximum range for ray tracing operations, which are used to clear space in the costmap by tracing from the robot to known obstacles.
  - Inf is Valid (inf_is_valid):
    * Value: True
    * Description: Indicates whether infinite ranges in sensor data (representing no return) are considered valid. This helps in marking and clearing operations in the costmap.

2. Practical Implications

- **Real-Time Obstacle Avoidance**: Frequent updates ensure the robot can react swiftly to new obstacles.
- Safety: The inflation layer provides a safety buffer around obstacles.
- Flexibility: Rolling window and adjustable dimensions make the local costmap adaptable to different environments and robot sizes.

Global Costmap

The global costmap represents the entire navigable space and is used for long-term path planning. It is static or updated less frequently than the local costmap and incorporates more comprehensive map data.

1. Key Parameters and Their Roles

- **Update Frequency (update_frequency)**:
  - **Value**: 10.0 Hz
  - **Description**: Determines how often the global costmap is updated. Although typically less frequent than the local costmap, it ensures the map stays current with significant changes.
- **Publish Frequency (publish_frequency)**:
  - **Value**: 10.0 Hz
  - **Description**: Controls how often the costmap is published for use by other nodes.
- **Global Frame (global_frame)**:
  - **Value**: map
  - **Description**: The coordinate frame for the global costmap, often a static reference frame for the entire environment.
- **Robot Base Frame (robot_base_frame)**:
  - **Value**: base_link
  - **Description**: The frame attached to the robot's base, serving as the reference point for the costmap.
- **Resolution (resolution)**:
  - **Value**: 0.05 meters
  - **Description**: Specifies the granularity of the costmap. Higher resolution provides more detailed representation but increases computational load.
- **Footprint (footprint)**:
  - **Value**:Custom polygon
  - **Description**: Defines the robot's shape for accurate collision checking in the costmap.
- **Plugins**:
  - Static Layer (static_layer): Represents the static map of the environment.
  - Obstacle Layer (obstacle_layer):Adds dynamic obstacles detected by sensors.
  - Inflation Layer (inflation_layer): Adds safety buffers around obstacles.
  - Speed Filter (speed_filter): Modulates speed based on costmap information.
  - Keepout Filter (keepout_filter): Defines no-go zones.
- **Inflation Layer Parameters**:
  - Cost Scaling Factor (cost_scaling_factor):
    - Value: 1.0
    - Description: Controls the rate at which cost values decrease with distance from an obstacle.

  - Inflation Radius (inflation_radius)
    * Value: 2.2 meters
    * Description: The Defines the radius around obstacles that is inflated to create a buffer zone.

- **Obstacle Layer Parameters:**:
  - Observation Sources (observation_sources): Lists sensors used to detect obstacles.
  - Obstacle Max Range (obstacle_max_range):
    * Value: 25.0 meters
    * Description: The maximum range at which obstacles are considered. Obstacles beyond this range are ignored to optimize performance.
  - Raytrace Max Range (raytrace_max_range):
    * Value: 100.0 meters
    * Description: The maximum range for ray tracing operations, used to clear space in the costmap by tracing from the robot to known obstacles.
  - Inf is Valid (inf_is_valid):
    * Value: True
    * Description: Indicates whether infinite ranges in sensor data (representing no return) are considered valid. This helps in marking and clearing operations in the costmap.

2. Practical Implications

- Comprehensive Path Planning: The global costmap provides a broad view of the environment, enabling strategic path planning over longer distances.
- Adaptability: Plugins like the speed filter and keepout filter enhance the robot's ability to navigate safely and efficiently.
- Safety and Efficiency: The inflation layer and footprint considerations ensure safe navigation by maintaining buffers around obstacles.

conclusion

The local and global costmaps work in tandem to provide the robot with a detailed and dynamic understanding of its environment. The local costmap focuses on immediate obstacle avoidance and real-time updates, while the global costmap is used for strategic path planning across the entire environment. Proper configuration of these costmaps ensures efficient, safe, and reliable navigation.

For more detailed information, you can refer to Nav2 Costmaps documentation.

4.1.5 Behavior Trees in Nav2

Behavior Trees (BTs) are a modular and flexible way to define complex robot behaviors, making them a crucial part of the Nav2 navigation framework. They offer a clear and structured method for decision-making and task execution, allowing for dynamic reconfiguration and runtime adjustments.

How Behavior Trees Work

1. **Structure and Nodes:**
    - **Tree Structure:** Behavior Trees are structured hierarchically, starting with a root node and branching out into various sub-nodes. Each node represents a specific task or condition.
    - **Node Types:** There are different types of nodes in BTs:
        - **Control Nodes:** Direct the flow of execution (e.g., Sequence, Selector).
        - **Decorator Nodes:** Modify the behavior of other nodes (e.g., Retry, Inverter).
        - **Action Nodes:** Perform specific actions (e.g., NavigateToPose, FollowPath).
        - **Condition Nodes:** Check conditions to decide the flow (e.g., IsBatteryLow).
2. **Execution Flow:**
    - Ticking: Nodes are "ticked" periodically, determining whether they should execute, continue, or terminate. This ticking mechanism allows BTs to handle asynchronous tasks and adapt to changing conditions dynamically.
3. **Modularity and Reusability:**
    - BTs promote modularity by allowing developers to create reusable nodes and sub-trees. This modularity simplifies the design of complex behaviors and makes the system easier to maintain and extend.

Key Parameters and Their Roles

1. **Behavior Tree XML Files:**
   - **Default Navigate To Pose BT XML (default_nav_to_pose_bt_xml):**
     - Value: Path to the XML file defining the behavior tree for navigating to a pose.
     - Description: Specifies the behavior tree used for the NavigateToPose action, including all the conditions, actions, and sequences needed for this task.
   - **Default Navigate Through Poses BT XML (default_nav_through_poses_bt_xml):**
     - Value: Path to the XML file for navigating through a series of poses.
     - Description: Defines the behavior tree for the NavigateThroughPoses action, handling the robot's movement through multiple waypoints.
2. **Plugin Libraries (plugin_lib_names):**
   - Description: Lists the shared libraries containing custom BT nodes. These plugins extend the functionality of the BTs by providing specialized nodes for specific tasks.
3. Behavior Tree Nodes:
   - Navigation Actions:
     - NavigateToPose: Commands the robot to navigate to a specific pose.
     - NavigateThroughPoses: Directs the robot to move through a sequence of waypoints.
     - FollowPath: Executes path-following behavior based on a planned path.
   - Conditions and Control:
     - IsBatteryLow: Checks if the robot's battery level is below a certain threshold.
     - GoalReached: Evaluates if the robot has reached its goal.
     - ReinitializeGlobalLocalization: Reinitializes the robot's global localization system.
     - ClearCostmap: Clears obstacles from the costmap to handle dynamic changes in the environment.

