ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

----------***----------

BÁO CÁO CUỐI KỲ

HỌC PHẦN: LẬP TRÌNH ROBOT VỚI ROS

Mã lớp: RBE3017_1

Giảng viên hướng dẫn: TS. Lê Xuân Lực

Trợ giảng: Dương Văn Tân

ĐỀ TÀI: ỨNG DỤNG TINY SLAM CHO ROBOT HÚT BỤI

Sinh viên thực hiện: Nguyễn Tuấn Anh

Phạm Trung Anh

Nguyễn Tuấn Cảnh

Phàn Quý Đường

- Hà Nội, ngày 20 tháng 4 năm 2025 –

Phân công nhiệm vụ

Họ và Tên	Nhiệm vụ
Nguyễn Tuấn Anh	Tìm hiểu thuật toán, viết báo cáo
Phạm Trung Anh	Vẽ, tạo urdf và viết chương trình điều khiển robot, chạy và xử lý video
Nguyễn Tuấn Cảnh	Tìm hiểu thuật toán, viết thuật toán, viết chương trình, chạy kiểm thử
Phàn Quý Đường	Tìm hiểu thuật toán, xử lý lỗi, chạy kiểm thử, viết báo cáo

Mục Lục

Mở đầu

Trong bối cảnh công nghệ phát triển nhanh chóng, các hệ thống robot ngày càng được ứng dụng rộng rãi trong nhiều lĩnh vực như tự động hóa, dọn dẹp nhà cửa, khảo sát môi trường và công nghiệp. Một trong những thách thức lớn đối với các robot tự động là khả năng xác định vị trí và điều hướng trong môi trường chưa biết. Để giải quyết vấn đề này, các phương pháp SLAM (Simultaneous Localization and Mapping) đã được nghiên cứu và áp dụng rộng rãi.

Dự án này tập trung vào việc phát triển một hệ thống SLAM cho robot nhỏ gọn sử dụng LiDAR và thuật toán Monte Carlo Scan Matching (MCSM). LiDAR là một công nghệ cảm biến tiên tiến giúp thu thập dữ liệu không gian chính xác, trong khi thuật toán MCSM cung cấp phương pháp mạnh mẽ để xác định vị trí và xây dựng bản đồ môi trường trong thời gian thực. Bằng cách kết hợp hai công nghệ này, hệ thống SLAM sẽ giúp robot tự động xác định vị trí của mình trong không gian và liên tục cập nhật bản đồ của môi trường xung quanh.

Mục tiêu của báo cáo này là mô tả chi tiết quá trình thiết kế, phát triển và thử nghiệm hệ thống SLAM sử dụng LiDAR và Monte Carlo Scan Matching. Báo cáo sẽ trình bày các phương pháp kỹ thuật, kết quả thực nghiệm, và ứng dụng của hệ thống trong các robot tự động, đặc biệt là trong các môi trường động và không xác định.

Chương 1.Giới thiệu và xác định các chỉ tiêu kỹ thuật

I.Giới thiệu về SLAM

Simultaneous localization and mapping (SLAM) là bài toán tính toán xây dựng hoặc cập nhật bản đồ của một môi trường không xác định đồng thời theo dõi vị trí của tác nhân bên trong nó. Mặc dù ban đầu đây có vẻ là một bài toán con gà và quả trứng nhưng có một số thuật toán được biết đến để giải nó, ít nhất là khoảng thời gian có thể giải quyết được cho một số môi trường nhất định. Các phương pháp giải gần đúng phổ biến bao gồm bộ lọc hạt, bộ lọc Kalman mở rộng, giao điểm hiệp phương sai và GraphSLAM . Các thuật toán SLAM dựa trên các khái niệm trong hình học tính toán và thị giác máy tính , và được sử dụng trong điều hướng robot , lập bản đồ robot và đo đường cho thực tế ảo hoặc thực tế tăng cường .

II.Tìm hiểu TinySLAM

TinySLAM là một thuật toán SLAM đơn giản, được thiết kế để hoạt động hiệu quả trên các hệ thống robot có tài nguyên hạn chế. Thuật toán này sử dụng dữ liệu từ cảm biến laser và odometry để xây dựng bản đồ và định vị robot trong môi trường.

III. Giới thiệu về dự án mô phỏng robot với TinySLAM

Trong dự án này, chúng tôi tập trung vào việc phát triển cơ bản về hệ thống robot ứng dụng công nghệ TinySLAM để quét bản đồ 2D và xác định vị trí của Robot trong môi trường. Hệ thống này sử dụng cảm biến LiDAR và thuật toán Monte Carlo Scan Matching (MCSM) để cho phép robot xác định vị trí của mình trong môi trường và đồng thời cập nhật bản đồ mà không cần kiến thức trước về môi trường đó.

1.Mục tiêu:

Tạo ra hệ thống robot có khả năng quét và lập bản đồ 2D của môi trường xung quanh.

Sử dụng cảm biến Lidar để thu thập dữ liệu về khoảng cách và hình dạng của các vật thể trong không gian

Từ các dữ liệu đó sử dụng thuật toán MCSM để giúp robot xác định chính xác vị trí của mình trong không gian, ngay cả khi có sự thay đổi hoặc nhiễu trong môi trường xung quanh.

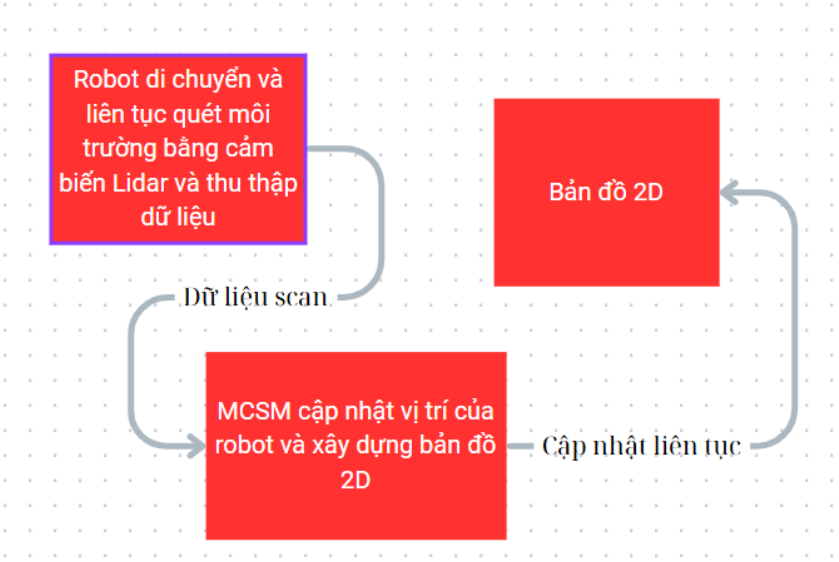
2.Công nghệ sử dụng:

Cảm biến Lidar: được sử dụng để đo khoảng cách đến các vật thể trong môi trường, từ đó tạo ra các điểm dữ liệu 2D cho bản đồ.

Monte Carlo Scan Matching (MCSM): Đây là một kỹ thuật cơ bản để đồng bộ hóa các phép đo LiDAR với bản đồ môi trường hiện tại. Thuật toán MCSM sử dụng các "hạt"

(particles) để tìm kiếm vị trí và hướng của robot bằng cách so sánh các phép đo LiDAR mới với bản đồ đã xây dựng. Mỗi hạt đại diện cho một khả năng vị trí của robot, và thông qua việc đối chiếu dữ liệu quét LiDAR, các hạt này sẽ được tái phân phối, giúp robot ước tính vị trí chính xác. Mỗi particle đại diện cho một khả năng khác nhau về vị trí và trạng thái của robot. Qua thời gian, các particle sẽ được điều chỉnh để phản ánh chính xác hơn vị trí thực tế của robot.

