Assignment: Planning and Control for Mobile Robot Navigation

Autonomous Systems

Overview

You will implement an integrated navigation system for a differential-drive mobile robot (TurtleBot3) that combines:

1. Global path planning using A* or Dijkstra
2. Kinematic waypoint tracking controller
3. Local obstacle avoidance using Potential Fields

All components should be implemented from scratch in Python, without using the ROS Navigation Stack (no `move_base`).

Learning bjectives

- Understand occupancy grid-based planning
- Apply kinematic control to track waypoints
- Implement local obstacle avoidance using sensor data
- Design an integrated reactive + deliberative system

Grading (100-point scale)

Global Path Planning (A*/Dijkstra)	20 pts
Kinematic Controller	20 pts
Potential Field Avoidance	20 pts
Simulation Demo	15 pts
Visualizations and Plots	10 pts
Code Quality and Structure	5 pts
Report and Analysis	10 pts
Total	**100 pts**

Conversion to 20-point grading scale

To convert your score from the 100-point scale to the 20-point grading system:

$$B_{20} = \frac{B_{100} \cdot 20}{100}$$

Where:

- B_{100}: Raw score out of 100
- B_{20}: Final grade on 20-point scale

This mapping is linear and preserves relative proportions.

Step-by-step implementation guide

Step 1: global planning

- Set up simulation (Gazebo + TurtleBot3 world)
- Load static map using `map_server`
- Implement A* or Dijkstra on `OccupancyGrid`
- Publish path as a list of (x, y) waypoints

Step 2: kinematic controller

- Subscribe to robot pose via `tf2`
- Implement differential-drive controller:

$$\begin{aligned}
\rho &= \sqrt{(x_g - x)^2 + (y_g - y)^2} \\
\alpha &= \arctan 2(y_g - y, x_g - x) - \theta \\
\beta &= -\theta - \alpha \\
v &= k_\rho \cdot \rho \\
\omega &= k_\alpha \cdot \alpha + k_\beta \cdot \beta
\end{aligned}$$

- Switch waypoints once $\rho < \epsilon$

Step 3: potential fields

Repulsive force:

$$\vec{F}_{\text{rep}} = \sum_{i=1}^{N} k_{\text{rep}} \cdot \left(\frac{1}{d_i^2} - \frac{1}{d_0^2} \right)_+ \cdot \frac{-\vec{r}_i}{d_i}$$

Attractive force:

$$\vec{F}_{\text{att}} = k_{\text{att}} \cdot (\vec{p}_g - \vec{p}_r)$$

Command:

$$\vec{F}_{\text{total}} = \vec{F}_{\text{att}} + \vec{F}_{\text{rep}}, \quad \theta = \arctan 2(F_y, F_x)$$

```
cmd.linear.x = min(norm(F), max_speed)
cmd.angular.z = gain * theta
```

Step 4: integration

- Combine controller and potential field planner
- Either:
  - Switch to PF planner if LIDAR detects obstacle ¡ threshold
  - Blend vectors:
    $$\vec{v}_{\text{cmd}} = w_1 \cdot \vec{v}_{\text{controller}} + w_2 \cdot \vec{v}_{\text{PF}}$$
- Tune all gains and test with various maps
- Collect plots, trajectories, and observations

Python-only architecture

- `global_planner.py`: implements A*/Dijkstra
- `navigator.py`: manages waypoint control and obstacle response
- `potential_fields.py`: computes local repulsion and attraction
- `kinematic_controller.py`: implements unicycle model control

Potential fields – practical Guide

This section describes how to compute repulsive forces from the LIDAR scan data published on the `/scan` topic, and convert them into a velocity command for obstacle avoidance.