4. BT Navigator Parameters:

   - BT Loop Duration (bt_loop_duration):
     - Value: 10 seconds
     - Description: Specifies the duration of each behavior tree loop. This defines how often the tree is ticked, ensuring timely decision-making and task execution.
   - Default Server Timeout (default_server_timeout):
     - Value: 20 seconds
     - Description: Sets the timeout for the behavior tree actions, ensuring the system does not get stuck on a task indefinitely.

5. Custom Nodes:

   - Nav2 Specific Nodes:
     - nav2_compute_path_to_pose_action_bt_node: Computes the path to a specified pose.
     - nav2_compute_path_through_poses_action_bt_node: Calculates a path through a series of poses.
     - nav2_smooth_path_action_bt_node: Smooths the computed path for better execution.
     - nav2_follow_path_action_bt_node: Follows the smoothed path.
     - nav2_clear_costmap_service_bt_node: Clears the costmap when needed.

Practical Implications

1. Modularity and Scalability:

   - The modular structure of BTs allows for easy expansion and modification of robot behaviors. Developers can add new nodes or sub-trees to introduce new capabilities without affecting the overall system.

2. Dynamic Reconfiguration:

   - BTs enable dynamic reconfiguration of tasks based on real-time conditions. This flexibility ensures that the robot can adapt to changes in its environment or mission objectives efficiently.

3. Robust Decision-Making:

   - By structuring tasks hierarchically and using condition nodes, BTs provide robust decision-making capabilities. The robot can evaluate multiple conditions and choose the best course of action dynamically.

4. Reusability
    - Nodes and sub-trees in BTs are reusable across different projects and tasks. This reusability reduces development time and ensures consistency in behavior across different applications.

Example Behavior Tree

An example behavior tree for navigating to a pose might include the following sequence of actions and conditions:

1. Check Battery Level: Use the IsBatteryLow condition node to ensure the battery is sufficient for the task.
2. Compute Path: Use the nav2_compute_path_to_pose_action_bt_node to calculate the path to the goal.
3. Clear Costmap: Use the nav2_clear_costmap_service_bt_node to clear the costmap if necessary.
4. Follow Path: Use the nav2_follow_path_action_bt_node to navigate along the computed path.
5. Reinitialize Localization: If the path following fails, use the nav2_reinitialize_ global_localization_service_bt_node to reset the localization.

Conclusion

Behavior Trees in Nav2 provide a powerful, flexible, and modular framework for defining complex robot behaviors. They enhance the robot's ability to make decisions and execute tasks dynamically, adapting to real-time conditions and changes in the environment. Proper configuration and utilization of BTs can significantly improve the robustness and efficiency of autonomous navigation systems.

For more detailed information, you can refer to Nav2 behavior tree documentation (GitHub).

4.1.6 SLAM using SLAM Toolbox

Overview

The SLAM Toolbox is a powerful set of tools for 2D SLAM (Simultaneous Localization and Mapping) designed to handle both lifelong and real-time mapping and localization. Developed by Steve Macenski, it integrates various advanced capabilities for mapping large spaces efficiently.

Key Features

1. **Mapping Modes:**
    - **Synchronous and Asynchronous Mapping:** Supports both synchronous mode (processing all scans in sequence) and asynchronous mode (processing scans as they arrive).
    - **Lifelong Mapping:** Continuously refines and updates maps over time, removing outdated information and integrating new data.
2. **Localization:**
    - **Pose-Graph Optimization:** Uses an optimization-based approach for localization, which can operate without a prior map in "lidar odometry" mode with local loop closures.
    - **Elastic Pose-Graph Localization:** Maintains a rolling buffer of recent scans for localization, providing robust performance without long-term map updates.
3. **Graph Manipulation:**
    - **Interactive Tools:** Provides RViz plugins for manual loop closures and map manipulations, allowing users to adjust the pose graph and correct errors interactively.
    - **Serialization and Deserialization:** Enables saving and loading of map data and pose graphs, facilitating continued mapping and map updates.

Configuration Parameters

1. **Mapping Parameters:**
    - **Mode:** Defines whether the toolbox operates in mapping or localization mode.
    - **Resolution:** Sets the resolution of the generated map.
    - **Map Save Path:** Specifies where the generated maps are saved.
2. **Solver Parameters:**

- **Solver Plugin:** The type of nonlinear solver to use for scan matching, with options like CeresSolver, SpaSolver, and G2oSolver.
- **Linear Solver:** Determines the linear solver used by Ceres (e.g., SPARSE_NORMAL_CHOL
- **Preconditioner:** Sets the preconditioner for the solver (e.g., JACOBI, SCHUR_JACOBI).
- **Trust Strategy:** Configures the trust region strategy for optimization.

3. **Optimization Settings:**

   - **Max Iterations:** Limits the number of iterations for optimization to balance accuracy and computational load.
   - **Loss Function:** Defines the loss function to handle outliers, with options like HuberLoss for robust performance in noisy environments.

4. **Graph Parameters:**

   - **Node Decay:** In lifelong mapping mode, old nodes can be removed to bound computational requirements.
   - **KD-Tree Search:** Uses KD-Tree for efficient pose estimation during reinitialization.

5. **RViz Plugin:**

   - **Interactive Mode:** Allows manipulation of graph nodes and edges in RViz.
   - **2D Pose Estimation Tool:** Provides a graphical interface for setting the initial pose or relocalizing the robot.

Practical Applications

1. **Large-Scale Mapping:**

   - Efficiently maps large environments, processing up to 200,000 square feet in synchronous mode and larger areas asynchronously.

2. **Lifelong Mapping and Localization:**

   - Suitable for dynamic environments where continuous updates and refinements to the map are necessary.

3. **Flexible Deployment:**

   - Supports both real-time mapping on live robots and offline map processing, making it versatile for different use cases.