3.Quá trình Lập Bản đồ và Định vị:



Hình 1. quá trình lập bản đồ và định vị

Chương 2. Nội dung

I.Mô tả các file

1. Foder tiny-slam-ros-cpp

1.1Các thành phần chính

Hệ thống được tổ chức thành nhiều lớp và cấu trúc dữ liệu, mỗi lớp đảm nhiệm một vai trò cụ thể:

- Dữ liệu cảm biến: ScanPoint, TransformedLaserScan (sensor_data.h).
- Trạng thái robot: RobotState, World (state_data.h).
- Bản đồ lưới: GridMap, GridCell, GridCellFactory (grid_map.h, grid_cell_factory.h).
- Chiến lược ô lưới: GridCellStrategy, CellOccupancyEstimator, AreaOccupancyEstimator, ConstOccupancyEstimator (grid_cell_strategy.h,

cell_occupancy_estimator.h, area_occupancy_estimator.h, const_occupancy_estimator.h).

- Khớp quét: MonteCarloScanMatcher (monte_carlo_scan_matcher.h).
- Quản lý môi trường: LaserScanGridWorld (laser_scan_grid_world.h).
- Tiny_slam.cpp: Triển khai các node ROS chính để thực hiện thuật toán TinySLAM, xử lý dữ liệu cảm biến và odometry để xây dựng bản đồ.

Mục đích tổng quan: đọc dữ liệu laser scan từ robot, dùng thuật toán tinyslam để xây dựng bản đồ và ước lượng vị trí robot, phát bản đồ(OccupancyGrid) và trạng thái robot ra các topic để các node khác và rviz có thể xem.

+ lớp PoseScanMatcherObserver dùng để phát vị trí robot sau mỗi lần scanmatcher hoặc cập nhật pose tốt nhất

Hàm on_scan_test(pose, scan, score) để phát ra pose hiện tại đang test

Gọi hàm publish_transform với các tham số:

- tên frame mới: "sm_curr_pose"
- frame gốc: _frame_odom
- vị trí robot: pose

Hàm on_pose_update(pose, scan, score) được gọi: để phát ra pose tốt nhất tìm được sau scan matching

Gọi hàm publish_transform với:

- frame mới: "sm_best_pose"
- frame gốc: _frame_odom
- vị trí robot:

Ngoài ra các hàm init để khởi tạo các tham số cần thiết được load từ file empty_would.launch trong foder robot.

Hàm main

Lấy các tham số cấu hình cho ros, định danh các frame name.

Tạo một đối tượng scan_observer với:

- NodeHandle: nh
- Tên topic: "laser_scan"

- Frame gốc: frame_odom

- Bộ đệm TF: ros_tf_buffer_size

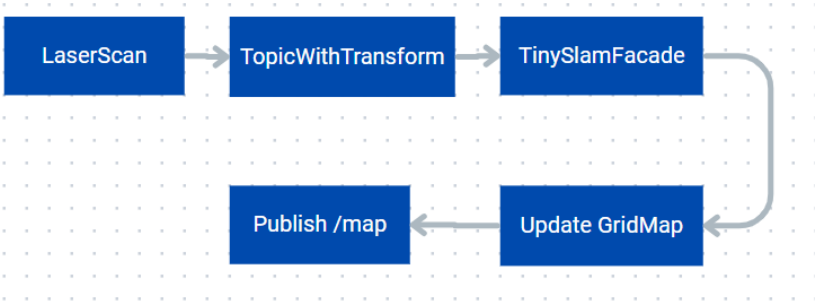
- Độ lớn hàng đợi lọc: ros_filter_queue

- Độ lớn hàng đợi subscribe: ros_subscr_queue

Tạo và đăng ký subcriber cho laserscan, mỗi lần scan mới sẽ gọi slam->process_scan và tự động lấy transform từ TF để chỉnh dữ liệu scan.

Tạo một viewer publish /map dưới dạng nav_msgs::OccupancyGrid. Đây là phần hiển thị trong rviz

Cuối cùng là lặp vô hạn để tiếp tục xử lý scan mới.

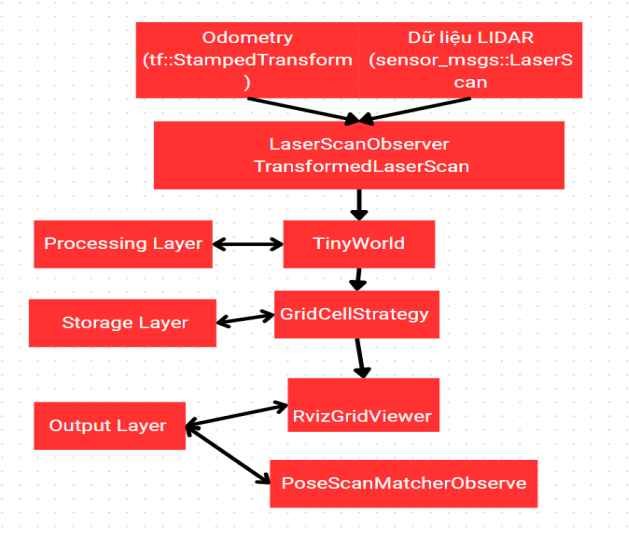


Hình 2. Sơ đồ luồng hoạt động

1.2 Kiến trúc hệ thống

Hệ thống tinySLAM được thiết kế theo mô hình phân lớp (layered architecture), với các thành phần tương tác chặt chẽ để xử lý dữ liệu cảm biến, cập nhật bản đồ, và định vị robot.

Sơ đồ kiến trúc



Hình 3. Sơ đồ kiến trúc

Trong đó các khối:

LaserScanObserver

- Subscribes:
  /scan, /tf
- Methods:
  convert_scan(),
  extract_odometry()
- Outputs:
  TransformedLaserScan
  - points: ScanPoint[]
  - d_x, d_y, d_yaw
  - quality

TinyWorld

- Manages:
  RobotState, GridMap
- Methods:
  handle_observation(),
  update_robot_pose()
- Inputs:
  TransformedLaserScan
- Calls:
  TinyScanMatcher,
  GridMap::update_cell()
- Outputs:
  Updated RobotState,
  Updated GridMap

Processing Layer
[RobotState]

- x, y, theta
- update(d_x, d_y, d_theta)

[GridMap]

- _cells: GridCell[][]
- _m_per_cell, _width, _height
- Methods:
  update_cell(),
  world_to_cell(),
  extend_if_needed()

[TinyScanMatcher]

- MonteCarloScanMatcher
- Methods:
  process_scan(),
  estimate_scan_cost()
- Inputs:
  RobotState, Scan, GridMap
- Outputs:
  pose_delta, min_scan_cost

GridCellStrategy

- Contains:
  GridCellFactory
  ScanCostEstimator
  CellOccupancyEstimator
- Methods:
  estimate_occupancy()
- Implementations:
  AreaOccupancyEstimator
  ConstOccupancyEstimator
- Outputs:
  Occupancy
    - prob_occ, quality

Storage Layer

[GridCell]

- value: float (0.0-1.0)

- Methods:
  value(), set_value()

[GridCellFactory]

- TinyBaseCellFactory
- Methods:
  create_cell()

RvizGridViewer

- Publishes:
  /map (OccupancyGrid)
  /pose (Pose)
- Methods:
  publish_map(),
  publish_pose()
- Inputs:
  GridMap, RobotState

Output Layer
[ROS Topics]

- /map: nav_msgs::OccupancyGrid
- /pose: geometry_msgs::Pose
- /tf: tf::Transform

[PoseScanMatherObserver]

- Publishes:
  /tf (test poses)
- Methods:
  on_scan_test(),
  on_pose_update()

PossScanMatcherObserver

- Publishes:
  /tf (test poses)
- Methods:
  on_matching_start(),

on_scan_test(),
on_pose_update(),
on_matching_end()

- Inputs:
  RobotState, Scan, Cost

Nhiệm vụ của mỗi khối:

- LaserScanObserver: Chuyển đổi dữ liệu ROS thành TransformedLaserScan.
- TinyWorld: Quản lý bản đồ (GridMap) và trạng thái robot (RobotState), xử lý quét laser.
- TinyScanMatcher: Thực hiện khớp quét sử dụng thuật toán Monte Carlo.
- GridMap: Lưu trữ bản đồ lưới với các ô (GridCell).
- GridCellStrategy: Container cho GridCellFactory, ScanCostEstimator, và CellOccupancyEstimator.
- RvizGridViewer: Xuất bản bản đồ và vị trí robot để hiển thị trên RViz.

