HW03 Robotics Technology

Assignment: Map Loading, Localization, and Path Planning

Narjes Ghazanfari . 40110917

February 12, 2026

Contents

List of Figures

List of Equations

1 Introduction

This report presents a complete navigation system for mobile robots, implemented in ROS 2. The system comprises three main components: map representation, probabilistic localization, and path planning. These components work together to enable autonomous navigation in known environments.

The implementation addresses three key questions:

- **Question 1**: Loading and publishing 2D occupancy grid maps
- **Question 2**: Global localization using particle filters
- **Question 3**: Path planning using the A* algorithm

Each component is essential for robot autonomy: the map provides environmental representation, localization determines the robot's position, and path planning generates collision-free trajectories to goal locations.

2 Question 1: Map Loading and Publishing

2.1 Overview

The map loading system reads 2D occupancy grid maps from disk and publishes them to the ROS network. Maps are stored in standard PGM (Portable Gray Map) format with YAML metadata files containing resolution and origin information.

2.2 Theoretical Background

2.2.1 Occupancy Grid Representation

An occupancy grid divides the environment into a grid of cells, where each cell stores occupancy probability:

$$p(m_{i,j}) = \begin{cases} 100 & \text{occupied} \\ 0 & \text{free} \\ -1 & \text{unknown} \end{cases} \tag{1}$$

The map is characterized by:

- Resolution r: size of each cell in meters
- Dimensions: width $\times$ height in cells
- Origin: position of the bottom-left cell in world coordinates

2.2.2 Coordinate Conversion

Converting from grid cell (i, j) to world coordinates (x_w, y_w):

$$\begin{bmatrix} x_w \\ y_w \end{bmatrix} = \begin{bmatrix} i \cdot r \\ j \cdot r \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{2}$$

where (x_0, y_0) is the map origin.

2.2.3 PGM Processing

PGM images store pixel intensities from 0 (black) to 255 (white). These are converted to occupancy values using thresholds:

$$\text{occupancy}(p) = \begin{cases} 100 & \text{if } p/255 < 0.65 \\ 0 & \text{if } p/255 > 0.25 \\ -1 & \text{otherwise} \end{cases} \tag{3}$$

Dark pixels represent obstacles, bright pixels represent free space.

2.3 Implementation

2.3.1 Node Structure

```
class MapPublisher(Node):
    def __init__(self):
        super().__init__('map_pub_node')

        qos_profile = QoSProfile(
            depth=1,
            durability=QoSDurabilityPolicy.TRANSIENT_LOCAL,
            reliability=QoSReliabilityPolicy.RELIABLE
        )

        self.map_pub = self.create_publisher(OccupancyGrid, '/map
           ', qos_profile)
        self.occupancy_grid = self.load_map()
        self.timer = self.create_timer(1.0, self.publish_map)
```

Listing 1: Map publisher initialization

The QoS settings ensure that late-joining subscribers receive the map even if they start after the publisher.

2.3.2 Map Loading Process

```
def load_map(self):
    pkg_path = get_package_share_directory('robot_localization')
    maps_path = os.path.join(pkg_path, 'maps')

    yaml_file = os.path.join(maps_path, 'my_map.yaml')
    with open(yaml_file, 'r') as f:
        map_metadata = yaml.safe_load(f)

    pgm_file = os.path.join(maps_path, map_metadata['image'])
    image = Image.open(pgm_file).convert('L')
    image_array = np.array(image)

    grid = OccupancyGrid()
    grid.header.frame_id = 'map'
    grid.info.resolution = float(map_metadata['resolution'])
```

```
    grid.info.width = image_array.shape[1]
    grid.info.height = image_array.shape[0]

    origin = map_metadata['origin']
    grid.info.origin.position.x = float(origin[0])
    grid.info.origin.position.y = float(origin[1])
    grid.info.origin.orientation.z = np.sin(origin[2] / 2.0)
    grid.info.origin.orientation.w = np.cos(origin[2] / 2.0)

    image_array = np.flipud(image_array)
    data = []
    for pixel in image_array.flatten():
        normalized = pixel / 255.0
        if normalized < 0.65:
            data.append(100)
        elif normalized > 0.25:
            data.append(0)
        else:
            data.append(-1)

    grid.data = data
    return grid
```

Listing 2: Loading map from files

The vertical flip operation corrects for different coordinate conventions between image files (origin at top-left) and ROS maps (origin at bottom-left).

2.4 Results

The map publisher successfully loads and distributes the occupancy grid map. Figure 1 shows the published map as visualized in RViz.

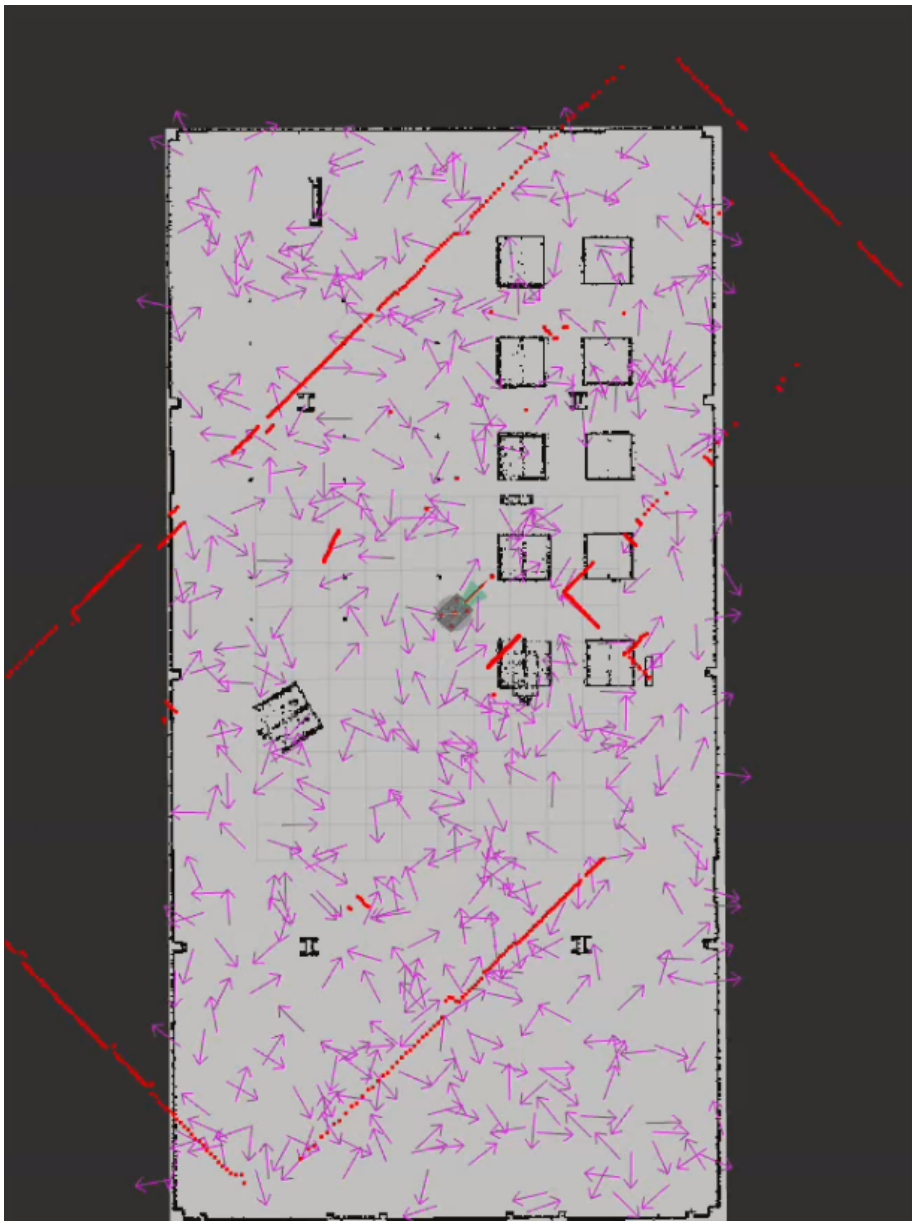

Figure 1: Published occupancy grid map in RViz showing the environment layout

3 Question 2: Particle Filter Localization

3.1 Overview

The particle filter implements Monte Carlo localization to estimate the robot's pose in the known map. The algorithm represents the belief about robot position using a set of weighted particles that are updated based on motion and sensor measurements.

3.2 Algorithm Fundamentals

3.2.1 Particle Representation

The robot's belief is represented by particles:

$$\text{belief}(x_t) \approx \{x_t^{[i]}, w_t^{[i]}\}_{i=1}^{N} \tag{4}$$

where each particle i has a pose $x_t^{[i]} = (x, y, \theta)$ and weight $w_t^{[i]}$.