Example Use Case

1. **Setup:**
   - Configure the SLAM Toolbox in synchronous mode for initial mapping.
   - Use RViz for visual feedback and manual adjustments.
2. **Mapping:**
   - Start mapping by collecting laser scans and odometry data.
   - Use the pose-graph to refine the robot's odometry and find loop closures.
3. **Localization:**
   - Switch to localization mode using the saved pose-graph.
   - Maintain a rolling buffer of recent scans for robust localization.

For more detailed information, you can refer to SLAM Toolbox GitHub page.

4.2 Python Scripts

4.2.1 Odometry.py

Algorithm Overview

1. Initialization (`main` function):
   - Initialize the ROS2 node (`OdometryNode`).
   - Start the ROS2 spin to keep the node running.
2. `OdometryNode` Class:
   - Inherits from `rclpy.node.Node` to create a ROS2 node for odometry.
   - Handles serial communication setup and data processing.
   - Manages publishers for IMU data (`imu/data_raw`), odometry data (`odom`), and magnetometer direction (`magnetometer/direction`).
   - Uses `tf2_ros.TransformBroadcaster` for publishing transforms.
   - Subscribes to `/cmd_vel` topic to receive velocity commands.
3. `Point` Class:
   - Represents a point in 2D space with x, y coordinates and a heading angle.
   - Has a method `move` to update the point's position based on linear and angular velocities over time (`dt`).

Equations Used

1. `Point` Class (`move` method):

   - heading

     $$\text{heading} += \text{angular\_velocity} \times dt$$

     Explanation
     - heading: Represents the current heading (or orientation) of the point.
     - angular_velocity: Indicates how fast the heading is changing over time.
     - dt: Time step or time interval.

   - Change in x-coordinate (`dx`):

     $$dx = \text{linear\_velocity} \cdot \cos(\text{heading}) \cdot dt$$

     Explanation: Calculates the change in the x-coordinate of a point based on its linear velocity and current heading angle over a time step dt.

   - Change in y-coordinate (`dy`):

     $$dy = \text{linear\_velocity} \cdot \sin(\text{heading}) \cdot dt$$

     Explanation: Calculates the change in the y-coordinate of a point based on its linear velocity and current heading angle over a time step dt.

2. `OdometryNode` Class (`read_data` method):

   - Orientation quaternion (`z`, `w` components):

     $$\text{orientation.z} = \sin\left(\frac{\text{heading}}{2}\right)$$

     $$\text{orientation.w} = \cos\left(\frac{\text{heading}}{2}\right)$$

     Explanation: Computes the quaternion components (`z` and `w`) representing the orientation of a robot based on its current heading angle. These components are used to describe the rotation in 3D space.

Functions and Their Roles

- `rclpy.init()` / `rclpy.shutdown()`:
  - Initialize and shutdown the ROS2 communication system.
- `Node.create_publisher()`:
  - Creates a publisher for publishing messages on specific topics.

- `Node.create_subscription()`:
  - Creates a subscriber for receiving messages from specific topics.
- `tf2_ros.TransformBroadcaster`:
  - Facilitates broadcasting of coordinate frame transforms.
- `serial.Serial()`:
  - Initializes serial communication object.
- `serial_comm.readline()`:
  - Reads a line from the serial port.
- `time.sleep()`:
  - Delays execution for a specified number of seconds.

Summary

- **Algorithm**: Integrates equations for calculating position updates (`dx`, `dy`) and orientation (quaternion components) based on sensor inputs and velocity commands. Operates within a ROS2 framework to handle communication, data processing, and transformation broadcasting in a robotic system effectively.
- **Equations**: Fundamental for updating the position and orientation of a robot based on sensor inputs and velocity commands.

4.2.2 Publish_Goal.py

Algorithm Description

Various libraries and modules are imported, including `logging`, `nest_asyncio`, `threading`, `rclpy` for ROS 2 communication, message types from `sensor_msgs` and `geometry_msgs`, `pyproj` for coordinate projections, `math` for mathematical calculations, and `telegram` for Telegram bot functionalities.

GoalPublisher Class

- Inherits from `rclpy.node.Node`, initializes a ROS node named `'goal_publisher'`.
- Creates a publisher for `PoseStamped` messages on the topic `'goal_pose'`.
- Defines a list of target locations with coordinates and headings.

- Provides methods:
  - `heading_to_quaternion(heading)`: Converts a heading angle to a quaternion $[qx, qy, qz, qw]$.
  - `calculate_distance(lat1, lon1, lat2, lon2)`: Calculates distance between two GPS coordinates using the Haversine formula.
  - `set_goal_from_location(latitude, longitude)`: Sets the closest predefined target location as a goal position based on given GPS coordinates.

Global Functions

- `extract_coordinates_from_message(message)`: Extracts latitude and longitude from a Telegram message object.
- `determine_message_type(message)`: Determines the type of Telegram message based on its attributes.

Telegram Bot Handlers

- `location(update, context)`: Handles location messages from Telegram users, sets goals using `GoalPublisher`, and manages live location updates.
- `start(update, context)`: Handles the `/start` command from Telegram, initializes a conversation.
- `help_command(update, context)`: Handles the `/help` command from Telegram, provides help information.

Main Function

- Initializes ROS 2 using `rclpy.init()` and creates an instance of `GoalPublisher`.
- Sets up a Telegram bot application with handlers for `/start`, `/help`, and location messages.
- Spins up a separate thread (`ros_thread`) to handle ROS 2 node spinning (`rclpy.spin(goal_publi`
- Runs the Telegram bot application with polling (`application.run_polling()`).
- Cleans up resources (`goal_publisher.destroy_node()` and `rclpy.shutdown()`) after the bot stops.

Equations Used

Haversine Formula

R is the Earth's radius (in meters),

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right)$$

$$c = 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right)$$

$$\text{meters} = R \cdot c$$

where ϕ is latitude, λ is longitude, $\Delta\phi = \phi_2 - \phi_1$, $\Delta\lambda = \lambda_2 - \lambda_1$.

Functions and Methods

- **rclpy.node.Node Methods**: `create_publisher()`, `get_logger()`, `get_clock()`, `destroy_node()`.
- **Telegram Bot Methods**: `send_message()`, `reply_html()`, `reply_text()`, `add_handler()`.

Summary

This script integrates ROS 2 functionality with a Telegram bot. It sets goal positions in a robotics scenario based on incoming GPS coordinates from Telegram users, utilizing ROS 2 for goal publishing and Telegram for user interaction and feedback. The Haversine formula calculates distances between GPS coordinates, crucial for determining proximity to predefined targets.