Luồng dữ liệu

a. Lớp nhập liệu (Input Layer):
   - Nhận dữ liệu từ LIDAR (sensor_msgs::LaserScan) và odometry (tf::StampedTransform) qua LaserScanObserver.
   - Chuyển đổi thành TransformedLaserScan (dữ liệu quét laser nội bộ).

b. Lớp xử lý (Processing Layer):
   TinyWorld (kế thừa LaserScanGridWorld): Quản lý bản đồ (GridMap) và vị trí robot (RobotState).
   TinyScanMatcher (kế thừa MonteCarloScanMatcher): Thực hiện khớp quét để hiệu chỉnh vị trí robot.
   GridCellStrategy: Cung cấp các chiến lược để tạo ô, ước lượng xác suất chiếm chỗ, và đánh giá chi phí quét.

c. Lớp lưu trữ (Storage Layer):

   GridMap: Lưu trữ bản đồ lưới với các ô (GridCell).

   GridCell: Lưu trữ xác suất chiếm chỗ của từng ô.

d. Lớp xuất liệu (Output Layer):

   RvizGridViewer: Xuất bản bản đồ và vị trí robot qua ROS để hiển thị trên RViz.

Luồng dữ liệu chi tiết

a. Nhập liệu:
    - Nguồn dữ liệu:
        - LIDAR cung cấp quét laser qua topic /scan (sensor_msgs::LaserScan), chứa danh sách khoảng cách và góc.
        - Odometry cung cấp thay đổi vị trí qua topic /tf (tf::StampedTransform), bao gồm dịch chuyển và xoay.
    - Xử lý: LaserScanObserver chuyển đổi thành TransformedLaserScan, bao gồm:
        - Danh sách ScanPoint (khoảng cách, góc, trạng thái chiếm chỗ).
        - Thay đổi odometry (d_x, d_y, d_yaw).
        - Chất lượng quét (quality, thường từ 0.0 đến 1.0).
b. Xử lý:
    - TinyWorld: Nhận TransformedLaserScan, thực hiện:
        - Gọi TinyScanMatcher để khớp quét, hiệu chỉnh vị trí robot trong RobotState.
        - Chiếu các điểm quét lên GridMap, cập nhật xác suất chiếm chỗ bằng GridCellStrategy.
    - TinyScanMatcher: Sử dụng MonteCarloScanMatcher để thử nghiệm ngẫu nhiên các vị trí, tìm vị trí tối ưu dựa trên chi phí quét.
c. Lưu trữ:
    - GridMap: Lưu trạng thái các ô (GridCell), cập nhật xác suất chiếm chỗ bằng GridCell::set_value.
    - GridCellStrategy: Cung cấp CellOccupancyEstimator để tính xác suất chiếm chỗ (Occupancy).
d. Xuất liệu:
    - RvizGridViewer: Xuất bản:
        - /map (nav_msgs::OccupancyGrid): Bản đồ lưới từ GridMap, với xác suất chiếm chỗ được chuyển đổi thành giá trị 0-100.
        - /pose (geometry_msgs::Pose): Vị trí robot từ RobotState (x, y, theta).

        - /tf (qua PoseScanMatcherObserver): Vị trí thử nghiệm trong scan matching.

1.3. Các bộ phận cốt lõi

a. Dữ liệu cảm biến

- ScanPoint (sensor_data.h):
    - Lưu trữ một điểm quét laser ở tọa độ cực (range, angle) và trạng thái chiếm chỗ (is_occupied).
    - Ví dụ: range = 2.5 (mét), angle = 0.3 (radian), is_occupied = true (chướng ngại).
    - Vai trò: Đơn vị cơ bản của dữ liệu quét, được sử dụng để chiếu lên bản đồ và cập nhật ô.
- TransformedLaserScan (sensor_data.h):
    - Biểu diễn toàn bộ quét laser với:
        - Danh sách points (std::vector<ScanPoint>).
        - Thay đổi odometry (d_x, d_y, d_yaw).
        - Chất lượng quét (quality, từ 0.0 đến 1.0).
    - Được tạo bởi LaserScanObserver từ dữ liệu ROS.
    - Nguồn: Được tạo bởi LaserScanObserver từ dữ liệu ROS.
    - Ứng dụng: Đầu vào chính cho TinyWorld để xử lý quét và cập nhật bản đồ.

b. Trạng thái robot (state_data.h)

- RobotState:
    - Cấu trúc:
        - x, y: Tọa độ Cartesian trong không gian thế giới (mét).
        - theta: Góc hướng (radian).
    - Phương thức:
        - update(d_x, d_y, d_theta): Cộng các thay đổi vào x, y, theta.
        - operator-: Tính chênh lệch giữa hai trạng thái để xác định pose_delta.
    - Vai trò: Lưu trữ và cập nhật vị trí robot trong quá trình định vị.

- World:
    - Mô tả: Lớp template trừu tượng quản lý trạng thái robot (_pose) và bản đồ (MapType).
    - Phương thức chính:
        - update_robot_pose(x, y, theta): Cập nhật vị trí robot.
        - handle_observation(ObservationType): Xử lý dữ liệu quét (trừu tượng).
        - pose(): Truy xuất RobotState.
        - map(): Truy xuất bản đồ.
    - Triển khai: TinyWorld kế thừa World, sử dụng GridMap làm MapType và TransformedLaserScan làm ObservationType.

c. Bản đồ lưới

- GridMap (grid_map.h):
    - Lưu trữ bản đồ dưới dạng ma trận 2D các ô (GridCell).
    - Hỗ trợ:
        - Khởi tạo với GridCellFactory và GridMapParams (chiều rộng, chiều cao, tỷ lệ ô).
        - Cập nhật ô (update_cell) với xác suất chiếm chỗ (Occupancy).
        - Truy xuất xác suất (cell_value) và chuyển đổi tọa độ (world_to_cell, world_cell_bounds).
        - Mở rộng động bản đồ theo lũy thừa 2 khi cần.
- GridCell (grid_cell_factory.h):
    - Lớp trừu tượng biểu diễn một ô, với:
        - value(): Trả về xác suất chiếm chỗ.
        - set_value(Occupancy, quality): Cập nhật xác suất.
- GridCellFactory (grid_cell_factory.h):
    - Lớp trừu tượng tạo ô bằng create_cell(), áp dụng Factory Method pattern.

d. Chiến lược ô lưới

- GridCellStrategy (grid_cell_strategy.h):
  - Container lưu trữ ba chiến lược:
    - GridCellFactory: Tạo ô.
    - ScanCostEstimator: Đánh giá chi phí quét.
    - CellOccupancyEstimator: Ước lượng xác suất chiếm chỗ.
- CellOccupancyEstimator (cell_occupancy_estimator.h):
  - Lớp trừu tượng định nghĩa giao diện estimate_occupancy để tính xác suất chiếm chỗ (Occupancy) dựa trên tia laser (Beam) và ranh giới ô (Rectangle).
- AreaOccupancyEstimator (area_occupancy_estimator.h):
  - Ước lượng xác suất dựa trên diện tích phần ô bị cắt bởi tia laser.
  - Xoay tia 90 độ nếu ô chiếm chỗ để mô phỏng vùng ảnh hưởng.
- ConstOccupancyEstimator (const_occupancy_estimator.h):
  - Trả về xác suất cố định (base_occ_prob hoặc base_empty_prob) dựa trên trạng thái chiếm chỗ, bỏ qua thông tin hình học.

e. Khớp quét

- MonteCarloScanMatcher (monte_carlo_scan_matcher.h):
  - Lớp trừu tượng triển khai thuật toán Monte Carlo để khớp quét.
  - Tìm vị trí robot tối ưu bằng cách thử nghiệm ngẫu nhiên và đánh giá chi phí quét.

f. Quản lý môi trường

- LaserScanGridWorld (laser_scan_grid_world.h):
  - Kế thừa World<TransformedLaserScan, GridMap>, quản lý bản đồ (GridMap) và xử lý quét laser.
  - Cập nhật bản đồ bằng cách chiếu các điểm quét và sử dụng GridCellStrategy.