3.2.2 Motion Model

Robot motion is decomposed into rotation-translation-rotation:

$$\delta_{rot1} = \arctan(\Delta y, \Delta x) - \theta_{old} \tag{5}$$
$$\delta_{trans} = \sqrt{\Delta x^2 + \Delta y^2} \tag{6}$$
$$\delta_{rot2} = \theta_{new} - \theta_{old} - \delta_{rot1} \tag{7}$$

Each component receives noise proportional to the motion magnitude. Updated particle positions are:

$$x_{new} = x_{old} + \delta_{trans} \cos(\theta_{old} + \delta_{rot1}) \tag{8}$$
$$y_{new} = y_{old} + \delta_{trans} \sin(\theta_{old} + \delta_{rot1}) \tag{9}$$
$$\theta_{new} = \theta_{old} + \delta_{rot1} + \delta_{rot2} \tag{10}$$

3.2.3 Measurement Model

For each LiDAR beam with range z, the expected endpoint is:

$$h_x = x + z \cos(\theta + \phi) \tag{11}$$
$$h_y = y + z \sin(\theta + \phi) \tag{12}$$

The likelihood is computed based on distance d to the nearest obstacle:

$$p(z|x, m) = z_{hit} \cdot e^{-d^2/(2\sigma^2)} + z_{rand}/z_{max} \tag{13}$$

3.2.4 Resampling

When particle diversity decreases (measured by effective sample size), resampling draws new particles proportional to their weights. This concentrates particles in high-probability regions while maintaining exploration through small random noise.

3.3 Implementation

3.3.1 Initialization

```
class ParticleFilter(Node):
    def __init__(self):
        super().__init__('particlefilter_node')

        self.declare_parameter('num_particles', 500)
        self.declare_parameter('alpha1', 0.05)
```

```
        self.declare_parameter('alpha2', 0.05)
        self.declare_parameter('alpha3', 0.02)
        self.declare_parameter('alpha4', 0.02)
        self.declare_parameter('likelihood_sigma_m', 0.15)

        self.particles = []
        self.map_data = None
        self.obstacle_kdtree = None

        self.map_sub = self.create_subscription(OccupancyGrid, '/
            map', self.map_callback, map_qos)
        self.odom_sub = self.create_subscription(Odometry, '/ekf/
            odom', self.odom_callback, 10)
        self.scan_sub = self.create_subscription(LaserScan, '/
            scan', self.scan_callback, sensor_qos)

        self.pose_pub = self.create_publisher(
            PoseWithCovarianceStamped, '/amcl_pose', 10)
        self.particle_cloud_pub = self.create_publisher(PoseArray
            , '/particlecloud', 10)
```

Listing 3: Particle filter setup

3.3.2 Particle Initialization

```
def initialize_particles_uniform(self):
    self.particles = []
    for _ in range(self.num_particles):
        idx = np.random.randint(0, len(self.free_cells_cache))
        x, y = self.free_cells_cache[idx]
        th = np.random.uniform(-math.pi, math.pi)
        self.particles.append(Particle(x, y, th, 1.0 / self.
            num_particles))

def build_free_cells_cache(self):
    self.free_cells_cache = []
    for my in range(self.map_height):
        for mx in range(self.map_width):
            if self.map_data[my, mx] == 0:
                wx = mx * self.map_resolution + self.map_origin
                    [0] + self.map_resolution / 2.0
                wy = my * self.map_resolution + self.map_origin
                    [1] + self.map_resolution / 2.0
                self.free_cells_cache.append((wx, wy))
```

Listing 4: Uniform initialization in free space

Free cells are cached for efficient uniform sampling across the map.

3.3.3 Motion Update

```
def motion_update(self, delta):
    trans = float(delta['trans'])
    rot1 = float(delta['rot1'])
    rot2 = float(delta['rot2'])

    for p in self.particles:
        trans_noise = self.alpha3 * abs(trans) + self.alpha4 * (
           abs(rot1) + abs(rot2))
        rot1_noise = self.alpha1 * abs(rot1) + self.alpha2 * abs(
           trans)
        rot2_noise = self.alpha1 * abs(rot2) + self.alpha2 * abs(
           trans)

        trans_noisy = trans + np.random.normal(0.0, max(
           trans_noise, 1e-4))
        rot1_noisy = rot1 + np.random.normal(0.0, max(rot1_noise,
            1e-4))
        rot2_noisy = rot2 + np.random.normal(0.0, max(rot2_noise,
            1e-4))

        p.x += trans_noisy * math.cos(p.theta + rot1_noisy)
        p.y += trans_noisy * math.sin(p.theta + rot1_noisy)
        p.theta = self.normalize_angle(p.theta + rot1_noisy +
           rot2_noisy)
```

Listing 5: Odometry motion model

3.3.4 Measurement Update

```
def measurement_update(self, scan_msg):
    angle_min = float(scan_msg.angle_min)
    angle_inc = float(scan_msg.angle_increment)
    ranges = np.asarray(scan_msg.ranges, dtype=np.float32)

    sigma_cells = max(self.likelihood_sigma_m / self.
       map_resolution, 1e-3)
    inv_2sigma2 = 1.0 / (2.0 * sigma_cells * sigma_cells)

    for p in self.particles:
        log_w = 0.0
        cnt = 0
        for i in range(0, ranges.shape[0], self.laser_subsample):
            z = float(ranges[i])
            if (not math.isfinite(z)) or z < self.min_range or z
               > self.max_range:
                continue
            cnt += 1

            a = p.theta + (angle_min + i * angle_inc)
            hx = p.x + z * math.cos(a)
```

```
            hy = p.y + z * math.sin(a)

            mx, my = self.world_to_map(hx, hy)
            if 0 <= mx < self.map_width and 0 <= my < self.
               map_height:
                dist_cells, _ = self.obstacle_kdtree.query([mx,
                   my], k=1)
                p_hit = math.exp(-float(dist_cells * dist_cells)
                   * inv_2sigma2)
                prob = self.z_hit * p_hit + (self.z_rand / self.
                   max_range)
            else:
                prob = 1e-6
            log_w += math.log(max(prob, 1e-12))

        if cnt > 0:
            p.weight *= math.exp(log_w / cnt)
```

Listing 6: LiDAR measurement update with KD-tree

The KD-tree enables efficient nearest-neighbor queries to find the distance from each beam endpoint to the closest obstacle.

3.3.5 Resampling

```
def resample(self):
    weights = np.array([p.weight for p in self.particles], dtype=
       np.float64)
    cdf = np.cumsum(weights)

    step = 1.0 / self.num_particles
    r = np.random.uniform(0.0, step)
    i = 0
    new_particles = []

    for m in range(self.num_particles):
        u = r + m * step
        while i < self.num_particles - 1 and u > cdf[i]:
            i += 1
        old = self.particles[i]
        x = old.x + np.random.normal(0.0, 0.01)
        y = old.y + np.random.normal(0.0, 0.01)
        th = self.normalize_angle(old.theta + np.random.normal
           (0.0, 0.005))
        new_particles.append(Particle(x, y, th, 1.0 / self.
           num_particles))

    self.particles = new_particles
```

Listing 7: Low variance resampling

3.3.6 Pose Estimation

```
def estimate_pose(self):
    wsum = sum(p.weight for p in self.particles)
    x = sum(p.x * p.weight for p in self.particles) / wsum
    y = sum(p.y * p.weight for p in self.particles) / wsum
    s = sum(math.sin(p.theta) * p.weight for p in self.particles)
        / wsum
    c = sum(math.cos(p.theta) * p.weight for p in self.particles)
        / wsum
    th = math.atan2(s, c)
    return {'x': x, 'y': y, 'theta': th}
```

Listing 8: Weighted mean estimation

Angular averaging uses circular mean to avoid discontinuity at $\pm\pi$.

3.4 Results

The particle filter successfully localizes the robot from global uncertainty. Figures 2 through 8 show the localization process.

In the RViz screenshots where a green arrow is visible, the initial pose was provided using the RViz tool **2D Pose Estimate**. This user-provided initialization helps the particle set focus around the correct region, allowing the filter to converge faster and more reliably to the true location.