4.2.3 Publish_goal_exact_xy.py

Algorithm Description

The provided Python script integrates functionality for a system that involves setting and publishing goal coordinates and orientation using a ROS (Robot Operating System) framework, alongside interaction with a Telegram bot for user communication.

Equations Used

1. Transformation from ENU to Map Frame

$$\begin{aligned} \text{map\_x} &= \text{enu\_x} \times \cos(\text{angle\_rad}) - \text{enu\_y} \times \sin(\text{angle\_rad}) \\ \text{map\_y} &= \text{enu\_x} \times \sin(\text{angle\_rad}) + \text{enu\_y} \times \cos(\text{angle\_rad}) \end{aligned}$$

   Where $\text{angle\_rad} = \text{angle\_deg} \times \frac{\pi}{180}$.

   These equations rotate ENU (East-North-Up) coordinates relative to the map frame. angle_deg_ represents the rotation angle between the map frame and the ENU frame.

2. Heading to Quaternion Conversion

$$\begin{aligned} \text{qx} &= 0 \\ \text{qy} &= 0 \\ \text{qz} &= \sin\left(\frac{\text{heading\_rad}}{2}\right) \\ \text{qw} &= \cos\left(\frac{\text{heading\_rad}}{2}\right) \end{aligned}$$

   Where $\text{heading\_rad} = \text{heading} \times \frac{\pi}{180}$.

   This transformation converts a heading angle (in degrees) into a quaternion representation suitable for describing orientation in 3D space.

Functions Description

1. `transform_enu_to_map(enu_x, enu_y)`

   This function takes ENU coordinates (`enu_x`, `enu_y`) and transforms them into the map frame coordinates using a rotation angle specified by angle_deg_. map_x and map_y are computed using standard rotation matrix formulas for 2D coordinates.

2. `heading_to_quaternion(heading)`

   Converts a heading angle (in degrees) into a quaternion representation (`[qx, qy, qz, qw]`) that can be used to represent the orientation (rotation) of an object in 3D space. This is achieved by first converting the heading angle into radians and then computing the corresponding quaternion components.

3. `set_goal_from_location(latitude, longitude)`

   This method initiates the process of setting a goal position based on GPS coordinates (`latitude`, `longitude`). It uses a local Cartesian projection defined by `local_cartesian_` (from the pyproj library) to convert GPS coordinates into ENU coordinates. The resulting ENU coordinates are then transformed into the map frame using `transform_enu_to_map()`. Finally, it converts the goal heading into a quaternion using `heading_to_quaternion()` and publishes the goal position and orientation as a PoseStamped message.

4.2.4 Publish_gps_flask.py

Algorithm Description

1. **Imports and Setup:**
   - Import necessary libraries: `rclpy`, `Node` from `rclpy.node`, `NavSatFix` from `sensor_msgs.msg`, `Flask`, `request`, `jsonify`, and `threading`.
   - Initialize a Flask application (`app`).
2. **Global Variables:**
   - `location_data`: A dictionary to store latitude, longitude, and altitude data received via POST requests.
3. **Class Definition - `GPSPublisher`:**
   - Inherits from `Node` of `rclpy`.
   - Constructor (`__init__`):
     - Initializes the node as `'gps_publisher'`.
     - Creates a publisher (`self.publisher_`) for publishing `NavSatFix` messages on topic `'/fix'`.
   - Method `publish_gps_data`:
     - Receives latitude, longitude, and altitude.
     - Constructs a `NavSatFix` message with these values.
     - Publishes the message via `self.publisher_`.
     - Logs the published data.

4. **Function Definition - `ros2_thread`:**
    - Initializes `rclpy`.
    - Creates an instance of `GPSPublisher` (`gps_publisher`).
    - Spins the node to handle ROS 2 messages.
    - Cleans up by destroying the node and shutting down `rclpy`.
5. **Flask Routes:**
    - **‘`/location`‘ Route (`POST`):**
        - Endpoint to receive JSON data (`latitude`, `longitude`, `altitude`) via POST request.
        - Updates `location_data` dictionary with received data.
        - Calls `gps_publisher.publish_gps_data` to publish GPS data via ROS 2.
        - Returns a JSON response indicating success.
    - **‘`/get_location`‘ Route (`GET`):**
        - Endpoint to retrieve current `location_data`.
        - Returns `location_data` as a JSON response.
6. **Main Execution:**
    - Starts a new thread (`ros2_thread`) to handle ROS 2 operations.
    - Starts the Flask application (`app.run`) to handle HTTP requests on host `'0.0.0.0'`, port `5000`.

Functions Used

- `rclpy.init()`: Initializes the ROS 2 Python client library.
- `rclpy.spin()`: Spins (i.e., enters a loop) to process ROS 2 messages.
- `request.json`: Retrieves JSON data from a Flask HTTP POST request.
- `jsonify()`: Converts Python dictionary to JSON format for HTTP response.
- `Flask.route()`: Decorator to define Flask application routes (`/location` and `/get_location`).
- `threading.Thread()`: Creates a new thread to run `ros2_thread()` concurrently with the Flask application.

4.2.5 publish_gps_telegram.py

Initialization and Logging Setup

The script starts by initializing necessary components like logging and async event handling.

GPSPublisher Class

- `GPSPublisher` class inherits from `rclpy.node.Node` and initializes a ROS node (`gps_publisher`) responsible for publishing GPS coordinates (`NavSatFix` messages) to the topic `/fix`.

Functions and Variables

- `extract_coordinates_from_message(message)`: Extracts latitude and longitude from a Telegram message object if available.
- `determine_message_type(message)`: Determines the type of message (single, edited, live period start/end) based on attributes of the message.
- `handle_end_live_period(user, context)`: Sends a message indicating the end of a live location period.

Async Functions

- `location(update, context)`: Handles incoming Telegram location messages. Publishes GPS coordinates via `gps_publisher.publish_gps()` and sends appropriate messages based on the type of location message received.
- `start(update, context)`: Sends a greeting message to users when they start interacting with the bot.
- `help_command(update, context)`: Sends a help message when users request assistance.

Main Functionality

- `main()`: Initializes ROS (`rclpy.init()`), creates an instance of `GPSPublisher`, sets up a Telegram bot (`Application`), adds command and message handlers, spins up a separate thread (`ros_spin()`) to handle ROS operations concurrently, and runs the Telegram bot (`application.run_polling()`).

