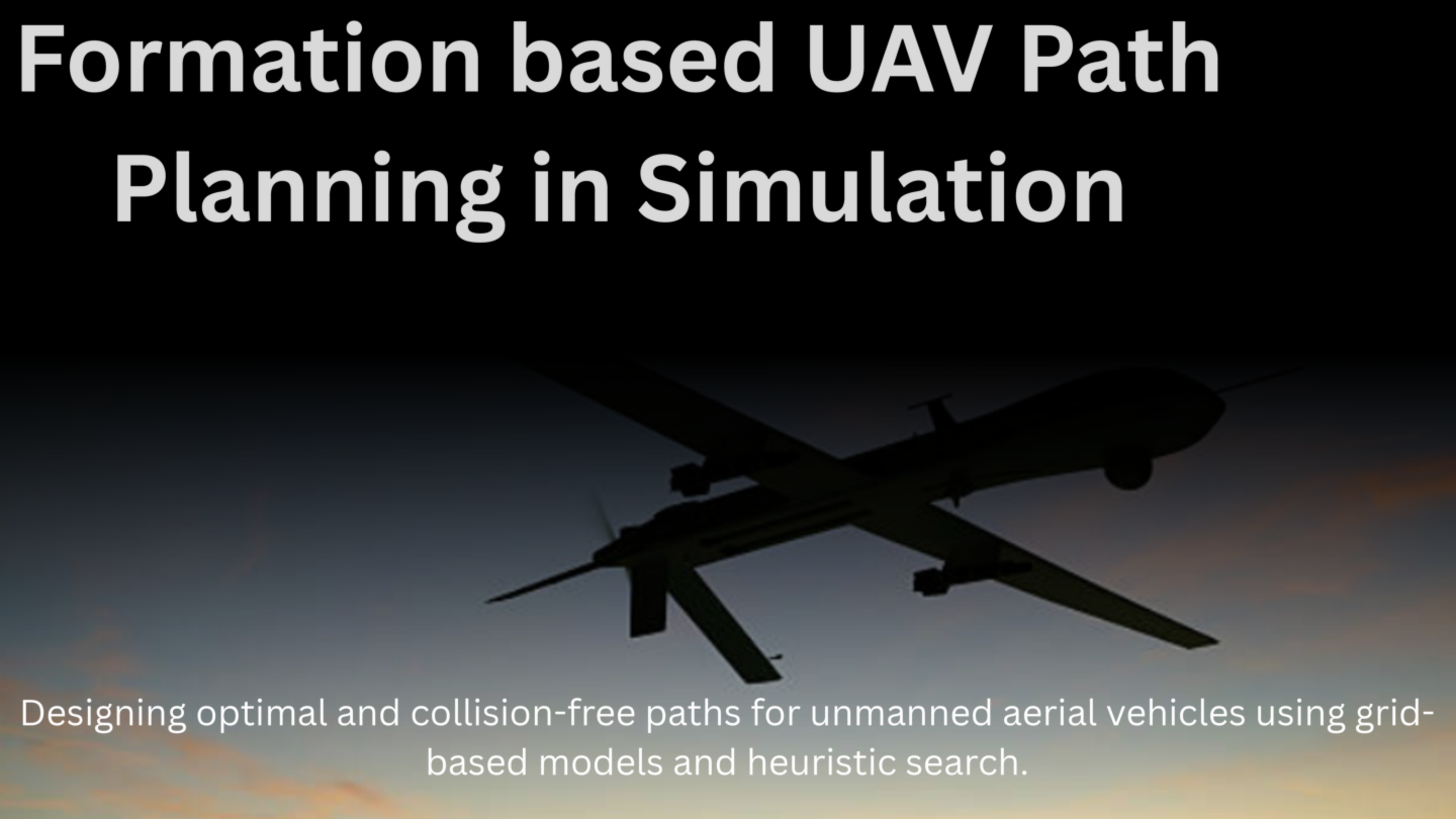
Formation based UAV Path Planning in Simulation



Designing optimal and collision-free paths for unmanned aerial vehicles using grid-based models and heuristic search.

The Development Environment: Google Colab

What is it?

A cloud-based platform for writing and executing Python in interactive "Jupyter Notebooks"

- Key Features:
  - Zero Setup: Runs in the browser; no local installation required.
  - Hardware Acceleration: Free access to GPUs and TPUs for intensive tasks like machine learning.
  - Cloud Storage: Direct integration with Google Drive and GitHub for saving and sharing work.
  - Interactive Documentation: Uses Markdown cells to combine code with formatted text and images.