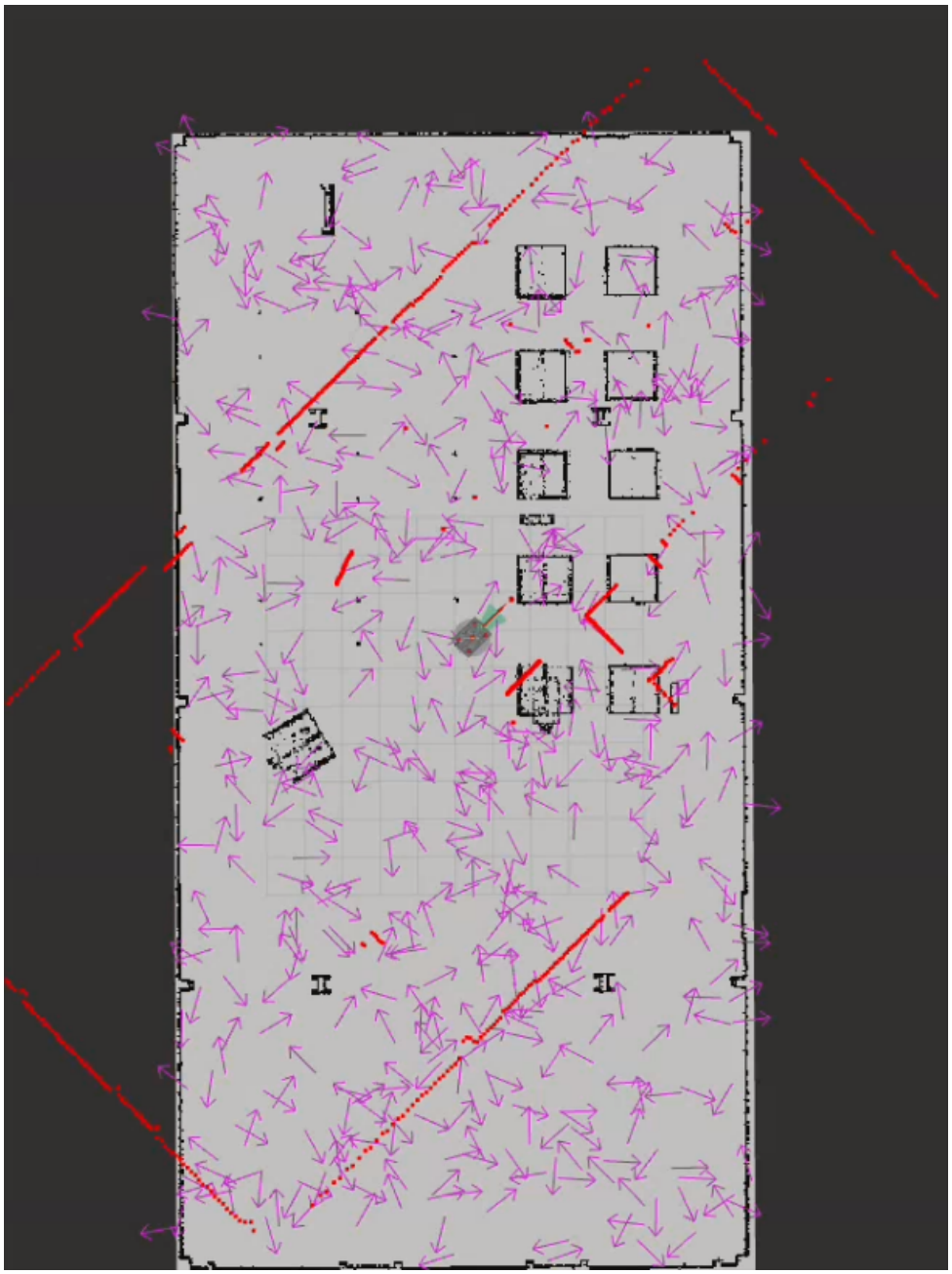

Figure 2: Initial state: particles uniformly distributed across free space

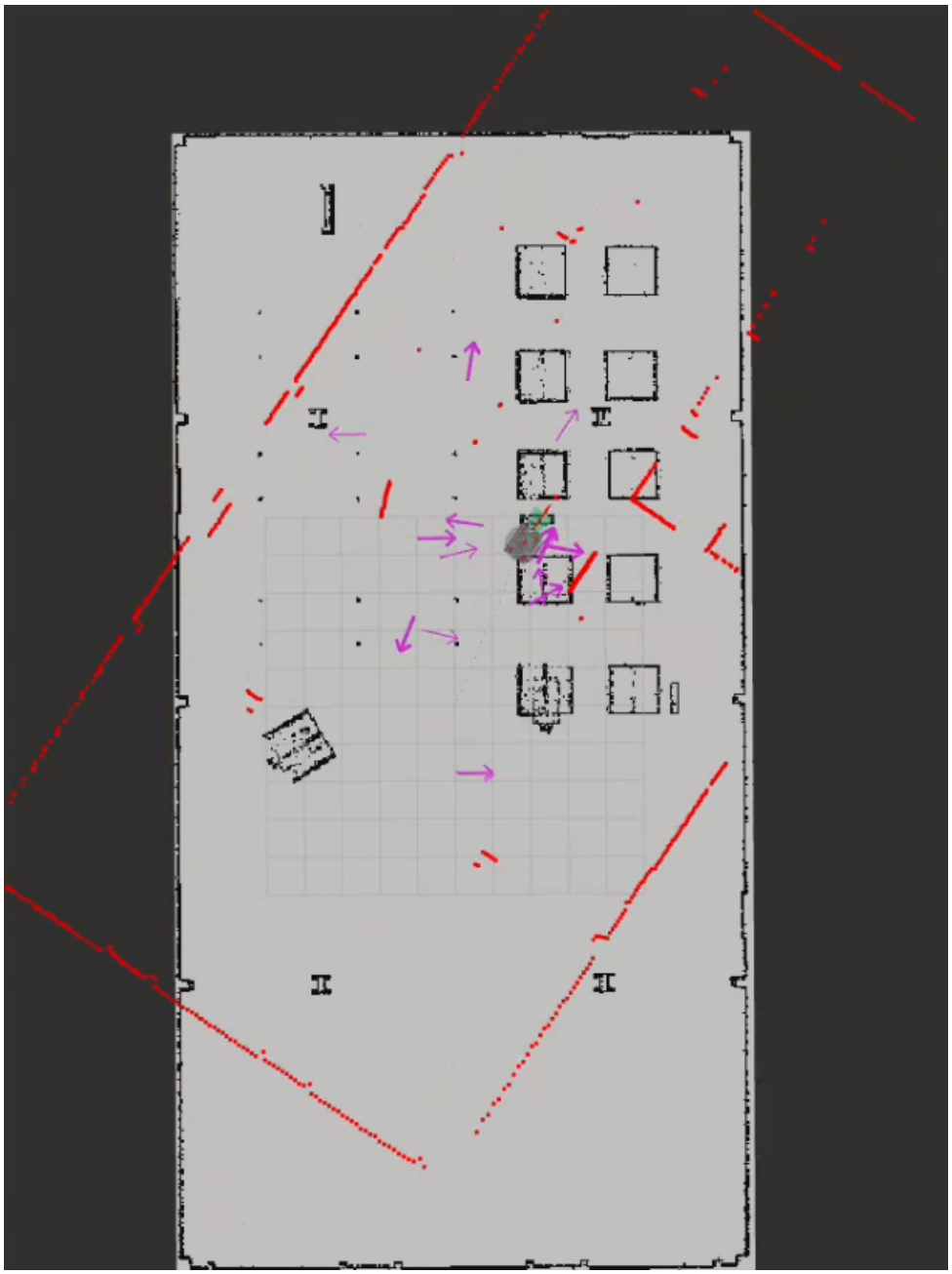

Figure 3: After initial measurements: particles begin to cluster in likely regions