2. Foder robot

Chứa các file để định nghĩa robot trong gazebo, rviz, config, điều khiển và launch để khởi chạy tất cả thuật toán Tiny SLAM

2.1 Định nghĩa robot Robot.urdf

Chứa model robot và các plugin cho việc điều khiển, trong dự án này sử dụng cơ chế điều khiển diff_drive_controller.

Diff_drive_controller: cho phép điều khiển chuyển động của robot bằng lệnh vận tốc, sử dụng cho bộ có 2 bánh xe chủ động. Dựa trên vận tốc nhận được sẽ thực hiện tính toán vận tốc cho từng bánh xe. Trong phương pháp điều khiển này được tích hợp sẵn để gửi các lệnh qua topic ROS đến gazebo hoặc phần cứng.

2.2 File config

Định nghĩa các controller cho robot, sử dụng *joint_state_controller* để phát thông tin về trạng thái của các joint robot. Thu thập dữ liệu từ các khớp và phát ra thông tin qua các topic

Đặt độ bất định (phương sai) cho:

- Vị trí (pose):

  + Phương sai theo mỗi trục (x, y, z, roll, pitch) = 0.001

  + Phương sai yaw (quay quanh trục z) = 0.01

- Vận tốc (twist):

  + Phương sai theo mỗi trục (vx, vy, vz, vroll, vpitch) = 0.001

  + Phương sai yaw rate (tốc độ quay quanh trục z) = 0.01.

Hai ma trận *pose_covariance_diagonal, twist_covariance_diagonal* lần lượt để hiệp định phương sai cho vị trí và vận tốc của robto, giúp xác định độ không chắc chắn trong ước lượng vị trí và vận tốc khi xây dựng map.

2.3 File launch

Cấu hình cho ROS, khởi động và cấu hình các nodes và tham số trong môi trường mô phỏng. Các nhiệm vụ trong file này:

- Khởi động thế giới gazebo
- Tải mô hình từ urdf
- Spawn robot và gazebo dưới model có tên là robot_description
- Fake_joint_calibration: node này để phát một thông điệp để giả lập việc hiệu chỉnh các khớp của robot
- Tạo một node có tên là "fake_joint_calibration":
  - Dùng gói (package) "rostopic"
  - Chạy file thực thi "rostopic"
  - Tham số truyền vào:

+ Publish (gửi) 1 tin nhắn lên topic "/calibrated"
+ Tin nhắn kiểu std_msgs/Bool, giá trị là true
+ Chỉ gửi 1 lần (--once)

- Hiển thị output lên màn hìnhPublish các thông tin về trạng thái của robot gồm vị trí, góc quay.

- joint_state_publisher_gui cho phép điều chỉnh trạng thái của robto thoogn qua giao diện đồ họa.
- tiny_slam :

+ remap: thực hiện định tuyến tại topic quét laser từ gazebo để nút tiny_slam có thể nhận dữ liệu quét từ cảm biến

+ Modules setup : Trung bình hóa nhiều lần đọc giá trị để bản đồ mượt hơn bằng cách sử dụng “avg”, occupancy_estimator="const" ước lượng xem có vật thể hay không. skip_exceeding_lsr_vals="false" tất cả các dữ liệu thu được laser kể cả các tia bất thường để vẽ map.

Thiết lập các tham số cho bản đồ SLAM:

- cell_type = "avg"
  -> Mỗi ô lưới (cell) lưu giá trị trung bình (average) của các lần đo.
- occupancy_estimator = "const"
  -> Sử dụng bộ ước lượng xác suất chiếm chỗ kiểu hằng số (constant model).
- skip_exceeding_lsr_vals = "false"
  -> Không bỏ qua các giá trị laser scan vượt quá ngưỡng (vẫn xử lý hết).

+ Frame name setup : odom_frame="odom_combined" lấy thông tin vị trí robot theo bản đồ quán tính

+ Map setup: Giới hạn kích thước cho bản đồ khi quét được.

+ Internal constants setup: Các thông số xác suất về trống, có vật thể.

+ Internal scan matcher constants: Các thông số cho bộ ghép quét bằng Monte Carlo, chứa các thông số độ lệch chuẩn

Thiết lập tham số cho thuật toán scan-matching Monte Carlo:

- scmtch_sigma_XY_MonteCarlo = 0.01
  -> Độ lệch chuẩn cho vị trí (x, y) khi tạo mẫu thử = 0.01 (đơn vị mét).
- scmtch_sigma_theta_MonteCarlo = 0.02
  -> Độ lệch chuẩn cho góc quay (theta) khi tạo mẫu thử = 0.02 (đơn vị radian).
- scmtch_limit_of_bad_attempts = 20
  -> Nếu có 20 lần thử liên tiếp mà không cải thiện, thì dừng việc thử thêm.
- scmtch_limit_of_total_attempts = 100
  -> Tổng số lần thử tối đa là 100, dù có cải thiện hay không.

+ Internal constants for ROS: tốc độ publish lên rviz, bộ đệm ghi lại các Tf.

- MapTransformPublisher: duy trì mối quan hệ giữa khung tọa độ odom_combined và map, cho phép robot xác định vị trí của nó trong không gian bản đồ một cách chính xác.
- Control: Sử dụng bàn phím để di chuyển robot
- Bản đồ sẽ được lưu lại trên rviz.

II. Giải thích mô tả thuật toán

1.Thuật toán Monte Carlo Scan Matching

Đây là thuật toán để tìm ra vị trí robot tốt nhất (pose x,y,θ), bằng cách thử nhiều pose ngẫu nhiên rồi so sánh độ khớp giữa laser scan thực tế và bản đồ, cuối cùng là chọn ra tư thế tốt nhất.

[2] "Phiên bản độc lập sử dụng một thuật toán Monte Carlo rất đơn giản để ghép nối quét hiện tại với bản đồ.""Từ dữ liệu odometry hoặc từ vị trí ban đầu, chúng tôi lấy mẫu các vị trí ngẫu nhiên, tính điểm khớp giữa quét và bản đồ, rồi chọn vị trí tốt nhất."

Các bước chi tiết:

Sinh ra các pose ngẫu nhiên quanh vị trí hiện tại của robot(Gaussian noise cho x,y,θ), vị trí hiện tại là vị trí mà robot được đưa vào được cài sẵn.

$$x' = x_0 + \Delta x \ (\Delta x \sim N(0, \sigma_x^2)$$
$$y' = y_0 + \Delta y \ (\Delta y \sim N(0, \sigma_y^2)$$
$$\theta' = \theta_0 + \Delta\theta \ (\Delta\theta \sim N(0, \sigma_\theta^2)$$

Tức là từ vị trí ban đầu sinh ra các pose ngẫu nhiên với các độ lệch lấy trong file launch.

Tính điểm số: với mỗi pose, giả lập scan và so sánh với map hiện tại cuối cùng là tính điểm để thể hiện mức độ khớp.

Chọn pose tốt nhất: Chọn tư thế có điểm số cao nhất để coi là vị trí hiện tại của robot.

Cập nhật map: Dùng pose tốt nhất để laser vào map.