1. LaserScan message overview

The topic /scan provides sensor_msgs/LaserScan messages. These include:

- ranges[] — array of distances [meters]
- angle_min, angle_max — angular limits of the scan
- angle_increment — angular resolution

Each reading corresponds to an angle:

$$\theta_i = \texttt{angle\_min} + i \cdot \texttt{angle\_increment}$$

2. Converting to Cartesian points

To convert scan data to 2D points in the robot frame:

```
import numpy as np

def laser_to_cartesian(scan):
    angles = np.linspace(scan.angle_min, scan.angle_max, len(scan.
       ranges))
    ranges = np.array(scan.ranges)
    mask = np.isfinite(ranges) & (ranges > 0.05)
    xs = ranges[mask] * np.cos(angles[mask])
    ys = ranges[mask] * np.sin(angles[mask])
    return np.stack((xs, ys), axis=-1)
```

3. Computing repulsive force

```
def compute_repulsive_force(points, d0=0.6, k_rep=0.8):
    force = np.zeros(2)
    for p in points:
        d = np.linalg.norm(p)
        if d == 0 or d > d0:
            continue
        direction = -p / d
        magnitude = k_rep * (1.0 / d**2 - 1.0 / d0**2)
        force += max(magnitude, 0) * direction
    return force
```

4. Converting to velocity commands

```
from geometry_msgs.msg import Twist

def force_to_cmd(force_vector, max_speed=0.3):
    angle = np.arctan2(force_vector[1], force_vector[0])
    speed = min(np.linalg.norm(force_vector), max_speed)
    cmd = Twist()
    cmd.linear.x = speed * np.cos(angle)
    cmd.angular.z = 2.0 * angle
    return cmd
```

5. Integration logic

```
if min(scan.ranges) < 0.4:
    cmd = force_to_cmd(compute_repulsive_force(laser_to_cartesian(
        scan)))
else:
    cmd = compute_tracking_cmd()
```

Integration of global and local planning

The navigation system must integrate two levels of planning:

- **Global planner** (A* or Dijkstra): plans a path through the known map from start to goal using a static occupancy grid.
- **Local planner** (Potential Fields): performs reactive obstacle avoidance based on LIDAR data in real time.

Control logic

Your `navigator.py` node should:

1. Compute the full global path using the planner once at the beginning (or when goal is updated).
2. Follow the path waypoint by waypoint using a kinematic controller.
3. At each control step:
   - Check LIDAR data for nearby obstacles.
   - If an obstacle is closer than a safety threshold (e.g., 0.4 m), activate potential fields to override the tracking command.
   - Otherwise, continue following the path.

Alternative: blended commands

Optionally, you may blend velocity vectors from both planners:

$$\vec{v}_{\text{cmd}} = \lambda \cdot \vec{v}_{\text{track}} + (1 - \lambda) \cdot \vec{v}_{\text{rep}}$$

where $\lambda \in [0, 1]$ is computed based on obstacle proximity. This results in smoother transitions but requires tuning.

Dijkstra's Algorithm for path planning

Dijkstra's algorithm computes the shortest path from a start cell to all other cells in a weighted grid where all edges have equal cost (typically 1 for free space).

Key steps

1. Initialize cost of all cells to ∞; set cost of start to 0.
2. Use a priority queue (min-heap) sorted by cost-to-come.
3. Repeatedly pop the lowest-cost node and expand its neighbors.
4. For each neighbor, update cost if a lower-cost path is found.
5. Keep a parent dictionary to reconstruct the final path.

Grid setup

- Convert OccupancyGrid to a 2D NumPy array. - Free space = 0, obstacle = 100 or greater. - Use 4- or 8-connected neighbor system.

Termination

- Stop when the goal is popped from the queue. - Backtrack from goal to start using the parent links.

Pseudocode

```
open_set = PriorityQueue()
open_set.put((0, start))
came_from = {}
cost = {start: 0}

while not open_set.empty():
_, current = open_set.get()
if current == goal:
break
```

```
for neighbor in get_neighbors(current):
new_cost = cost[current] + 1
if neighbor not in cost or new_cost < cost[neighbor]:
cost[neighbor] = new_cost
open_set.put((new_cost, neighbor))
came_from[neighbor] = current
```

A* Algorithm for path planning

A* extends Dijkstra by adding a heuristic that estimates the cost from each cell to the goal. This accelerates search by prioritizing nodes likely to lead to the goal.

Cost function

$$f(n) = g(n) + h(n)$$

- $g(n)$: cost-to-come from start to node n
- $h(n)$: heuristic (e.g., Euclidean or Manhattan distance to goal)

Key differences from Dijkstra

- A* uses both actual cost and estimated cost. - If $h(n) = 0$, A* reduces to Dijkstra. - A* is optimal if the heuristic is admissible (does not overestimate).