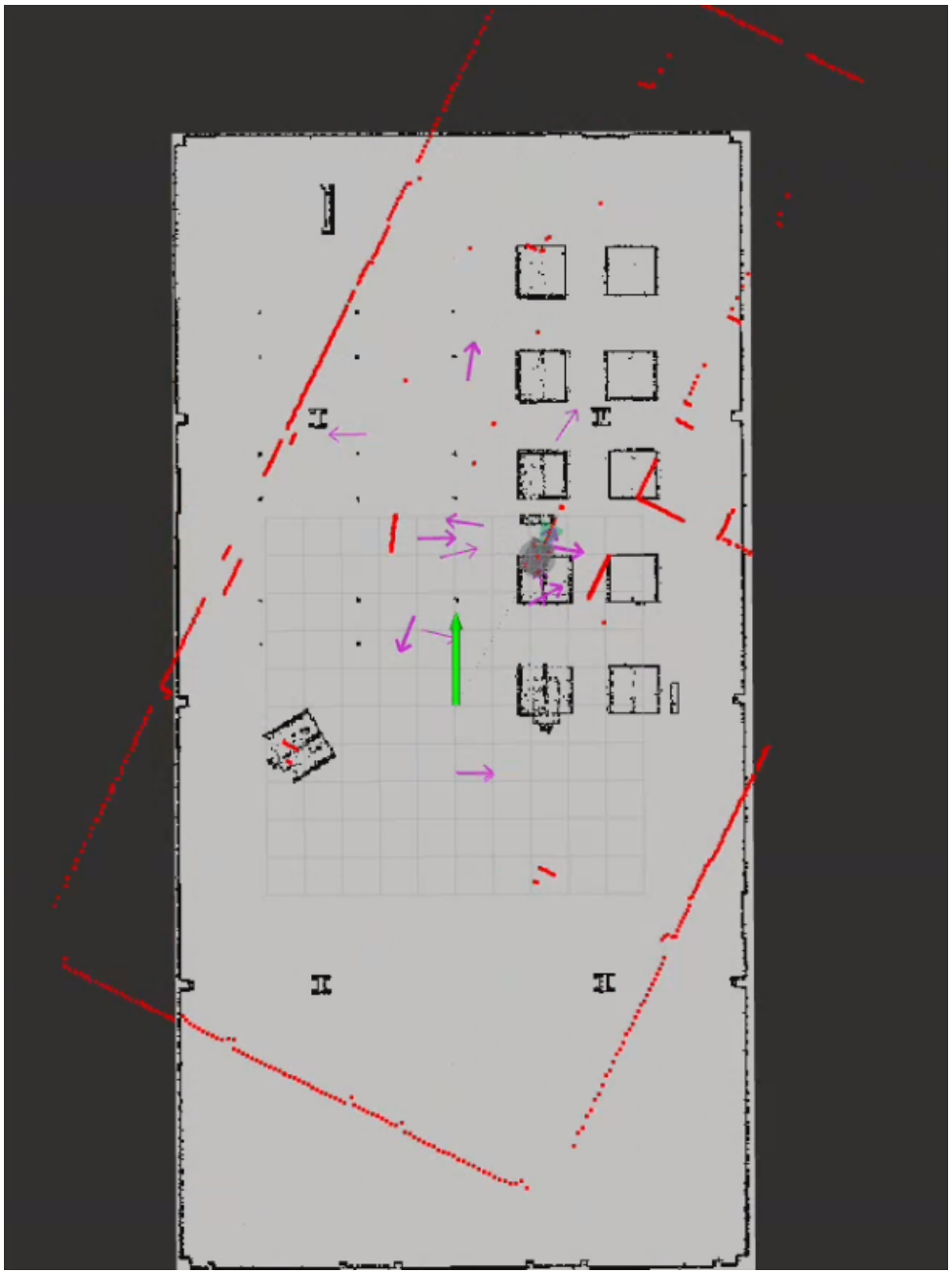

Figure 4: Convergence in progress: particle distribution narrows around true pose