1.1. Xử lý quét laser (LaserScanGridWorld::handle_observation)

- Mục đích: Chiếu các điểm quét laser lên bản đồ và cập nhật xác suất chiếm chỗ.
- Đầu vào:
  - scan: TransformedLaserScan (danh sách điểm quét, chất lượng).
  - pose: RobotState (vị trí robot hiện tại: x, y, theta).
- Đầu ra: Bản đồ được cập nhật.

Giả mã:

Hàm handle_observation(scan):

Lấy pose từ World

Cho mỗi điểm sp trong scan.points:

// Chuyển tọa độ điểm sang khung thế giới

x_world = pose.x + sp.range * cos(sp.angle + pose.theta)

y_world = pose.y + sp.range * sin(sp.angle + pose.theta)

// Cập nhật bản đồ với điểm quét

handle_scan_point(map, pose.x, pose.y, x_world, y_world, sp.is_occupied, scan.quality)

- Giải thích:
  - Mỗi điểm quét (ScanPoint) được chuyển từ tọa độ cực sang tọa độ Cartesian trong khung thế giới, sử dụng vị trí robot (pose).
  - Gọi handle_scan_point để cập nhật các ô trên đường đi của tia laser.

1.2. Cập nhật ô lưới (LaserScanGridWorld::handle_scan_point)

- Mục đích: Cập nhật xác suất chiếm chỗ cho các ô trên đường đi của tia laser.
- Đầu vào:
  - map: GridMap (bản đồ lưới).
  - laser_x, laser_y: Tọa độ cảm biến (thường là pose.x, pose.y).
  - beam_end_x, beam_end_y: Tọa độ điểm cuối tia.
  - is_occ: Trạng thái chiếm chỗ (true nếu chướng ngại).
  - quality: Chất lượng quét.
- Đầu ra: Các ô được cập nhật xác suất chiếm chỗ.

Giả mã:

Hàm handle_scan_point(map, laser_x, laser_y, beam_end_x, beam_end_y, is_occ, quality):

// Chuyển tọa độ sang chỉ số ô

robot_pt = map.world_to_cell(laser_x, laser_y)

obst_pt = map.world_to_cell(beam_end_x, beam_end_y)

// Tạo danh sách ô trên đường thẳng từ robot_pt đến obst_pt

pts = DiscreteLine2D(robot_pt, obst_pt).points()

// Cập nhật ô cuối (chướng ngại)

map.update_cell(pts.back(), Occupancy{is_occ ? 1.0 : 0.0, 1.0}, quality)

// Cập nhật các ô trên đường đi (trống)

Loại bỏ pts.back()

Cho mỗi pt trong pts:

map.update_cell(pt, Occupancy{0.0, 1.0}, quality)

- Giải thích:
  - Sử dụng thuật toán Bresenham (DiscreteLine2D) để liệt kê các ô từ cảm biến đến điểm cuối.
  - Ô cuối được cập nhật là chiếm chỗ (prob_occ = 1.0) nếu is_occ = true, hoặc trống (prob_occ = 0.0) nếu is_occ = false.
  - Các ô trên đường đi được cập nhật là trống (prob_occ = 0.0).
  - GridMap::update_cell gọi GridCell::set_value để lưu xác suất.

1.3. Ước lượng xác suất chiếm chỗ (AreaOccupancyEstimator::estimate_occupancy)

- Mục đích: Tính xác suất chiếm chỗ của ô dựa trên diện tích phần ô bị cắt bởi tia laser.
- Đầu vào:
  - beam: Tia laser (x_st, y_st, x_end, y_end).
  - cell_bnds: Ranh giới ô (Rectangle).
  - is_occ: Trạng thái chiếm chỗ.
- Đầu ra: Occupancy (xác suất chiếm chỗ và chất lượng).

Giả mã:

Hàm estimate_occupancy(beam, cell_bnds, is_occ):

// Tạo tia

Nếu is_occ:

// Xoay 90 độ quanh điểm cuối để mô phỏng vùng chướng ngại

```
        x_delta = beam.y_st - beam.y_end
        y_delta = beam.x_end - beam.x_st
        ray = Ray(beam.x_end, beam.y_end, x_delta, y_delta)
    Ngược lại:
        // Tia từ cảm biến đến điểm cuối
        x_delta = beam.x_end - beam.x_st
        y_delta = beam.y_end - beam.y_st
ray = Ray(beam.x_st, beam.y_st, x_delta, y_delta)
    // Tìm giao điểm với các cạnh ô
    intersections = []
    Cho mỗi cạnh trong {cell_bnds.top, cell_bnds.bot, cell_bnds.left,
  cell_bnds.right}:
        Nếu cạnh ngang (top hoặc bot):
            t = (cạnh.y - ray.y_st) / ray.y_delta
            Nếu 0 <= t <= 1:
                x_inter = ray.x_st + t * ray.x_delta
                Nếu cell_bnds.left <= x_inter <= cell_bnds.right:
                    Thêm Intersection(cạnh, x_inter, cạnh.y) vào intersections
        Ngược lại (left hoặc right):
            t = (cạnh.x - ray.x_st) / ray.x_delta
            Nếu 0 <= t <= 1:
                y_inter = ray.y_st + t * ray.y_delta
                Nếu cell_bnds.bot <= y_inter <= cell_bnds.top:
                    Thêm Intersection(cạnh, cạnh.x, y_inter) vào intersections

    // Tính diện tích phần ô bị cắt
    Nếu intersections rỗng:
```

```
            // Trường hợp biên: Tia song song hoặc không cắt
            chunk_area = cell_bnds.area() / 2
        Ngược lại nếu intersections.size() == 4:
    // Trường hợp tia cắt đường chéo ô
            chunk_area = cell_bnds.area() / 2
        Ngược lại nếu intersections.size() == 2:
            // Xác định loại hình
            Nếu một giao điểm ngang và một giao điểm dọc:
                // Tam giác
                corner = chọn góc cơ sở dựa trên vị trí giao điểm
                area = 0.5 * |intersections[0].x - corner.x| * |intersections[1].y - corner.y|
            Ngược lại:
                // Hình thang
                base_sum = |intersections[0].x - corner.x| + |intersections[1].x - corner.x|
                area = 0.5 * (cell_bnds.top - cell_bnds.bot) * base_sum
            // Trường hợp biên: Điều chỉnh nếu ô chiếm chỗ
            Nếu is_occ và cả hai giao điểm cùng phía với beam:
                area = cell_bnds.area() - area
        // Tính xác suất
        area_rate = chunk_area / cell_bnds.area()
    Nếu is_occ:
        Trả về Occupancy{area_rate, 1.0}
    Ngược lại:
        Nếu area_rate > 0.5:
                area_rate = 1 - area_rate
        Trả về Occupancy{base_empty_prob(), area_rate}
```

- Giải thích:

- Tạo tia:
    - Nếu is_occ = true, tia được xoay 90 độ để mô phỏng vùng ảnh hưởng của chướng ngại, đảm bảo xác suất phản ánh đúng khu vực bị ảnh hưởng.
    - Nếu is_occ = false, tia đại diện cho đường đi trống từ cảm biến đến điểm cuối.
- Tìm giao điểm:
    - Sử dụng phương trình tham số để tính giao điểm với các cạnh ô.
    - Chỉ giữ các giao điểm hợp lệ (nằm trong đoạn tia và ranh giới ô).
- Tính diện tích:
    - Trường hợp không giao điểm: Dùng area / 2, phù hợp khi tia song song với cạnh ô hoặc đi qua trung tâm.
    - Trường hợp 4 giao điểm: Xảy ra khi tia cắt đường chéo ô, dùng area / 2.
    - Trường hợp 2 giao điểm:
        - Tam giác: Dựa trên giao điểm và góc cơ sở, tính diện tích bằng công thức 0.5 * base * height.
        - Hình thang: Tính diện tích bằng (base1 + base2) * height / 2.
        - Nếu is_occ = true và giao điểm cùng phía, lấy phần diện tích còn lại để mô phỏng vùng chướng ngại.
- Tính xác suất:
    - Tỷ lệ diện tích (area_rate) được dùng trực tiếp nếu is_occ = true.
    - Nếu is_occ = false, đảo ngược area_rate khi lớn hơn 0.5 để ưu tiên trạng thái trống.
- Chất lượng ước lượng: Luôn là 1.0, giả định độ tin cậy tối đa.

o Trường hợp biên:

- Tia song song với cạnh ô: Không có giao điểm, dùng diện tích mặc định (area / 2).
- Tia đi qua góc ô: Giao điểm trùng, diện tích tam giác có thể rất nhỏ, cần kiểm tra để tránh chia cho 0.