Typical heuristics

- Manhattan: $h = |x_1 - x_2| + |y_1 - y_2|$ (grid-like maps)
- Euclidean: $h = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Pseudocode

```
open_set = PriorityQueue()
open_set.put((0, start))
came_from = {}
g_cost = {start: 0}

while not open_set.empty():
_, current = open_set.get()
if current == goal:
break
for neighbor in get_neighbors(current):
tentative_g = g_cost[current] + 1
if neighbor not in g_cost or tentative_g < g_cost[neighbor]:
```

```
        g_cost[neighbor] = tentative_g
        h = heuristic(neighbor, goal)
        f = tentative_g + h
        open_set.put((f, neighbor))
        came_from[neighbor] = current
```

Implementation of Global and Local planners in ROS

This section outlines the precise steps to integrate a global planner (A*/Dijkstra) and a local reactive planner (potential fields) within a ROS-based system. All communication occurs through standard ROS topics using Python nodes (no move_base required).

Stage 1: Global map and path planning

1. **Subscribe to** `/map` (`nav_msgs/OccupancyGrid`) to obtain the static occupancy grid.
2. **Receive or define goal pose** in the `map` frame via:
   - RViz interactive marker,
   - or hardcoded coordinates.
3. **Run A* or Dijkstra** on the occupancy grid to generate a path as a list of 2D waypoints.
4. **Optionally publish the path** to `/planned_path` (`nav_msgs/Path`) for visualization in RViz.

Stage 2: Local control loop (Real-Time)

1. **Subscribe to robot pose**:
   - Either use `/odom` (`nav_msgs/Odometry`),
   - Or use TF transform from `map` to `base_footprint`.
2. **Subscribe to** `/scan` (`sensor_msgs/LaserScan`) for live obstacle data.
3. **Track the path:**
   - Use a queue of waypoints from the global planner.
   - Switch to the next waypoint when the robot is within a threshold distance.

Stage 3: Command generation and arbitration

1. **Check obstacle distance** using `/scan`:
   - If an obstacle is closer than a defined safety threshold (e.g., 0.4 m), activate the local planner (potential fields).
   - Else, follow the waypoint using the kinematic controller.
2. **Publish velocity commands** to `/cmd_vel` (`geometry_msgs/Twist`).

Optional: command blending

Instead of switching between planners, use blended control:

$$\vec{v}_{\text{cmd}} = \lambda \cdot \vec{v}_{\text{track}} + (1 - \lambda) \cdot \vec{v}_{\text{rep}}$$

where λ is based on obstacle proximity. This approach yields smoother behavior but requires careful tuning.

Topic summary

- `/map` — `nav_msgs/OccupancyGrid`
  Global map input for A*/Dijkstra.
- `/scan` — `sensor_msgs/LaserScan`
  Obstacle distances for reactive planning.
- `/odom` or `/tf` — `nav_msgs/Odometry` or TF
  Robot's current position in the map.
- `/cmd_vel` — `geometry_msgs/Twist`
  Final velocity command to control the robot.
- `/planned_path` (optional) — `nav_msgs/Path`
  Visualization of the planned global path.

Submission guidelines and deadlines

Deadline: The complete assignment must be submitted no later than [**July, 7th, 23.59**] via GitHub and email/upload to iLearn.

What to submit

- A public or private **GitHub repository** containing:
  - Full ROS package with source code (`scripts/`, `launch/`, `rviz/`)
  - Launch files and README with instructions to run the planner and controller

- A written **project report (minimum 3 pages)** in IEEE conference format (`ieeeconf.cls`). The report must include:
  - Clear description of your architecture and planning/control strategy
  - Plots of resulting robot trajectories (with and without obstacles)
  - Screenshots of RViz/Gazebo simulations
  - Quantitative performance metrics (e.g., time to goal, tracking error)
  - Summary of experimental observations and tuning
- A short **demo video** (1–3 minutes) demonstrating:
  - Path planning result
  - Obstacle avoidance behavior
  - Robot navigation in simulation or real-world setup (optional)

Minimum deliverables: working simulation, valid IEEE-style report, GitHub repo, and video demonstration.

Review of the reports: July 9, 2025