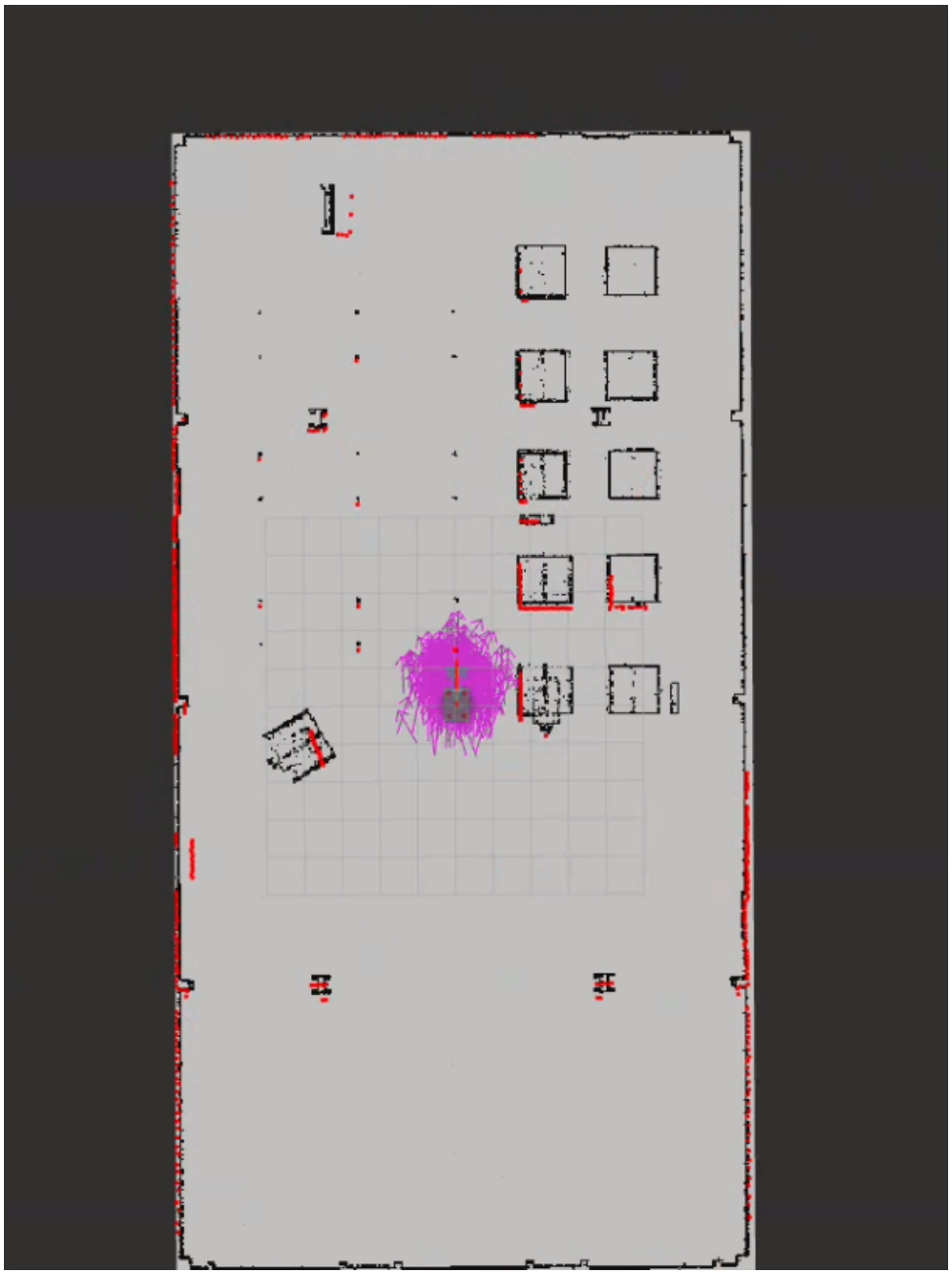

Figure 5: Further refinement as robot moves and collects more measurements

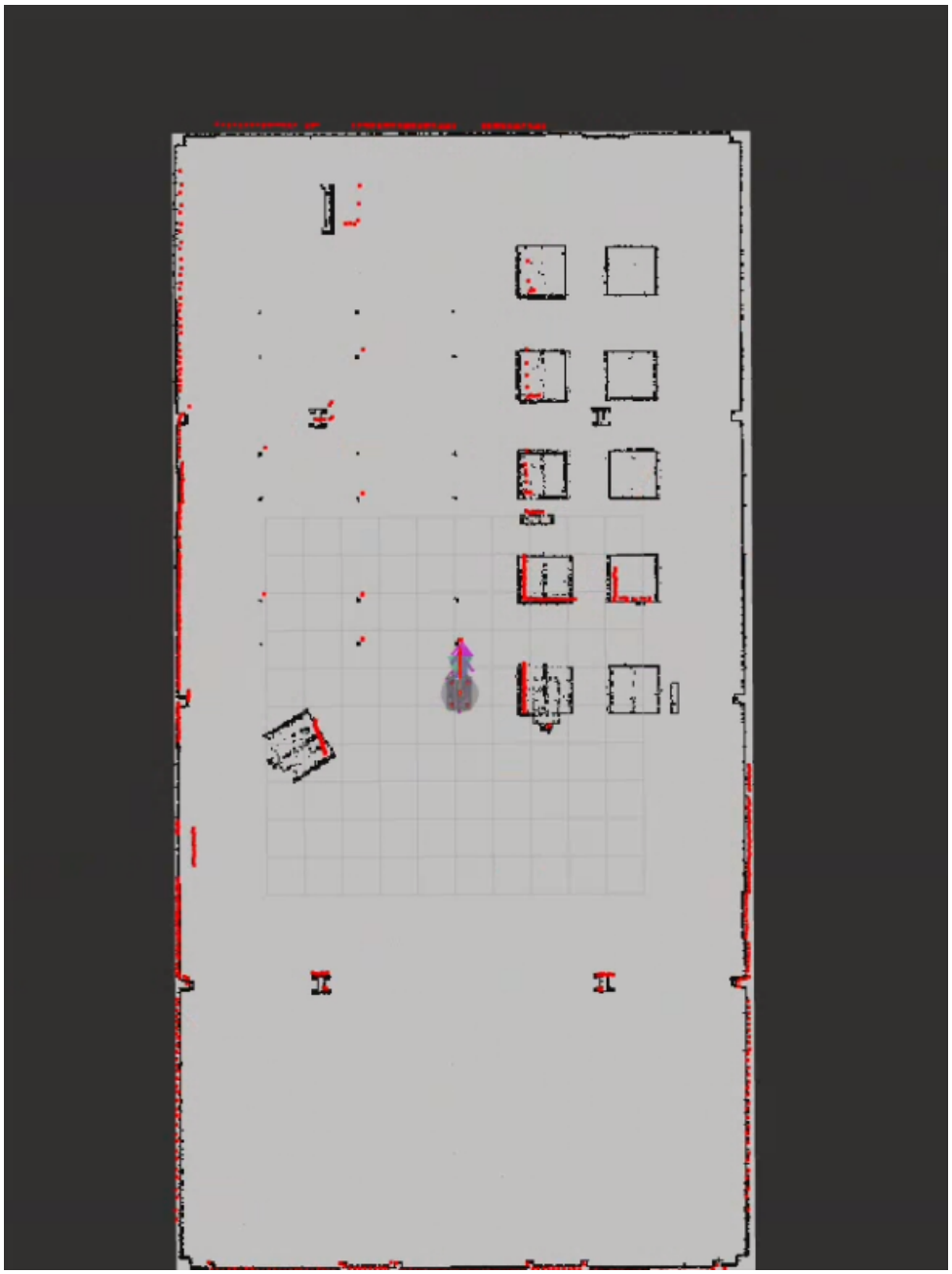

Figure 6: High confidence localization with tight particle cluster

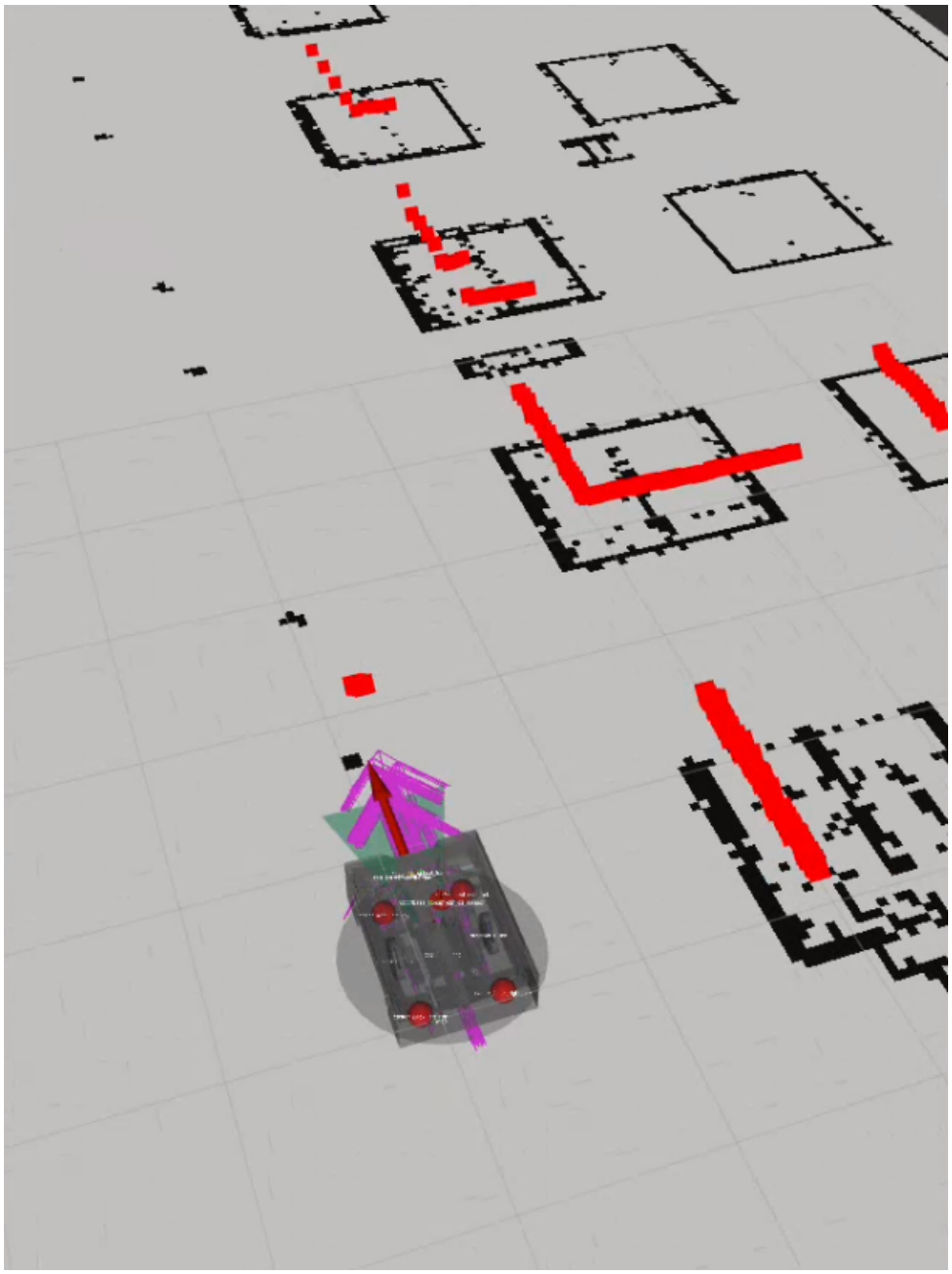

Figure 7: Tracking during motion: particles follow robot movement

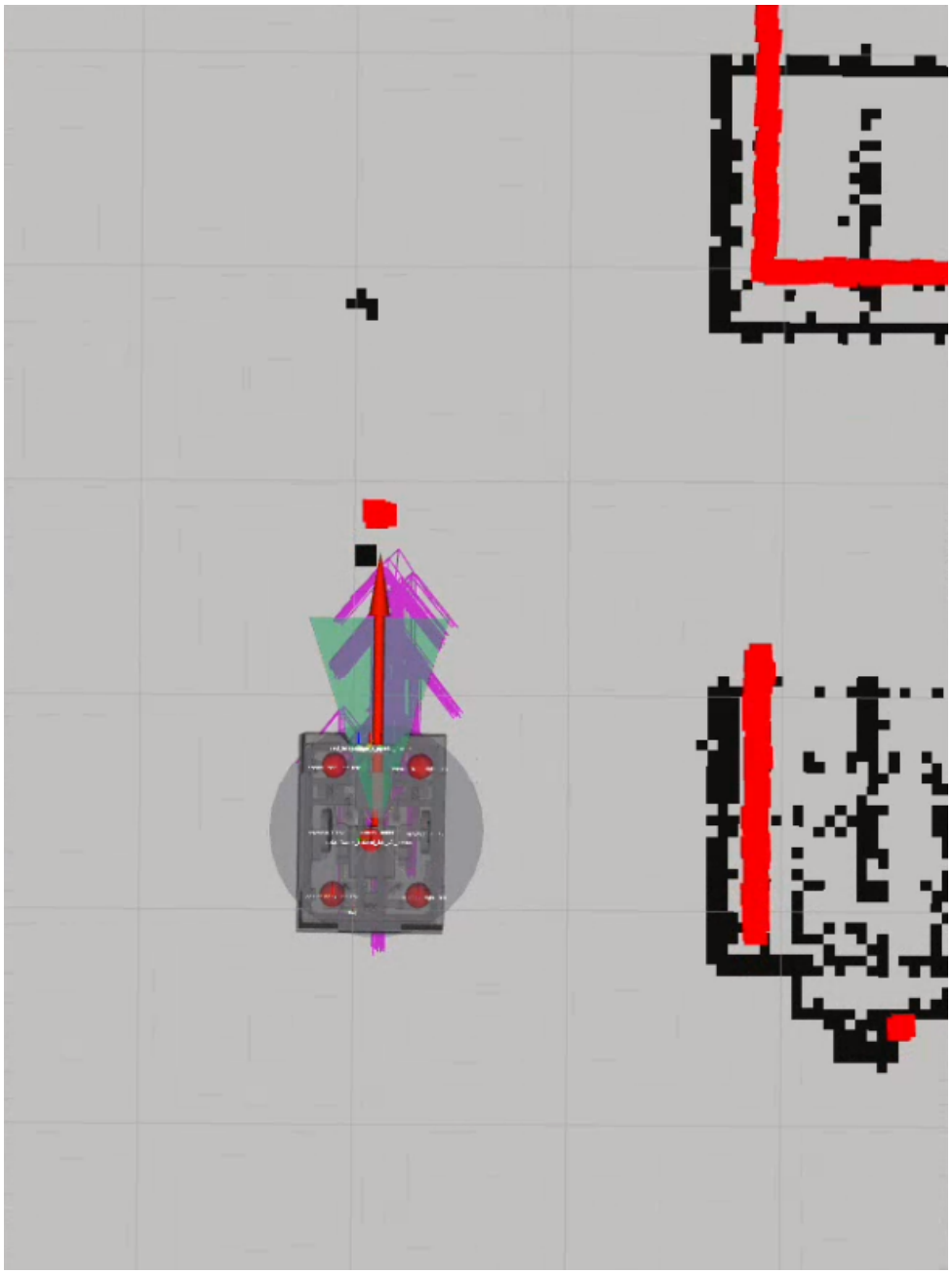

Figure 8: Final localization: accurate pose estimate with particle convergence

The purple arrows represent individual particles, with their distribution indicating localization uncertainty. The red arrow shows the estimated robot pose. As the robot moves and collects LiDAR measurements, particles converge to the correct location.

4 Question 3: A* Path Planning

4.1 Overview

The A* planner generates collision-free paths from the robot's current position to a goal location. The implementation includes obstacle inflation to account for robot size and uses an 8-connected grid for smooth paths.

4.2 Algorithm Theory

4.2.1 A* Search

A* is an informed search algorithm that finds the shortest path by expanding nodes with minimum cost:

$$f(n) = g(n) + h(n) \tag{14}$$

where:

- $g(n)$ is the cost from start to node n
- $h(n)$ is the heuristic estimate from n to goal
- $f(n)$ is the total estimated cost

4.2.2 Heuristic Function

For grid-based planning, Euclidean distance provides an admissible heuristic:

$$h(n) = \sqrt{(x_{goal} - x_n)^2 + (y_{goal} - y_n)^2} \tag{15}$$

This never overestimates the true cost, ensuring optimal paths.