        - Ô nhỏ hơn điểm cuối: Tỷ lệ diện tích được giới hạn để đảm bảo area_rate nằm trong [0, 1].
    - Vai trò:
        - Cung cấp ước lượng chính xác, đặc biệt hiệu quả cho các ô gần ranh giới chướng ngại.
        - Tốn tài nguyên hơn ConstOccupancyEstimator do cần tính toán hình học.

1.4. Ước lượng xác suất chiếm chỗ đơn giản (ConstOccupancyEstimator::estimate_occupancy)

- Mục đích: Trả về xác suất cố định dựa trên trạng thái chiếm chỗ.
- Đầu vào: beam, cell_bnds, is_occ.
- Đầu ra: Occupancy.

Giả mã:

Hàm estimate_occupancy(beam, cell_bnds, is_occ):

Nếu is_occ:

Trả về Occupancy{base_occ_prob(), 1.0}

Ngược lại:

Trả về Occupancy{base_empty_prob(), 1.0}

- Giải thích:
    - Bỏ qua beam và cell_bnds, chỉ dựa vào is_occ để trả về xác suất cơ bản.
    - Nhanh nhưng kém chính xác so với AreaOccupancyEstimator.

1.5. Khớp quét Monte Carlo (MonteCarloScanMatcher::process_scan)

- Mục đích: Tìm vị trí robot tối ưu bằng cách thử nghiệm ngẫu nhiên và đánh giá chi phí quét.
- Đầu vào:
    - init_pose: Vị trí ban đầu (RobotState).
    - scan: TransformedLaserScan.
    - map: GridMap.
- Đầu ra: pose_delta (thay đổi vị trí) và chi phí thấp nhất.

Giả mã:

Hàm process_scan(init_pose, scan, map):

Thông báo observer: on_matching_start(init_pose, scan, map)

Khởi tạo:

failed_tries = 0

total_tries = 0

min_scan_cost = vô cực

optimal_pose = init_pose

min_scan_cost = estimate_scan_cost(optimal_pose, scan, map)

Thông báo observer: on_scan_test(optimal_pose, scan, min_scan_cost)

Trong khi failed_tries < failed_tries_limit và total_tries < total_tries_limit:

total_tries += 1

sampled_pose = optimal_pose

sample_pose(sampled_pose) // Thêm nhiễu ngẫu nhiên

sampled_scan_cost = estimate_scan_cost(sampled_pose, scan, map)

Thông báo observer: on_scan_test(sampled_pose, scan, sampled_scan_cost

Nếu min_scan_cost <= sampled_scan_cost:

failed_tries += 1

Tiếp tục

min_scan_cost = sampled_scan_cost

optimal_pose = sampled_pose

failed_tries = on_estimate_update(failed_tries, failed_tries_limit)

Thông báo observer: on_pose_update(optimal_pose, scan, min_scan_cost)

pose_delta = optimal_pose - init_pose

Thông báo observer: on_matching_end(pose_delta, min_scan_cost)

Trả về min_scan_cost

- Giải thích:

- Thử nghiệm ngẫu nhiên các vị trí (sampled_pose) quanh optimal_pose.
- Đánh giá chi phí quét bằng ScanCostEstimator (thường là TinyScanCostEstimator).
- Cập nhật vị trí tốt nhất nếu tìm thấy chi phí thấp hơn, dừng khi đạt giới hạn thử nghiệm.

1.6. Tích hợp với ROS

- Nhập liệu:
  - LaserScanObserver nhận sensor_msgs::LaserScan (từ topic LIDAR) và tf::StampedTransform (từ topic odometry).
  - Chuyển đổi thành TransformedLaserScan để sử dụng nội bộ.
- Xử lý:
  - TinySlamFascade và tiny_slam_node.cpp khởi tạo các thành phần (TinyWorld, TinyScanMatcher, GridCellStrategy) với tham số từ file launch (như ~map_width, ~occ_prob).
- Xuất liệu:
  - RvizGridViewer xuất bản:
    - Bản đồ (nav_msgs::OccupancyGrid) từ GridMap.
    - Vị trí robot (geometry_msgs::Pose) từ RobotState.
    - Vị trí thử nghiệm trong scan matching qua tf (bằng PoseScanMatcherObserver).
- Cấu hình:
  - Tham số launch file: map_width, map_height, meters_per_cell, occ_prob, empty_prob, scmtch_limit_of_bad_attempts, scmtch_limit_of_total_attempts, v.v.

2.Cơ chế xây dựng và update map dùng tiny slam

2.1.Xác định vị trí robot ở đâu (Scan Matching)

Mỗi lần nhận được LaserScan, TinySLAM:
Dùng thuật toán Monte Carlo Scan Matching:Random thử rất nhiều vị trí mới xung quanh vị trí hiện tại (thêm nhiễu sigma_XY, sigma_theta).
Với mỗi vị trí thử, nó dự đoán laser scan sẽ quét như thế nào trên bản đồ hiện có.
So sánh scan thật với scan giả này -> chấm điểm ("score").
Chọn vị trí có score tốt nhất làm vị trí mới.

Nhờ vậy, robot "bám" vào bản đồ đang xây mà tự xác định vị trí chính xác.

2.2.Update bản đồ từ dữ liệu Laser

Khi biết robot đang ở đâu rồi, TinySLAM update bản đồ bằng từng tia laser:
Với mỗi tia laser:

Tính toán đường đi của tia laser từ robot đến điểm quét được.

Điền thông tin vào bản đồ như sau:

Các ô đi qua (nhưng không va chạm): cập nhật là trống (tăng xác suất "empty").

Ô cuối cùng (nơi laser đụng vào vật): cập nhật là bị chiếm chỗ (tăng xác suất "occupied").

Công thức cập nhật

```
// If cell is hit
occupancy = occupancy + (1.0 - occupancy) * base_occupied_prob;
// If cell is traversed (empty)
occupancy = occupancy * (1.0 - base_empty_prob);
```

base_occupied_prob = 0.95, base_empty_prob = 0.01 được cài đặt trong file launch

2.3.Cập nhật lên rviz để hiển thị map quét được.

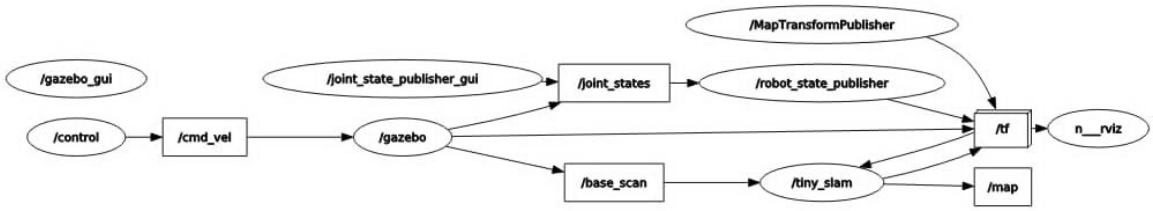
3. Các đặc điểm thông minh

- Trung bình hóa (vì cell_type="avg"): Nếu một ô bị quét đi quét lại nhiều lần, TinySLAM không overwrite mà làm trung bình xác suất lại, tránh nhiễu.