Thread Handling

- Uses Python's `threading.Thread` to manage concurrent ROS operations (`ros_spin()`).

Shutdown

- Cleans up resources (`gps_publisher.destroy_node()`, `rclpy.shutdown()`).

Functions and Libraries

- **Libraries**:
  - `logging`: Configures logging for the script.
  - `nest_asyncio`: Enables asyncio support in environments that don't natively support it.
  - `threading`, `rclpy`, `telegram`, `sensor_msgs.msg`, etc.: Import necessary libraries for threading, ROS communication, Telegram bot API, and message types.
- **Functions**:
  - Functions defined (`extract_coordinates_from_message`, `determine_message_type`, `handle_end_live_period`, `location`, `start`, `help_command`) handle various aspects of message processing, user interaction, and ROS node management.

4.2.6 publish_initial_pose.py

Initialization

- Initialize a ROS node (`initial_position_publisher`).
- Set up subscriptions (`gps_sub`, `mag_sub`) and publishers (`scan_pub`, `pose_pub`).
- Initialize parameters related to GPS, map coordinates, and magnetometer.

Map Loading

- `load_map_params`: Loads map parameters from a YAML file (`map_yaml_path`).
- `load_map`: Loads the map image (`map_file_path`) and converts it to a binary representation (`map_data`).

Coordinate Transformations

- `transform_enu_to_map`: Converts ENU (East-North-Up) coordinates to map coordinates using a rotation angle (`angle_deg`).

Magnetometer Callback

- `mag_callback`: Receives orientation data from the magnetometer (`msg.data`) and adjusts it (`self.orientation`) based on an offset (`self.mag_offset`).

GPS Callback

- Converts GPS coordinates (`msg.longitude`, `msg.latitude`) to local Cartesian coordinates (`enu_x`, `enu_y`).
- Transforms Cartesian coordinates to map coordinates using the previously defined rotation angle.
- Sets initial pose (`initial_pose_x`, `initial_pose_y`, `initial_pose_yaw`) once orientation data is available.

Simulated Scan Generation

- `simulate_and_publish_scan`: Generates a simulated LIDAR scan (`scan`) based on the map data.
- Uses ray tracing to determine obstacles in the environment and publishes the simulated scan (`simulated_scan_data`) to `/simulated_scan`.

Actual Scan Callback

- Converts actual LIDAR scan data (`msg`) to points (`actual_scan`).
- Calls `simulate_and_publish_scan` to generate and publish a simulated scan.
- Performs ICP (Iterative Closest Point) registration between actual and simulated scans to correct the initial pose.

ICP and Pose Correction

- Uses Open3D library for point cloud operations.
- Performs two iterations of ICP registration (`reg_p2p`, `reg_p2p2`) to refine the transformation (`transform2`).
- Combines transformations (`combined_transform`) to correct the initial pose (`x`, `y`, `yaw`) and publishes it.

Equations Used

1. Coordinate Transformation

$$\text{map\_x} = \text{enu\_x} \cdot \cos(\text{angle\_rad}) - \text{enu\_y} \cdot \sin(\text{angle\_rad})$$
$$\text{map\_y} = \text{enu\_x} \cdot \sin(\text{angle\_rad}) + \text{enu\_y} \cdot \cos(\text{angle\_rad})$$

   where $\text{angle\_rad} = \text{angle\_deg} \cdot \pi/180.0$.

2. Quaternion Calculation

$$q.w = \cos(\text{yaw}/2.0)$$
$$q.x = 0.0$$
$$q.y = 0.0$$
$$q.z = \sin(\text{yaw}/2.0)$$

Functions Used

1. ROS Functions
   - `create_subscription`, `create_publisher`: Create ROS subscriptions and publishers.
   - `get_logger`: Access ROS logger for debugging messages.
   - `get_clock`: Access ROS clock for timestamping.
2. Math and Utility Functions
   - Trigonometric functions (`math.cos`, `math.sin`) for angle calculations.
   - File handling functions (`open`, `os.path.join`) for YAML and map file operations.
3. Open3D Functions
   - `registration_icp`: Perform ICP registration between point clouds.
   - `PointCloud`, `utility.Vector3dVector`: Manipulate and visualize 3D point clouds.

4.2.7 simulate_scan.py

Initialization

The node `SimulatedScanNode` is initialized, setting up publishers, loading map parameters from a YAML file , and initializing the initial pose of the robot.

Map Loading

- The method `load_map_params` loads parameters from a YAML file, including the map image file path, resolution, origin coordinates, and orientation.
- The method `load_map` loads the map image, converts it to a grayscale numpy array, flips it vertically to align the origin, and binarizes it where obstacles are marked as 1 and free space as 0.

Static Transform Broadcast

The method `publish_static_transform` broadcasts a static transformation between the `map` frame and `laser_frame`, using the initial pose coordinates and orientation converted to quaternion format.

Scan Generation and Publishing

- The method `generate_and_publish_scan` generates a simulated laser scan:
  - Defines laser parameters such as minimum and maximum ranges, start and end angles of the scan, number of samples, and angle increment.
  - Initializes a `LaserScan` message with these parameters.
  - Iterates through each sample angle, computes the endpoint coordinates based on robot pose and angle, converts these to map coordinates, and checks against the loaded map data.
  - If an obstacle is detected at a given range, updates the `ranges` array in the `LaserScan` message.
  - Publishes the `LaserScan` message to `/simulated_scan`.

Utility Functions

- `compute_distance_to_obstacle(x, y, resolution)`: Computes the Euclidean distance from a point to the origin (not directly used in the main scan generation).
- `yaw_to_quaternion(yaw)`: Converts a yaw angle (orientation) to a `Quaternion` message for orientation representation in ROS.

Equations

- $\text{angle\_increment} = \frac{\text{angle\_max} - \text{angle\_min}}{\text{samples}}$
- $x = \text{initial\_pose\_x} + r \cdot \cos(\text{angle} + \text{initial\_pose\_yaw})$
- $y = \text{initial\_pose\_y} + r \cdot \sin(\text{angle} + \text{initial\_pose\_yaw})$
- $\text{map\_x} = \frac{(x - \text{origin\_x})}{\text{resolution}}$
- $\text{map\_y} = \frac{(y - \text{origin\_y})}{\text{resolution}}$

Functions

- `load_map_params(yaml_file_path)`: Loads map parameters from a YAML file.
- `load_map(map_file_path)`: Loads and processes the map image.
- `publish_static_transform()`: Publishes a static transformation between map and laser frame.
- `generate_and_publish_scan()`: Generates and publishes a simulated laser scan based on map data.
- `compute_distance_to_obstacle(x, y, resolution)`: Computes the distance to an obstacle from a given point (not directly used in scan generation).
- `yaw_to_quaternion(yaw)`: Converts yaw angle to a quaternion for orientation representation.