4.2.3 Obstacle Inflation

To ensure collision-free paths, obstacles are inflated by the robot's radius:

$$r_{inflation} = \sqrt{(L/2)^2 + (W/2)^2} \times \text{safety\_margin} \tag{16}$$

where L and W are robot length and width. This diagonal distance ensures the robot can fit through passages.

4.3 Implementation

4.3.1 Map Inflation

```
def create_inflated_map(self):
    width = self.map_data.info.width
    height = self.map_data.info.height
    resolution = self.map_data.info.resolution

    original_map = np.array(self.map_data.data).reshape((height,
       width))
    inflation_cells = int(np.ceil(self.robot_radius / resolution)
       )

    self.inflated_map = original_map.copy()
    obstacle_coords = np.where(original_map > 50)

    for oy, ox in zip(obstacle_coords[0], obstacle_coords[1]):
        for dy in range(-inflation_cells, inflation_cells + 1):
```

```
            for dx in range(-inflation_cells, inflation_cells + 1):
                if dx*dx + dy*dy <= inflation_cells*inflation_cells:
                    ny, nx = oy + dy, ox + dx
                    if 0 <= ny < height and 0 <= nx < width:
                        if self.inflated_map[ny, nx] < 50:
                            self.inflated_map[ny, nx] = 99
```

Listing 9: Obstacle inflation for robot footprint

4.3.2 A* Search Implementation

```
def astar(self, start_x, start_y, goal_x, goal_y):
    counter = 0
    open_set = []
    heapq.heappush(open_set, (0, counter, start_x, start_y))
    counter += 1

    g_score = {(start_x, start_y): 0}
    came_from = {}
    closed_set = set()

    while open_set:
        _, _, current_x, current_y = heapq.heappop(open_set)

        if current_x == goal_x and current_y == goal_y:
            return self.reconstruct_path(came_from, current_x, current_y)

        if (current_x, current_y) in closed_set:
            continue

        closed_set.add((current_x, current_y))

        for nx, ny, cost in self.get_neighbors(current_x, current_y):
            if (nx, ny) in closed_set:
                continue

            tentative_g = g_score[(current_x, current_y)] + cost

            if (nx, ny) not in g_score or tentative_g < g_score[(nx, ny)]:
                came_from[(nx, ny)] = (current_x, current_y)
                g_score[(nx, ny)] = tentative_g
                f = tentative_g + self.heuristic(nx, ny, goal_x, goal_y)
                heapq.heappush(open_set, (f, counter, nx, ny))
                counter += 1

```

```
    return None
```

Listing 10: A* pathfinding algorithm

4.3.3 Neighbor Generation

```
def get_neighbors(self, x, y):
    neighbors = []
    directions = [
        (1, 0, 1.0), (-1, 0, 1.0), (0, 1, 1.0), (0, -1, 1.0),
        (1, 1, 1.414), (1, -1, 1.414), (-1, 1, 1.414), (-1, -1,
           1.414)
    ]

    for dx, dy, cost in directions:
        new_x, new_y = x + dx, y + dy
        if self.is_valid(new_x, new_y):
            if abs(dx) == 1 and abs(dy) == 1:
                if not self.is_valid(x + dx, y) or not self.
                   is_valid(x, y + dy):
                    continue
            neighbors.append((new_x, new_y, cost))

    return neighbors
```

Listing 11: 8-connected neighbors with corner checking

Diagonal moves use cost $\sqrt{2} \approx 1.414$ and check adjacent cells to prevent corner-cutting through obstacles.

4.4 Results

The A* planner successfully generates optimal collision-free paths. Figures 9 and 10 show planned paths.