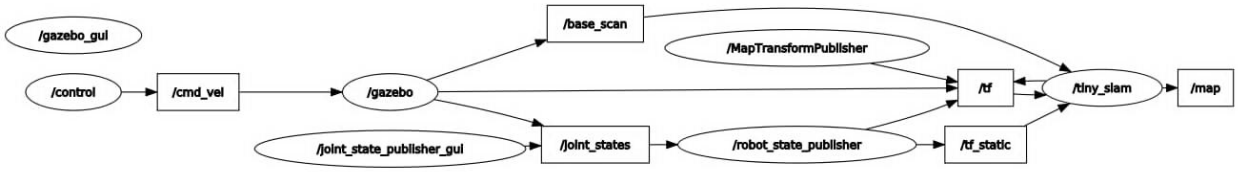
- Hole filtering (hole_width=0.05): Nếu giữa 2 tia laser gần nhau có lỗ nhỏ hơn 5cm -> coi như liên tục, bỏ qua lỗ.

- Không bỏ dữ liệu bất thường (skip_exceeding_lsr_vals=false): Vẫn tận dụng được các scan lạ (ví dụ có vật rất xa hoặc sensor bị nhiễu).

III. Cấu trúc của dự án



Hình 4 . Sơ đồ luồng hoạt động tổng quát của dự án



Hình 5 . Sơ đồ luồng hoạt động chi tiết

Topics chính:

- /cmd_vel: tốc độ điều khiển (velocity command) — thường chứa lệnh di chuyển (linear, angular).
- /base_scan: dữ liệu từ laser scanner (ví dụ như LIDAR).
- /joint_states: trạng thái khớp (joint) của robot — ví dụ vị trí, tốc độ, mô-men.
- /tf và /tf_static: hai topic cực kỳ quan trọng để quản lý hệ thống biến đổi tọa độ (transformations giữa các frame robot).
- /map: bản đồ kết quả sau quá trình SLAM.

Node cụ thể:

- /gazebo: môi trường mô phỏng robot vật lý.
- /gazebo_gui: giao diện người dùng cho Gazebo.
- /tiny_slam: node thực hiện thuật toán SLAM gọn nhẹ (tiny_slam).
- /robot_state_publisher: đọc từ /joint_states và xuất ra các frame vào /tf.

- /joint_state_publisher_gui: giao diện để người dùng chỉnh joint state thủ công.
- /MapTransformPublisher: giúp publish các biến đổi map liên quan (khá chuyên dụng trong SLAM).
- /control: node điều khiển robot.

Luồng dữ liệu tổng thể:

- /control gửi lệnh điều khiển vận tốc đến /cmd_vel.
- /cmd_vel được truyền cho /gazebo -> để mô phỏng chuyển động robot trong môi trường ảo.
- /gazebo xuất ra:
    - /base_scan: dữ liệu cảm biến (giống LIDAR quét môi trường).
    - /joint_states: trạng thái các khớp robot.
- /joint_states được đưa đến:
    - /robot_state_publisher -> publish thông tin vị trí các frame trên /tf.
    - /joint_state_publisher_gui -> giao diện để người dùng xem/sửa joint state.
- /base_scan và /tf được đưa đến /tiny_slam để xử lý SLAM:
    - Tiny Slam xử lý bản đồ hóa dựa trên scan laser và vị trí robot.
    - Sau đó xuất bản đồ ra /map.
- /MapTransformPublisher cũng hỗ trợ publish thêm thông tin transform cần thiết cho việc SLAM.
- TinySLAM lấy laser + tf -> dò tìm vị trí tốt nhất -> cập nhật bản đồ /map.
- So với sơ đồ tổng quát:

  /tf_static được thêm -> giúp cố định mối liên kết cứng giữa các frame (ví dụ laser scanner không thay đổi vị trí so với base_link).

  Dòng dữ liệu rõ ràng hơn ở chỗ nguồn TF đến từ nhiều nơi, không chỉ robot_state_publisher.

Một cách hình dung đơn giản nhất: Robot di chuyển theo lệnh - Nhìn xung quanh bằng laser - Tự đoán lại mình đang ở đâu trong map - Vẽ lại bản đồ theo dự đoán mới.

IV. Thực hiện mô phỏng và fix lỗi

Các lỗi trong dự án:

- Lỗi map khi quét được bị nghiêng do ước lượng góc khi quay chưa chính xác, tuy đã cải thiện rất nhiều so với khi bắt đầu chạy
- Lỗi vị trí của robot do nhiễu chưa được cải thiện

V. Ưu điểm và hạn chế của tinyslam

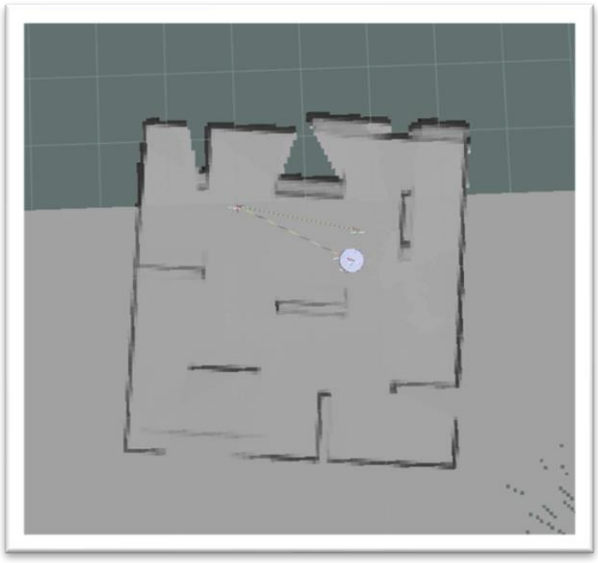
Ưu điểm	Hạn chế
Đơn giản, dễ thay đổi	Độ chính xác không cao bằng thuật toán nặng
Nhẹ và linh hoạt, không cần tính toán gradient hay hệ phương trình như ICP	Nếu map quá trống (không có tường, vật thể rõ ràng) thì scan matching khó tìm đúng vị trí.
Chịu được lỗi odometry vì scan matching liên tục, nếu odometry trôi thì vẫn kéo lại được	Nếu robot bị lạc (khác nhiều với map), TinySLAM không tự tìm lại vị trí tốt như particle filter đúng nghĩa (ví dụ AMCL).
Không cần bản đồ có sẵn	Nếu laser bị lỗi, đo nhiễu nhiều Matching dễ sai
Dùng cho robot trong môi trường hẹp	TinySLAM không phát hiện vòng lặp (loop closure) như các hệ SLAM lớn (Cartographer, GMapping, ORB-SLAM).

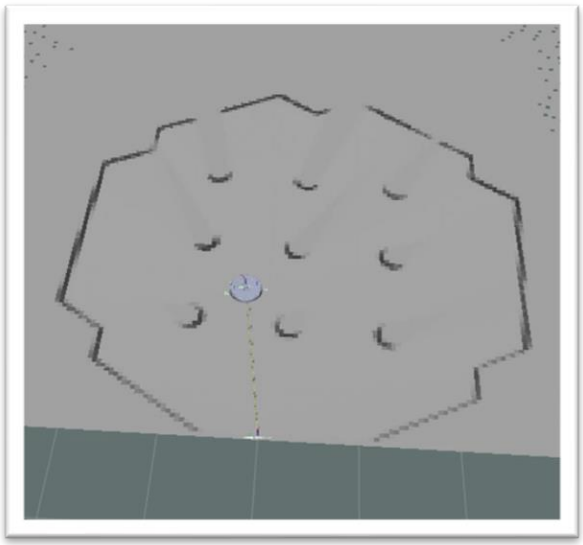
Tóm lại: TinySLAM là một thuật toán scan-matching-based SLAM, cực nhẹ, đơn giản, chạy nhanh, nhưng đánh đổi bằng độ chính xác và khả năng xử lý các tình huống khó phức tạp. [2]"*We developed a simple and efficient algorithm which is able to perform SLAM using data from a laser sensor. Our work proves that simple things could give good results.*" Có nghĩa là "Chúng tôi đã phát triển một thuật toán đơn giản và hiệu quả, có khả năng thực hiện SLAM bằng dữ liệu từ cảm biến laser. Công trình của chúng tôi chứng minh rằng những điều đơn giản cũng có thể mang lại kết quả tốt."