4.2.8 send_commands.py

This Python script defines a ROS2 node (`TwistConverterNode`) responsible for converting Twist messages into steering and velocity commands for controlling a vehicle via an Arduino interface. It subscribes to two topics (`cmd_vel` and `stop_car`) for receiving movement commands and stop signals, respectively.

Classes and Libraries Used

- `rclpy`: ROS 2 Python client library.
- `geometry_msgs.msg`: ROS 2 message types, including `Twist` for defining linear and angular velocities.
- `std_msgs.msg`: ROS 2 standard message types, including `Int32` for signaling stop commands.

- `math`: Python standard library for mathematical functions (`math.tan`, `math.atan2`, `math.degrees`).
- `serial`: Python library for serial communication with external devices (Arduino).
- `time`: Python standard library for time-related functions (`time.sleep`).
- `struct`: Python standard library for packing and unpacking data into bytes (`struct.pack`).

ROS Node Initialization (`__init__` method)

- Initializes the ROS2 node (`twist_converter_node`).
- Subscribes to the `cmd_vel` topic to receive Twist messages (`Twist`) containing velocity and steering commands.
- Subscribes to the `stop_car` topic to receive stop signals (`Int32`).
- Initializes parameters for wheel separation (`self.wheel_separation`), wheel base (`self.wheel_base`), and the initial steering angle (`self.steering_angle`).
- Establishes a serial connection (`self.arduino`) to an Arduino at a specified port and baud rate.

Twist Callback (`twist_callback` method)

- **Function:** Receives a Twist message (`msg`) and extracts linear and angular velocities.
- Calls `convert_twist_to_velocity_and_steering` to calculate the steering angle and linear velocity.
- Sends the calculated data to the Arduino using `send_data_to_arduino`.

Stop Car Callback (`stop_car_callback` method)

- **Function:** Receives an Int32 message (`msg`) indicating a stop command (if `msg.data == 1`).
- Constructs a stop command as bytes (`command_bytes`) using `struct.pack`.
- Sends the stop command to the Arduino via `self.arduino.write`.
- Stops the ROS node and shuts down ROS 2 (`rclpy.shutdown`).

Calculate Steering Angle (`calculate_steering_angle` method)

- **Function:** Takes a target rotation (`target_rot`) and computes the steering angles for both left and right wheels.

  Equations:

$$\text{tanSteer} = \tan(\text{target\_rot})$$

$$\text{steering\_angle\_left} = \tan^{-1}\left(\frac{\text{tanSteer}}{1 - \frac{\text{wheel\_separation}}{2 \times \text{wheel\_base}} \times \text{tanSteer}}\right)$$

$$\text{steering\_angle\_right} = \tan^{-1}\left(\frac{\text{tanSteer}}{1 + \frac{\text{wheel\_separation}}{2 \times \text{wheel\_base}} \times \text{tanSteer}}\right)$$

- **Description:** Calculates the steering angles (`steering_angle_left` and `steering_angle_right`) for the left and right wheels based on the desired rotation (`target_rot`). It adjusts the steering based on the vehicle's physical parameters (`wheel_separation` and `wheel_base`) to ensure balanced and effective turning.

- Adjusts and limits the calculated steering angle to ensure it remains within a safe operational range.

  Equation:

$$\text{self.steering\_angle} = \max(\min(\text{self.steering\_angle}, 0.74), -0.74)$$

- **Description:** Limits the calculated steering angle (`self.steering_angle`) to a maximum of 0.74 radians (approximately 42.4 degrees) and a minimum of -0.74 radians (approximately -42.4 degrees) to prevent excessive steering and maintain stability.

- Converts the steering angle from radians to degrees for clarity and control purposes.

  Equation:

$$\text{self.steering\_angle} = \text{math.degrees}(\text{self.steering\_angle})$$

- **Description:** Converts the steering angle (`self.steering_angle`) from radians to degrees to facilitate easier understanding and control over the vehicle's steering operations.

- Computes the change in steering angle relative to its previous value (`old_steering_angle`).

  Equation:

$$\text{steering\_angle} = \text{self.steering\_angle} - \text{old\_steering\_angle}$$

- **Description:** Determines the difference (`steering_angle`) between the current steering angle (`self.steering_angle`) and its previous value (`old_steering_angle`), indicating how much the steering angle has changed or adjusted.

Convert Twist to Velocity and Steering (`convert_twist_to_velocity_and_steering` method)

- **Function:** Extracts linear (`target_linear`) and angular (`target_rot`) velocities from a received Twist message.
- Computes the steering angle using `calculate_steering_angle` based on the angular velocity (`target_rot`).

Send Data to Arduino (`send_data_to_arduino` method)

- **Function:** Takes the calculated steering angle and linear velocity as inputs.
- **Adjust Steering Angle for Arduino:**

  Equation:

$$\text{steering\_angle} = \text{int}(\text{steering\_angle} \times 497.84) \times (-1)$$

- **Description:** Converts the calculated steering angle to an integer format (`int(...)`) suitable for communication with the Arduino. It scales the steering angle by 497.84 and negates it to match the Arduino's expected format and directionality.
- **Adjust Linear Velocity for Arduino:**

  Equation:

$$\text{linear\_velocity} = \text{linear\_velocity} \times \left(\frac{18}{5}\right)$$

- **Description:** Scales the linear velocity (`linear_velocity`) by a factor of 3.6 to adjust its units or scale to match the requirements of the Arduino.
- **Pack and Send Data:**

  Equation:

$$\text{command\_bytes} = \text{struct.pack}('< fi', \text{linear\_velocity}, \text{steering\_angle})$$

- **Description:** Packs the adjusted linear velocity and steering angle into bytes (`command_bytes`) using the `struct.pack` function with format '¡fi' (float and integer types) for transmission to the Arduino.
- **Send Command to Arduino:**
- **Description:** Transmits the packed bytes (`command_bytes`) containing the adjusted linear velocity and steering angle to the Arduino via the established serial connection (`self.arduino.write`).