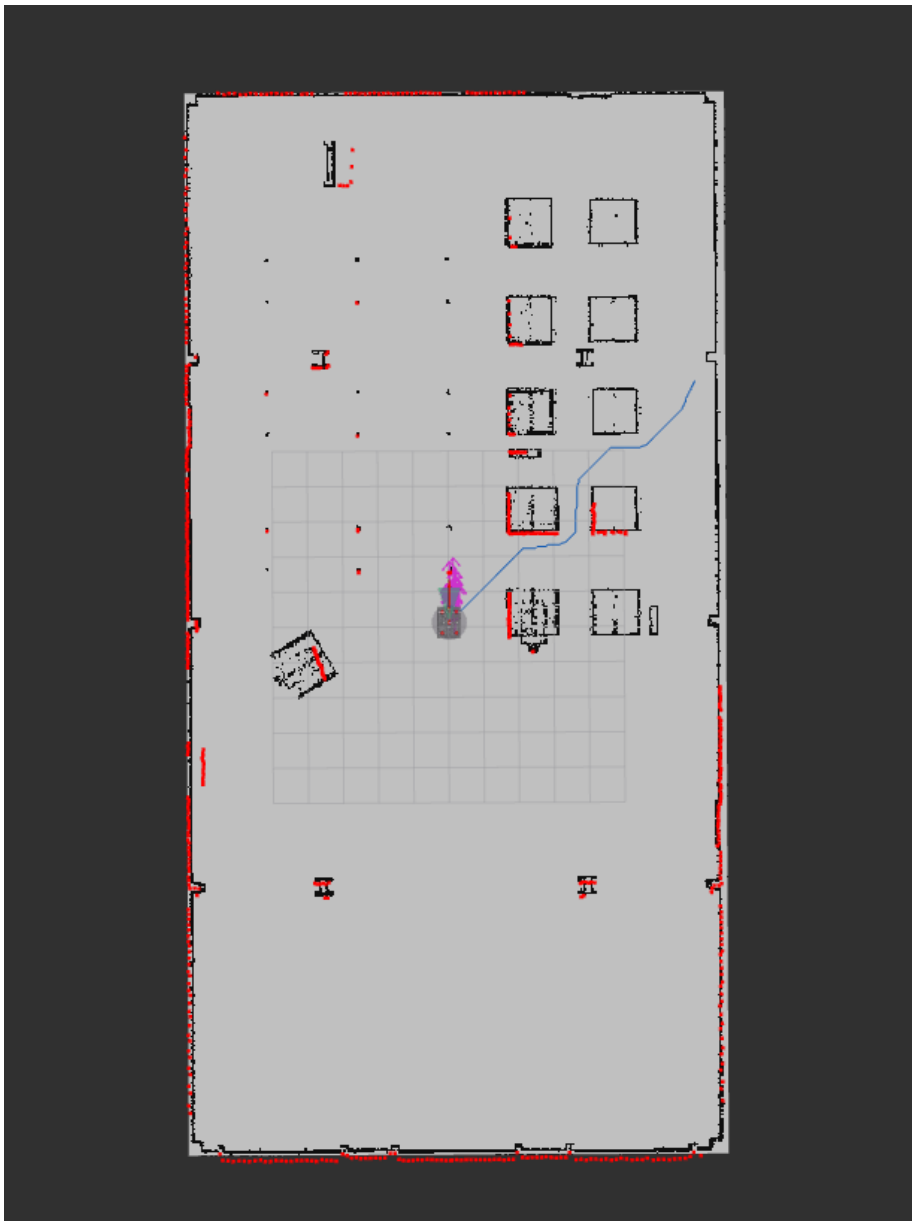

Figure 9: A* path planning result showing optimal path from start to goal

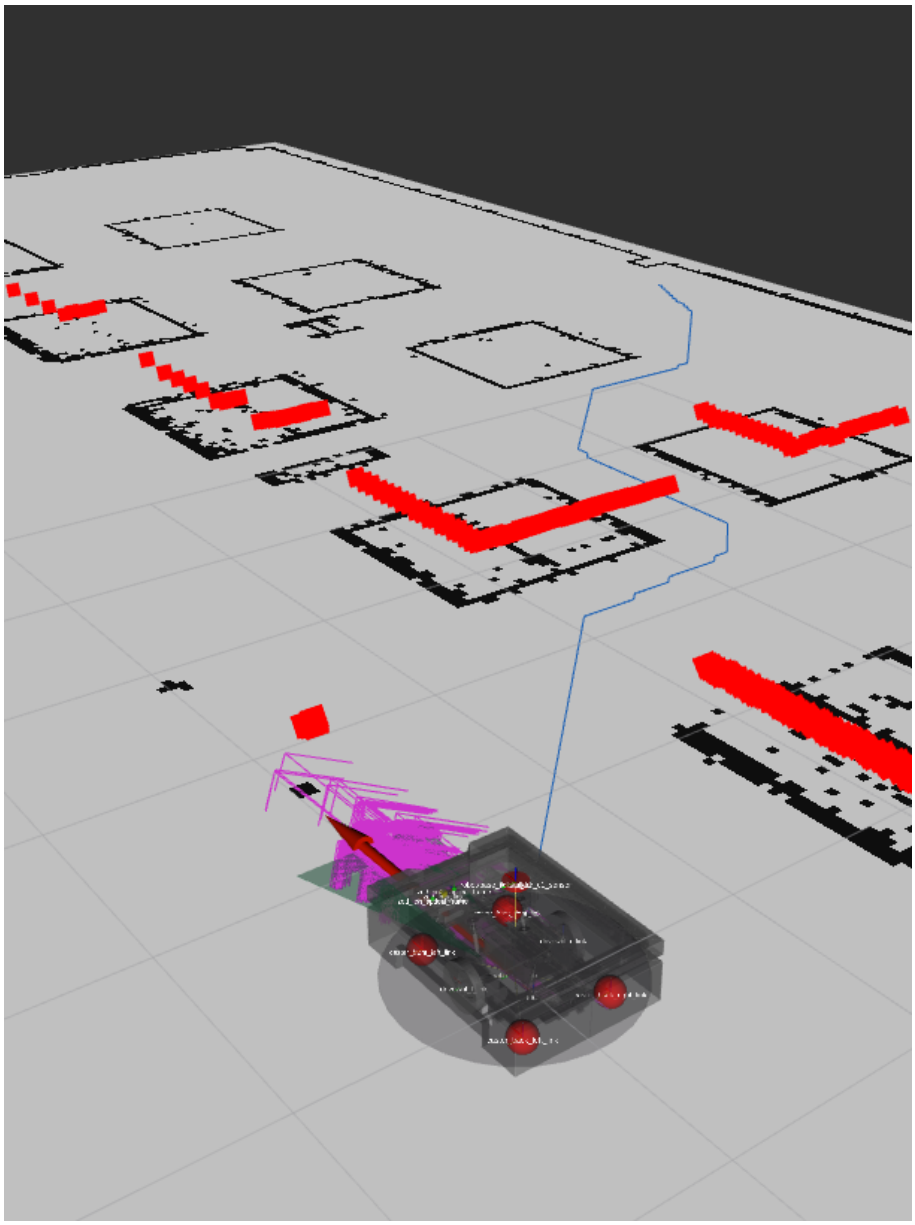

Figure 10: Path planning with obstacle avoidance around environment features

The green line represents the computed path. The planner successfully navigates around obstacles while minimizing path length.

5 Question 4: SLAM Implementation

5.1 Overview

The SLAM (Simultaneous Localization and Mapping) system builds a map of the environment while simultaneously determining the robot's position within it. This implementation uses scan matching and occupancy grid mapping.

5.2 System Design

The SLAM pipeline consists of:

- Motion model: predicts robot pose from odometry
- Scan matcher: aligns LiDAR scans to refine pose estimate
- Occupancy grid mapper: integrates scans into global map
- Loop closure: detects revisited locations for consistency

5.3 Results

Figures 11 through **??** show the progressive map building as the robot explores the environment.

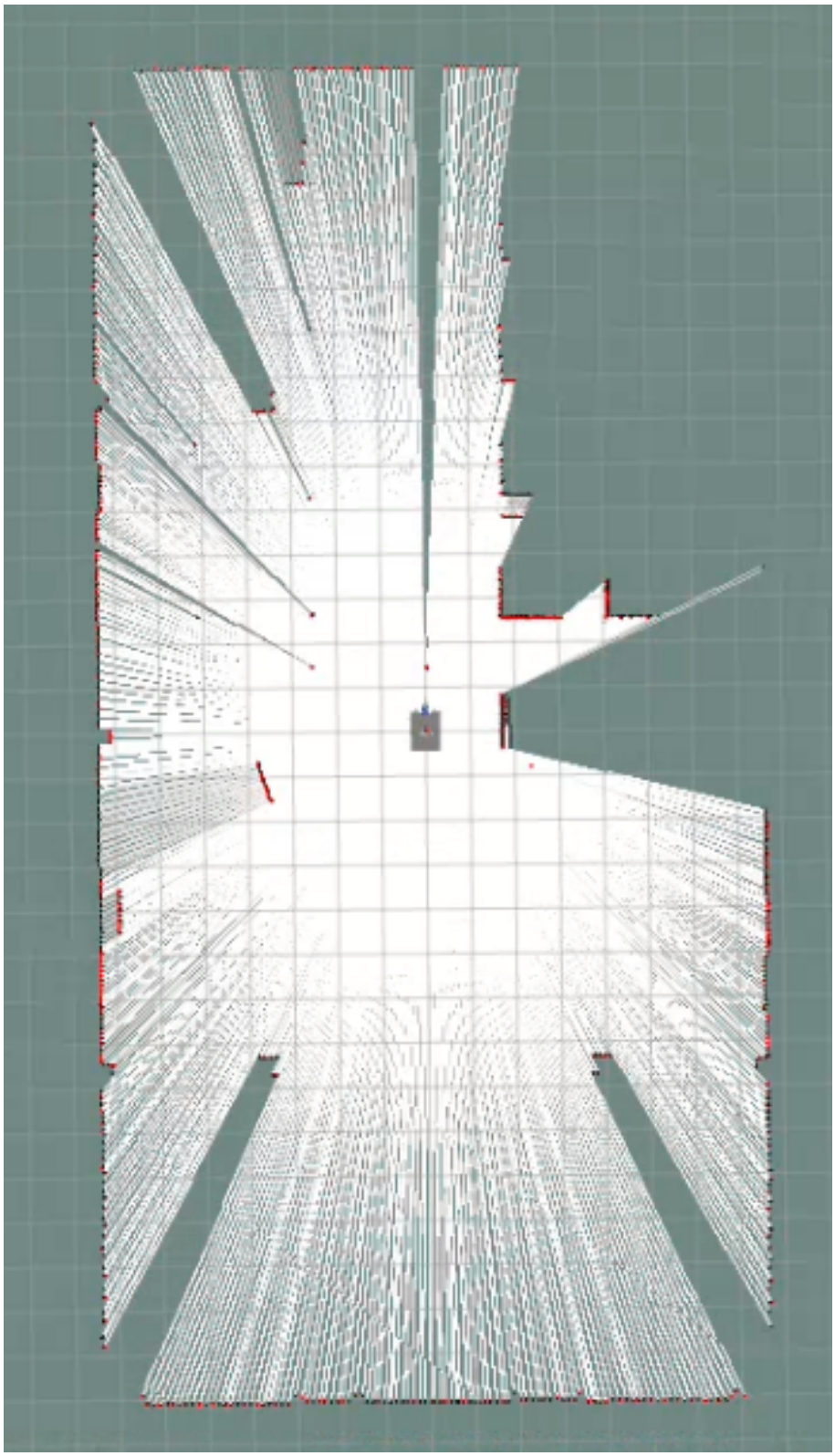

Figure 11: SLAM initialization: robot begins exploring and mapping the environment