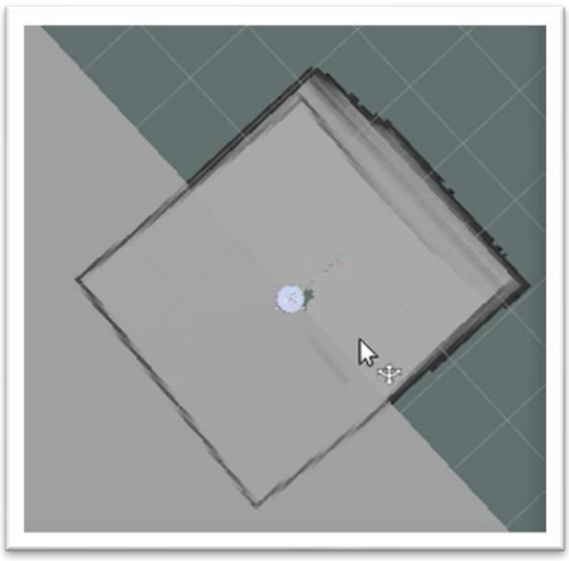
Chương 3. Kết quả

I.Kết quả

Các map đã quét được:







II.Đánh giá map quét được

Ưu điểm nổi bật:

- Hình dạng môi trường tổng thể được quét đúng: các tường, các vật thể cố định như cột trong phòng được nhận diện rõ, không mất hình. Thêm vật thể mới vào cũng có thể xác định chính xác các góc, cạnh.
- Biên tường sắc nét, ít nhiễu, chứng tỏ TinySLAM làm tốt việc scan matching và cập nhật map.
- Khả năng phát hiện vật thể nhỏ (như các cột nhỏ) vẫn đảm bảo tương đối chính xác, dù đây vốn là điểm yếu tự nhiên của các thuật toán nhẹ như TinySLAM.

- Tốc độ xây dựng map nhanh, mượt, ít bị gián đoạn.

Hạn chế còn tồn tại:

- Toàn bộ map bị nghiêng nhẹ, thể hiện yaw drift (lệch góc quay) - đặc điểm khá phổ biến khi dùng TinySLAM vì chỉ dựa vào scan matching và không có loop closure.
- Một số biên vật thể nhỏ còn răng cưa do laser scan không đều hoặc thiếu smoothing filter.
- Không có cơ chế tự phục hồi khi robot bị lệch vị trí quá xa và khi hoạt động lâu (do bản chất TinySLAM không dùng particle filter đúng nghĩa).

III. Câu hỏi và trả lời của nhóm khác

Câu hỏi đặt ra cho nhóm 6:

- Câu hỏi 1: Thuật toán GMapping là gì và nó hoạt động dựa trên nguyên lý nào?
- Câu hỏi 2: Tại sao việc sử dụng teleoperation keyboard không phải là giải pháp tối ưu cho việc khám phá bản đồ tự động?
- Câu hỏi 3: Thước đo nào được sử dụng để đánh giá độ chính xác của các thử nghiệm quét bản đồ?

Câu hỏi của nhóm 2:

- Câu 1: Tại sao TinySLAM sử dụng Monte Carlo Scan Matching thay vì thuật toán ICP hoặc EKF-SLAM để xác định vị trí robot, và điều này ảnh hưởng gì đến khả năng phục hồi khi robot bị lạc (lost)?
- Câu 2: Trong mô hình ROS node của dự án TinySLAM, tại sao cần phải có một node riêng MapTransformPublisher thay vì chỉ rely vào tf tree thông thường? Và nếu bỏ MapTransformPublisher, hệ thống sẽ gặp lỗi gì?
- Câu 3: Trong phần mô tả cấu trúc của dự án, báo cáo có nêu rõ các thành phần chính của hệ thống TinySLAM. Bạn có thể phân tích vai trò của từng thành phần trong việc đảm bảo hiệu suất tổng thể của hệ thống và cách chúng tương tác với nhau không?

Trả lời câu hỏi của nhóm 2:

Câu 1: TinySLAM sử dụng Monte Carlo Scan Matching (MCSM) thay vì ICP hay EKF-SLAM vì:

- Đơn giản và nhẹ hơn rất nhiều: TinySLAM cần chạy trên robot tài nguyên thấp, nên tránh các thuật toán cần giải hệ phương trình phức tạp như ICP hoặc cập nhật ma trận hiệp phương sai như EKF.
- Không cần tính gradient hay đạo hàm Jacobian như ICP.
- Chịu đựng lỗi odometry tốt: vì mỗi lần scan đều làm matching lại, nếu odometry trôi thì TinySLAM vẫn tự sửa được theo scan mới.
- Tuy nhiên, do chỉ dựa vào scan hiện tại mà không có bước re-localization mạnh như particle filter đúng nghĩa, nên nếu robot bị lạc xa map, TinySLAM không tự tìm lại được vị trí như AMCL hoặc FastSLAM.
- Ảnh hưởng: TinySLAM sẽ "chết" nếu robot bị trôi ra xa bản đồ ban đầu mà không có đủ vật thể để scan matching tốt.

Câu 2: Bình thường, tf tự động truyền transform giữa các frame như odom -> base_link, map -> odom.

- TinySLAM không tự publish transform map -> odom_combined, trong khi cần thông tin này để:
    - RViz hiển thị robot đúng vị trí trên bản đồ (/map frame).
    - Navigation Stack (nếu dùng) biết cách biến đổi tọa độ từ map->robot.
- MapTransformPublisher có nhiệm vụ:
    - Định kỳ tính và gửi transform từ /map đến /odom_combined.
    - Thường assume rằng pose của robot trong map (do TinySLAM tính) và odometry từ gazebo là đủ gần.
- Nếu bỏ MapTransformPublisher:
    - RViz sẽ không biết "gắn" robot vào bản đồ.
    - Các thuật toán như move_base, planner sẽ không hoạt động vì thiếu frame transform map-> odom.
    - Mặc định RViz sẽ báo lỗi: "No transform from map to odom".

=> MapTransformPublisher là cầu nối cần thiết để TinySLAM tích hợp với ROS navigation ecosystem đúng cách.

Câu 3: Các thành phần chính của TinySLAM bao gồm LaserScanObserver, TinyWorld, và TinyScanMatcher. LaserScanObserver chịu trách nhiệm thu thập và chuyển đổi dữ liệu từ cảm biến, trong khi TinyWorld quản lý trạng thái robot và bản đồ. TinyScanMatcher thực hiện khớp quét để xác định vị trí robot. Sự tương tác giữa các thành phần này là rất quan trọng; dữ liệu từ LaserScanObserver được

sử dụng bởi TinyWorld để cập nhật trạng thái, và TinyScanMatcher sử dụng thông tin này để thực hiện khớp quét chính xác.

Tài liệu tham khảo

[1] Huletski, A., Kartashov, D., & Krinkin, K. (2016, September). TinySLAM improvements for indoor navigation. In *2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)* (pp. 493-498). IEEE.

[2] Steux, B., & El Hamzaoui, O. (2010, December). tinySLAM: A SLAM algorithm in less than 200 lines C-language program. In *2010 11th International Conference on Control Automation Robotics & Vision* (pp. 1975-1979). IEEE.

[3] https://wiki.ros.org/

[4] SLAM Course - Cyrill Stachniss