Summary

This script integrates ROS 2 functionalities with Arduino-based hardware to control a vehicle's motion based on Twist messages. It utilizes mathematical calculations for steering angle adjustments, byte packing for serial communication, and ROS 2 message handling for seamless integration with other ROS 2 nodes and systems. The combined functionality ensures accurate and responsive control over the vehicle's movement and operational states.

4.2.9 stop_car_flask.py

Algorithm Description

1. **Imports and Setup:**
   - Imports necessary libraries such as `requests`, `Flask`, `Thread`, `subprocess`, `time`, `rclpy`, and specific ROS2 packages (`Node`, `Int32`).
2. **Flask App Setup:**
   - Initializes a Flask application (`app`) to handle incoming HTTP requests.
   - Defines a route `/message` that accepts both `GET` and `POST` requests.
3. **ROS2 Node Setup:**
   - Defines a ROS2 node `StopCarPublisher` that publishes messages of type `Int32` to the topic `stop_car`.
   - Provides a method `publish_stop_signal()` to publish a specific signal (`Int32` data set to 1) indicating a stop command.
4. **Server Functions:**
   - `receive_message()`: Handles incoming messages sent to `/message` endpoint. If a POST request is received, retrieves JSON data. If the message contains `"Stop"`, invokes `stop_car_publisher.publish_stop_signal()`.
   - `run_server()`: Starts the Flask server on port 5000 without reloading for development (`use_reloader=False`).
   - `start_localtunnel()`: Initiates a local tunneling service using the `lt` command-line tool to expose the Flask server to the internet using a specified subdomain (`yoursubdomain`).
   - `ros2_init()`: Initializes ROS2 using `rclpy.init()`. Creates an instance of `StopCarPublisher`. Spins (runs) the ROS2 node (`rclpy.spin(stop_car_publisher)`) to handle ROS2 events and message publishing.

5. **Main Execution:**

   - Starts the ROS2 node (`ros2_thread`) and Flask server (`server_thread`) in separate threads to run concurrently.
   - Launches the `start_localtunnel()` function to make the Flask server publicly accessible.
   - Uses a `while True` loop to keep the main thread alive, allowing the application to continue running and handle events.

6. **Error Handling:**

   - Catches exceptions (`Exception` and `KeyboardInterrupt`) to gracefully shut down the server and threads if errors occur or if the user interrupts the program.

Functions Used

- `requests`: Handles HTTP requests (`request.get_json()`, `request.args.to_dict()`).
- `Flask`: Web framework for handling HTTP requests and responses (`Flask`, `request`, `jsonify`).
- `Thread`: Allows concurrent execution of multiple tasks (`Thread`).
- `subprocess`: Runs external processes such as `lt` for local tunneling (`subprocess.Popen`).
- `time`: Provides time-related functions (`time.sleep`).
- `rclpy`: ROS2 Python client library (`rclpy.init()`, `Node`, `Int32`, `create_publisher`, `spin`, `destroy_node`, `shutdown`).

4.2.10 stop_car_telegram.py

Imports and Setup

- Imports necessary libraries and modules (`logging`, `nest_asyncio`, `threading`, `rclpy`, `telegram`, etc.).
- Configures logging for the application.

`StopCarPublisher` **Class**

- **Description:** Inherits from `Node` in `rclpy.node`.
- **Purpose:** Represents a ROS node (`stop_car_publisher`) responsible for publishing a stop signal (`Int32` message) on topic `/stop_car`.
- **Equations:**
  - `msg.data` = 1: Assigns the integer 1 to the `Int32` message.

Functions and Handlers

- `handle_message(update, context)`:
  - **Description:** Asynchronously handles incoming messages from Telegram.
  - **Equations:** None directly, but triggers `stop_car_publisher.publish_stop_signal()` if message equals `"1"`.
- `start(update, context)`:
  - **Description:** Asynchronously sends a greeting message when the `/start` command is received on Telegram.
  - **Equations:** Uses `update.message.reply_html()` to reply with a formatted message.
- `help_command(update, context)`:
  - **Description:** Asynchronously sends a help message when the `/help` command is received on Telegram.
  - **Equations:** Uses `update.message.reply_text()` to send a simple text reply.

Main Function (`main()`)

- **Description:** Entry point of the application.
- **Functions:**
  - `rclpy.init()`: Initializes the ROS client library.
  - `StopCarPublisher()`: Creates an instance of `StopCarPublisher`.
  - `Application.builder().token().build()`: Configures and builds a Telegram bot application instance.
  - `application.add_handler()`: Registers command and message handlers for Telegram commands and messages.

- `Thread(target=ros_spin)`: Creates a thread to run `ros_spin()` function asynchronously.
- `application.run_polling()`: Starts polling for Telegram updates.
- `stop_car_publisher.destroy_node()`: Cleans up ROS node resources.
- `rclpy.shutdown()`: Shuts down the ROS client library.

Functions and Libraries

- **Imports:**
  - `logging`: Python's logging framework for application logging.
  - `nest_asyncio`: Integrates asyncio with other event loops.
  - `threading`: Provides threading support for concurrent execution.
  - `rclpy`: ROS 2 Python client library.
  - Telegram libraries (`telegram`, `telegram.ext`): For interacting with the Telegram API.
- **Functions:**
  - `handle_message(update, context)`: Handles incoming messages on Telegram, publishing a stop signal if the message is `"1"`.
  - `start(update, context)`: Sends a greeting message when `/start` command is received on Telegram.
  - `help_command(update, context)`: Sends a help message when `/help` command is received on Telegram.
  - `ros_spin()`: Spins the ROS node to handle ROS events.
  - `main()`: Initializes ROS, sets up Telegram bot, starts threads for ROS and Telegram event handling, and cleans up resources.

4.3 Arduino Sketches

4.3.1 sensor_data

Detailed Algorithm Description

1. Setup Function
   - **Serial Communication Setup:** Initializes serial communication with a baud rate of 115200, essential for debugging and data output to a connected device.

- **Pin Mode Setup:** Configures `Pin1` and `Pin2` as input pins, likely connected to wheel speed sensors (`sensor_1` and `sensor_2`).