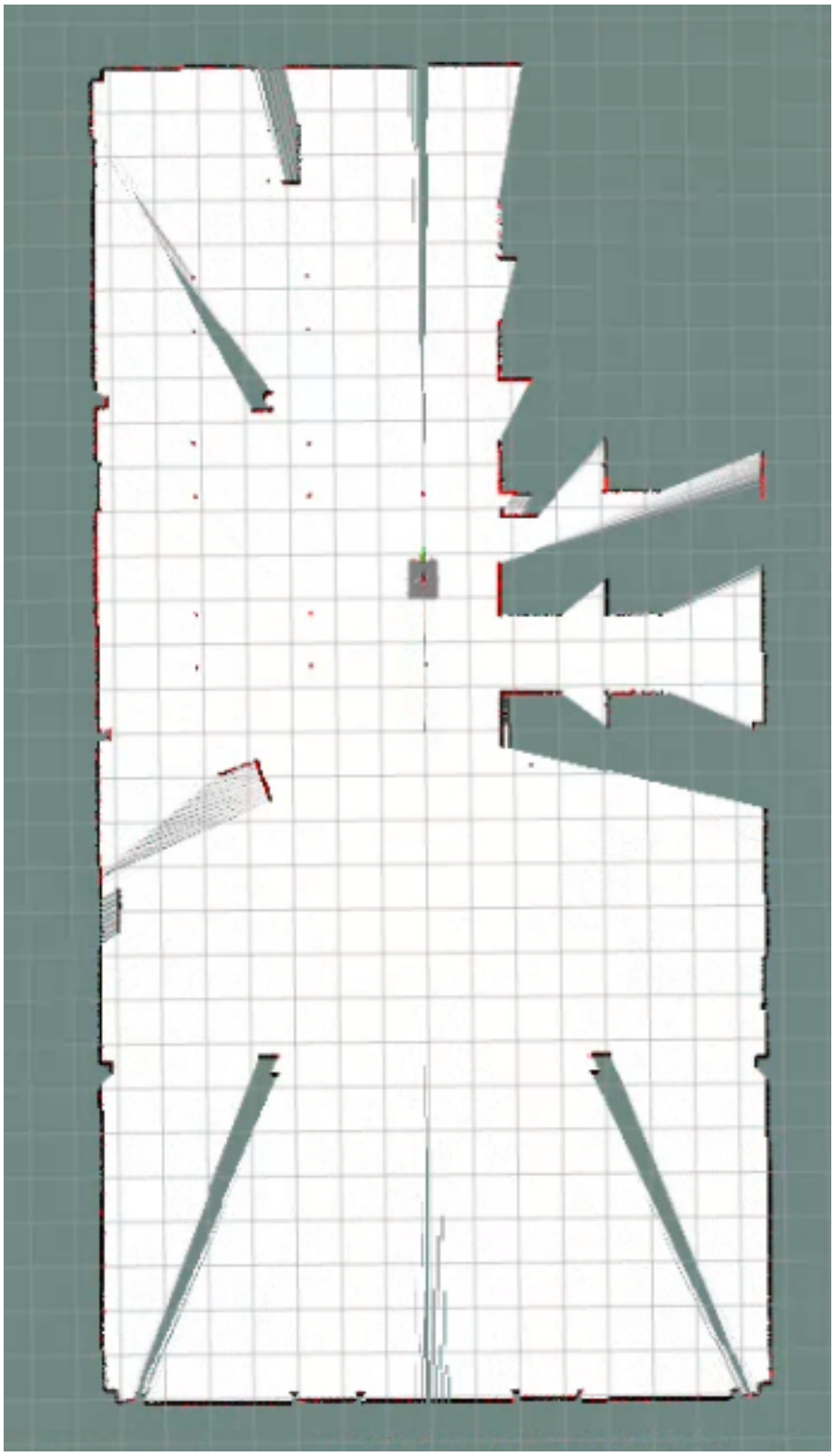

Figure 12: Map expansion: more of the environment becomes known as robot moves

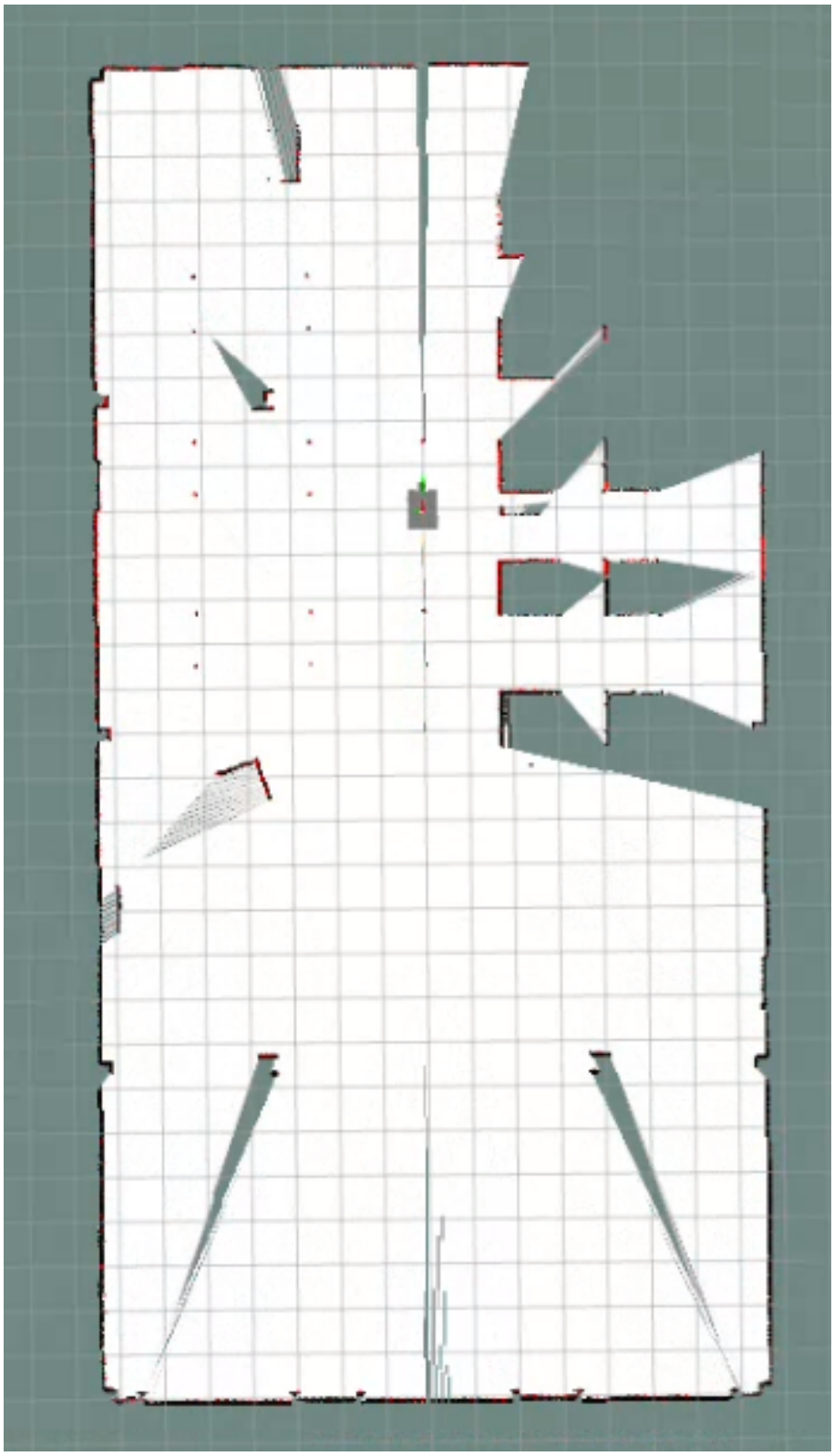

Figure 13: Continued exploration: map grows with additional sensor measurements

The grayscale visualization shows the occupancy grid, with darker regions representing obstacles and lighter regions indicating free space. As the robot moves, the map progressively becomes more complete and accurate.

6 System Integration and Performance

6.1 Complete Navigation Pipeline

The three main components work together in an integrated navigation system:

1. Map publisher provides environment representation
2. Particle filter determines robot pose within the map
3. A* planner generates paths to goal locations
4. Navigation controller follows planned paths

6.2 Key Parameters

Table 1 summarizes important system parameters.

Component	**Parameter**	**Value**
Map	Resolution	0.05 m/cell
Map	Origin	(-15, -7.9) m
Particle Filter	Particles	500
Particle Filter	Motion noise (α_1-α_4)	0.02-0.05
Particle Filter	Measurement sigma	0.15 m
Particle Filter	Resample threshold	0.2
A* Planner	Robot radius	0.33 m
A* Planner	Safety margin	1.05
A* Planner	Connectivity	8-connected

Table 1: Key system parameters

6.3 Performance Characteristics

6.3.1 Computational Complexity

The system's computational requirements:

- Map publishing: constant time after initial load
- Particle filter motion update: $O(N)$ where N is particle count
- Particle filter measurement: $O(N \times K)$ where K is beam count
- A* planning: $O(n \log n)$ where n is nodes explored

6.3.2 Real-time Performance

With 500 particles and LiDAR subsampling, the particle filter achieves real-time performance (10 Hz update rate). The A* planner typically finds paths in under 100ms for typical warehouse-scale environments.

6.4 Design Decisions

6.4.1 KD-Tree for Measurements

Using a KD-tree for obstacle distance queries provides $O(\log m)$ lookup time instead of $O(m)$ linear search, where m is the number of obstacles. This is critical for real-time performance with dense maps.

6.4.2 Map Inflation for Planning

Pre-inflating the map simplifies collision checking during path planning. Instead of checking robot geometry against each obstacle, we only need to verify that path cells are free in the inflated map.

6.4.3 Particle Filter vs Extended Kalman Filter

Particle filters handle multimodal distributions (global localization) better than EKFs, which assume unimodal Gaussian beliefs. This makes them suitable for the kidnapped robot problem and global localization scenarios.

6.5 Running Instructions

The following commands were used to run and validate the package:

```
ros2 service call /plan_path nav_msgs/srv/GetPlan "{goal: {header
   : {frame_id: 'map'}, pose: {position: {x: 7.0, y: -7.0, z:
   0.0}, orientation: {w: 1.0}}}}"
```

Listing 12: Calling the A* planner service with a desired goal (map frame)

This service call allows setting any desired goal coordinates by changing `x` and `y`.

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

Listing 13: Keyboard teleoperation for manual robot motion (run alongside the launch file)

This command is executed in parallel with the launch file to manually drive the robot while observing localization and planning behavior in RViz.

7 Conclusion

This report presented a complete navigation system for mobile robots implemented in ROS 2. The three main components work together seamlessly:

- The map publisher loads and distributes occupancy grid maps with proper coordinate transformations
- The particle filter achieves robust global localization through probabilistic state estimation
- The A* planner generates optimal collision-free paths accounting for robot geometry

The implementation demonstrates key concepts in mobile robotics including probabilistic localization, informed search algorithms, and spatial reasoning. The system successfully enables autonomous navigation in known environments.

Future improvements could include:

- Adaptive particle counts based on localization uncertainty
- Dynamic replanning for moving obstacles
- Path smoothing for improved trajectory quality
- Integration with velocity controllers for execution

The complete system provides a solid foundation for autonomous robot navigation and can be extended for more complex scenarios including dynamic environments and multi-robot systems.

8 GitHub Repository and Demonstration Videos

The complete implementation (including the ROS 2 package) is available in the following GitHub repository:

`https://github.com/Narjes-Gh2024/HW03_Robotic_Technology.git`

This repository also contains **two demonstration videos** that show the correct functionality of the package, including map publishing, particle filter localization convergence, and A* path planning.