Core Foundations: Python & NumPy

Python Essentials: Utilizing Corey Schafer’s "Python Tutorials" for in-depth language mastery, covering development tips and tricks.

NumPy Fundamentals:

- Essential for handling large, multi-dimensional arrays and matrices.
- Foundation for almost all scientific computing in Python.

Interactive Learning: Using provided Colab notebooks to practice basic syntax and array manipulations in real-time.

Introduction to Matplotlib

The Industry Standard: Matplotlib is the primary library for creating static, animated, and interactive visualizations in Python.

Basic Plotting Workflow:

- Defining data (X and Y arrays).
- Using plt.plot() for quick visualizations.
- Enhancing plots with labels (xlabel, ylabel) and legends for clarity.

Professional Aesthetics: Introduction to styles like scienceplots to transform "crayon-like" default plots into journal-ready figures.

Advanced Statistical Visualization

Histograms & Density Plots:

- Moving beyond simple counts to Density Plots where the area under the curve equals one.
- Using histtype='step' for overlapping histograms to compare multiple datasets without clutter.

Professional Formatting:

- Customizing figure sizes (e.g., 8x3 ratio for wide data).
- Adding mathematical notation (LaTeX) and standard deviation text boxes to plots.

Multi-Dimensional Data Visualization

- Visualizing Surfaces (2D to 3D):
  - Contour Plots: Using contourf for filled color maps to represent 3D data on a 2D plane.
  - Color Bars: Adding scales to relate color to specific values (e.g., voltage or temperature).
- Vector Fields:
  - Stream Plots: Visualizing flow and magnitude in fluid dynamics or magnetic fields.
  - Using line width to represent speed/intensity for a more "artistic" and informative look.

Dynamic Visuals & Animations

Why Animate? Better for showing temporal changes (e.g., moving waves) or exploring 3D structures.

Key Techniques:

- Function Animation: Dynamically updating a plot frame-by-frame (e.g., a shifting sine wave).
- 3D Rotation: Creating rotating GIFs of surface plots to provide a full 360-degree perspective.

Exporting: Saving high-resolution results as GIFs or MP4s for presentations.

Practical Application: Simulations

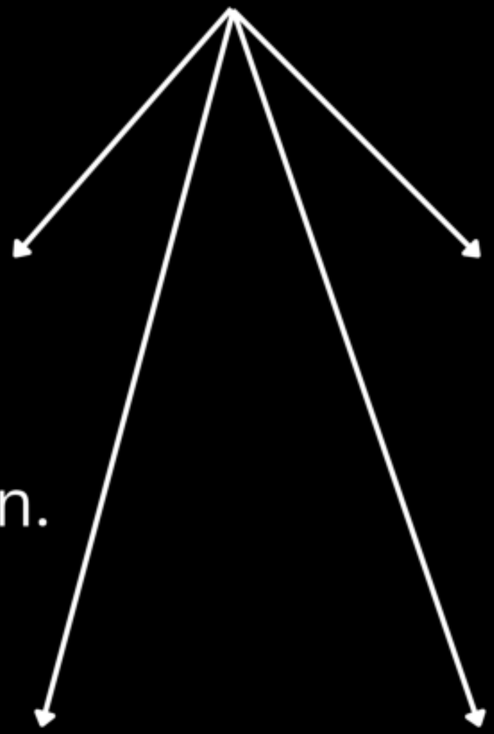
Modeling with Matplotlib: Using the Matplotlib_for_simulation.ipynb notebook to apply these tools to real-world scenarios.

Integrated Workflow:

1. Simulate: Use NumPy to generate mathematical models.
2. Visualize: Use Matplotlib to plot results.
3. Analyze: Use subplots to compare different simulation trials side-by-side.

Graphs

Set of nodes and edges connecting adjacent pairs of nodes



Undirected Graph

- Edges have no direction.

