Terrain-Navigate: 3D Pathfinding Library

Flexible Navigation for Robots in Complex Environments

Tavish Mankash

Plaksha University

December 6, 2025

Outline

Idea & Motivation

- **Problem:** To make a simple, few-line Python library for 3D path-finding, particularly focussed on robots.
- **Goal:** Build a lightweight, modular Python library for 3D A* pathfinding.
- **Key Features:**
  - Custom cost functions (slope, energy, roughness) useful for robots.
  - Easy integration with point cloud data (LAS/LAZ) and 3D meshes.
  - "Plug-and-play" architecture for environments and algorithms.
- **Why?** Enable rapid prototyping for robots in unstructured environments (like MDRS).

A* Search Fundamentals

- Path search for rovers moving in an open field is computationally difficult.
- **A* Algorithm:** An informed search algorithm that efficiently finds the shortest path between nodes in a graph.
- **Key Idea:** Combines the cost to reach a node with an estimate of the cost to reach the goal.
- **Formula:** $f(n) = g(n) + h(n)$
    - $f(n)$: Total estimated cost from start to goal through node n.
    - $g(n)$: Actual cost from the start node to node n.
    - $h(n)$: Heuristic estimate of the cost from node n to the goal node.
- **Intuition:** $g(n)$ tells us how far we've come, $h(n)$ tells us how far we *think* we still need to go. A* tries to minimize their sum.

Cost Functions

- **Approach 1**:

$$h(n) = \sum_{i=start}^{n} \left(\frac{\Delta z_i}{\Delta r_i}\right)^n$$

- **Approach 2**:

$$h(n) = \sum_{i=start}^{n} \left(\frac{\Delta z_i}{\Delta r_i}\right)^{n_1} + \left(\frac{\Delta z_i}{\Delta r_{pi}}\right)^{n_2}$$

- **Approach 3**:

$$h(n) = \sum_{i=start}^{n} (\alpha_1 \cdot \frac{\Delta z_i}{\Delta r_i} + \beta_1)^{n_1} + (\alpha_2 \cdot \frac{\Delta z_i}{\Delta r_{pi}} + \beta_2)^{n_2}$$

- **Approach 4**:

$$h(n) = \sum_{i=start}^{n} \Delta r \cdot (\Delta z)^{n_1} + \alpha \cdot \Delta r_p \cdot (\Delta z)^{n_2}$$

Cost Functions

- **Implemented Costs:**
  - `EuclideanCost`: Standard shortest path.
  - `PowerCost`: Penalizes vertical movement ($Cost = Dist \times (1 + \alpha \cdot slope^n)$).
  - `EnergyBasedCost`: Physics-based (Rowe & Ross, 2000) considering mass, gravity, friction.
  - `TerrainTraversabilityCost`: Composite metric (slope, elevation).
- **Flexibility:** Users can define custom cost classes by inheriting from `CostFunction`, which is an ABC abstract class. (Show example)

From Point Clouds to Grids

- **Challenge:** Raw point clouds (LAS/LAZ) are very difficult to search in.
- **Solution:** Voxelization.
- **Process:**
  1. Read data using `laspy`.
  2. Determine bounding box and resolution (e.g., 0.02m grid).
  3. Search for points, map points to grid cells, average out points in case of low resolution.
  4. Fill holes to ensure continuity.
- **Libraries Explored:** `laspy` (chosen for speed/simplicity), `open3d` (explored but heavier).

Visualizations & Demo

Integration: The MDRS Dataset Effort

- **Context:** Mars Desert Research Station (MDRS) in Utah. Analog environment for Mars missions.
- **Data Source:** USGS Lidar Point Cloud (UT). High-fidelity terrain data.
- **Challenge:** Finding the *exact* patch of land corresponding to MDRS from terabytes of US data.
- **Solution - Data Pipeline:**
  1. **Metadata Parsing:** Parsed thousands of XML metadata files from USGS repository.
  2. **Geospatial Filtering:** Filtered files based on MDRS latitude/longitude bounding box.
  3. **Candidate Selection:** Identified specific LAS files covering the station area.

Integration: Processing Workflow

- **From Raw Data to Pathfinding:**
  - **Input:** Raw .las/.laz point cloud files (millions of points).
  - **Processing (`preprocess_data.py`):**
    - Voxelization: Mapping 3D points to a 2D grid.
    - Grouped Point Clouds: Initial approach to group point clouds.
    - Hole Filling: Handling missing lidar returns.
  - **Output:** `processed_map.npy` - A clean, navigable heightmap.
- **Result:** We can now run A* with physics-based costs on *real* Martian analog terrain, not just synthetic noise.

Quick Tutorial: How to Use

```python
from terrain_navigate import AStar, GridEnvironment,
    PowerCost
import numpy as np

# 1. Load your map (numpy array)
terrain_map = np.load("map.npy")

# 2. Setup Environment
env = GridEnvironment(Z=terrain_map, resolution=1.0)
# Supports Point Clouds and Grouped Point Clouds (
    Explain)

# 3. Define Cost Function
cost_fn = PowerCost(n=2.0, alpha=10.0)

# 4. Find Path
astar = AStar()
path, cost = astar.find_path(start_node, goal_node,
    env, cost_fn)

```

Future Work

- **Optimization:** Implement Jump Point Search (JPS), Theta* or D*.
- **Dynamic Costs:** Time-varying costs
- **Environment:** Support higher dimensional information about environment such as soil conditions, etc to improve cost functions used.

Thank You

Questions?