2. Loop Function
   - **Sensor State Detection:** Continuously monitors the digital states (`HIGH` or `LOW`) of `Pin1` and `Pin2`, representing the states of the wheel speed sensors.
   - **Velocity Calculation and Printing:** Depending on the sensor states, calculates velocities (`velocity1` and `velocity2`) using the `calculateVelocity` function.
   - **Data Output:** Calls `print_data` function to print sensor data and calculated velocities to the serial monitor.
3. calculateVelocity Function
   - **Velocity Calculation:** Computes the current velocity of the wheels based on the time elapsed (`timeElapsed`) since the last sensor activation.
   - **Distance Calculation:** Uses the circumference of the wheel (`circumference`) to compute the distance traveled for each sensor activation (`distance_traveled`).
   - **Unit Conversion:** Converts the velocity from cm/s to m/s for a standardized output.
4. print_imu_data Function
   - **IMU Data Acquisition:** Reads raw accelerometer, gyroscope, and magnetometer data from the MPU9250 sensor.
   - **Calibration:** Applies calibration matrices (`A_a`, `b_a` for accelerometer and `A_m`, `b_m` for magnetometer) to correct for biases and scaling errors in the sensor readings.
   - **Conversion to Physical Units:** Converts the calibrated sensor readings into meaningful physical units (e.g., acceleration in m/s^2, magnetic direction in degrees).
   - **Data Output:** Prints the calibrated accelerometer data (x, y, z), gyroscope data (z-axis only), and magnetometer data (direction) to the serial monitor.
5. print_data Function
   - **Sensor Data and Velocity Printing:** Utilizes `print_imu_data` to fetch calibrated sensor data.
   - **Average Velocity Calculation:** Computes the average velocity (`velocity1`+ `velocity2`)/2.
   - **Time Stamp:** Includes `timeElapsed` to indicate the time interval since the last data print.
   - **Output:** Prints the averaged velocity and elapsed time to the serial monitor, along with the calibrated sensor data.

Equations and Formulas Used

1. Velocity Calculation

$$\text{velocity} = \frac{\text{distance\_traveled}}{\text{timeElapsed}} \times \frac{1}{100}$$

Explanation: Calculates the current velocity of the wheels by dividing the distance traveled (`distance_traveled`) by the time elapsed (`timeElapsed`). The result is converted from cm/s to m/s for standardization.

2. Distance Traveled per Sensor Activation

$$\text{distance\_traveled} = \frac{2 \times \pi \times 22.5}{4}$$

Explanation: Computes the distance traveled for each sensor activation based on the wheel's circumference ($2 \times \pi \times 22.5$) divided by 4, reflecting the number of activations per complete wheel rotation.

Functions and Libraries

1. Libraries
   - `MPU9250_asukiaaa.h`: Integrates the MPU9250 sensor library, providing functions to initialize and read data from the accelerometer, gyroscope, and magnetometer.
2. Functions
   - `calculateVelocity(unsigned long &previousMillis)`: Calculates and returns the current wheel velocity based on elapsed time since the last sensor activation.
   - `print_imu_data()`: Reads raw sensor data from the MPU9250, applies calibration matrices to adjust for sensor biases, converts data into physical units, and prints calibrated sensor readings to the serial monitor.
   - `print_data(double timeElapsed)`: Fetches and prints calibrated sensor data, computes the average wheel velocity, and outputs both velocity and time elapsed to the serial monitor for real-time monitoring and analysis.

Application Context

This code is suitable for applications requiring precise motion tracking and orientation sensing, such as robotics, unmanned vehicles, or inertial measurement units (IMUs). By integrating sensor data with real-time velocity calculations, it provides comprehensive feedback on vehicle dynamics and environmental conditions. The calibration matrices (`A_a`, `b_a`, `A_m`, `b_m`) ensure accurate sensor readings, critical for reliable operation in dynamic environments.

4.3.2 car_control

Algorithm Description

1. PID Setup
    - Initialize PID constants (kp, ki, kd) for proportional, integral, and derivative gains respectively.
    - Set up PID controller (`myPID`) with input (`input`), output (`output`), and setpoint (`setpoint`).
    - Configure PID to operate in automatic mode (`myPID.SetMode(AUTOMATIC)`) and set output limits (`myPID.SetOutputLimits(0, 21)`).
2. Setup Function
    - Configure pin modes for various pins used for motor control (`Relay1`, `Relay2`, `velInput`, `ForwardPin`, `BackwardPin`, `feedbackPin`).
    - Initialize serial communication and stepper motor settings (`stepper.setSpeed(1)`).
3. Main Loop (`loop` function)
    - Check if serial data is available (`Serial.available()`).
    - Read velocity (`float`) and steps (`int32_t`) data from serial input.
    - Call `processVelocity` function with the received velocity to control motor direction and speed based on the velocity value.
    - Call `steering` function to control the stepper motor to move a specific number of steps (`steps`).
4. Process Velocity (`processVelocity` function)
    - Determine motor direction (`ForwardPin` or `BackwardPin`) based on the sign of `velocity`.
    - Calculate feedback frequency from a sensor connected to `feedbackPin` to estimate `CarSpeed`.
    - Set `input` (current speed) and `setpoint` (target speed) for PID controller.
    - Compute PID output (`output`) to adjust motor speed (`velInput`).
5. Steering Function
    - Calculate stepper motor speed (`speedo`) based on the required steps per minute (`stepsPerMin`) and sampling time (`samplingtime`).
    - Move the stepper motor (`stepper.step(steps)`) by the specified number of steps (`steps`).

Equations Used

1. PID Calculation

$$\text{output} = K_p \cdot e(t) + K_i \cdot \int e(t)\,dt + K_d \cdot \frac{de(t)}{dt}$$

where $e(t) = \text{setpoint} - \text{input}$, and K_p, K_i, K_d are the proportional, integral, and derivative gains respectively.

2. Motor Speed Calculation

$$\text{speedo} = \lceil \frac{\text{stepsPerMin}}{\text{samplingtime}} \times |\text{steps}| \rceil$$

3. Feedback Frequency Calculation

$$\text{feedbackFrequency} = \frac{1}{\text{feedbackDuration} \times 0.000001}$$

where `feedbackDuration` is the pulse duration from the feedback sensor.

Functions and Libraries

1. Libraries Used
   - `PID_v1.h`: Provides PID control functionalities.
   - `Stepper.h`: Handles stepper motor control.
2. Functions
   - `brakeInitiation, brakeRelease, brakePause, stopMotor`: Functions commented out, but intended for braking control, involving relay operations and motor stopping.