Directed Graph

- Edges have direction

Unweighted

- All edges treated as cost = 1

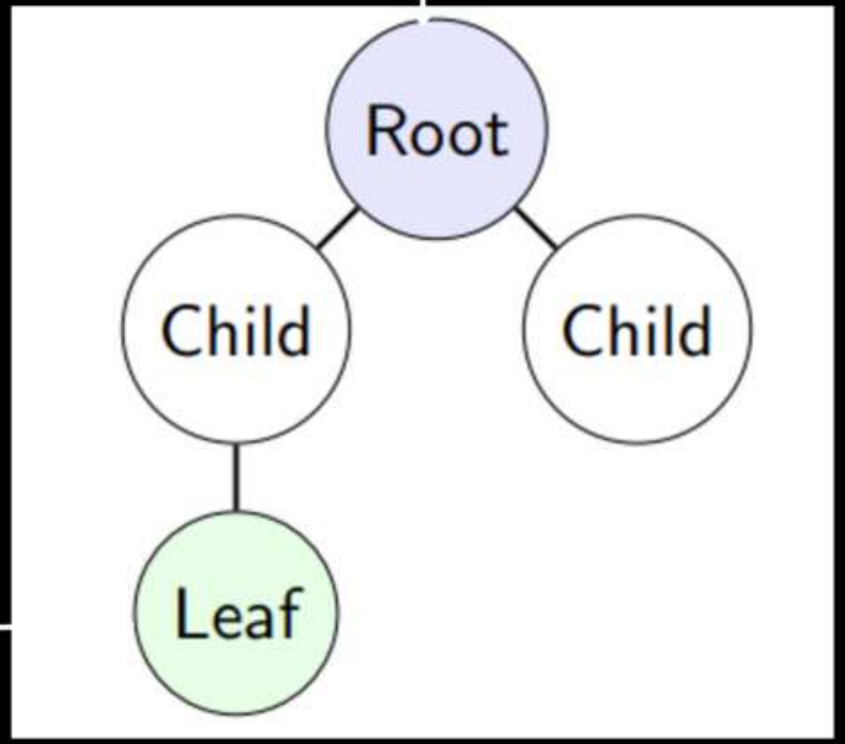
Weighted

- Each edge has a cost

Tree

A tree is a special graph that has no cycles , has exactly one simple path between any two nodes and if it has N nodes, it has N − 1 edges.

The chosen starting node



Depth

Distance (in edges) from the root

Nodes with no children

Traversal methods

- Two fundamental graph traversal methods

Depth First Search (DFS)

- Goes deep before backtracking
- Implemented using recursion or stack
- Used for exploring all possible paths

A → B → D → C → E

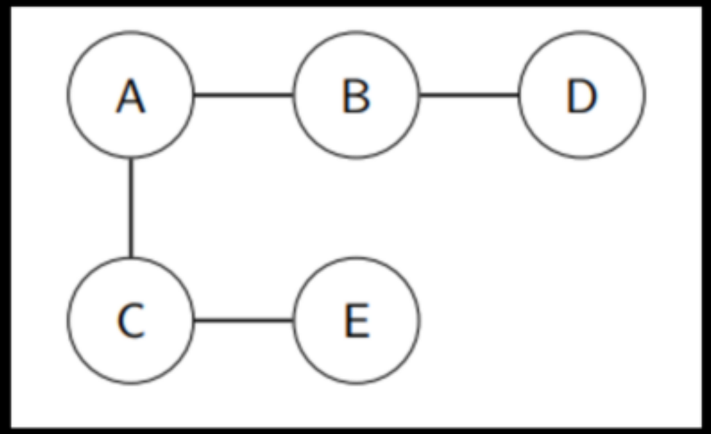
(Order can vary slightly depending on neighbor order.)

Breadth First Search (BFS)

- Traverses level by level
- Uses a queue
- Finds shortest path in unweighted graphs

A → B → C → D → E

(First visit all neighbors of A, then neighbors of those, and so on.)



BFS vs DFS

BFS uses Queue – Level order
BFS guarantees shortest path

↓

It explores in layers:

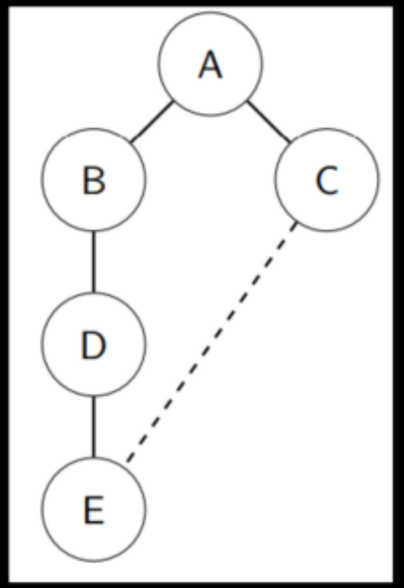
- Distance 0: source node.
- Distance 1: neighbors.
- Distance 2: neighbors of neighbors.

The first time it reaches the target, it has used the minimum number of edges

DFS uses Stack/Recursion – Depth first
DFS does not guarantee shortest path

↓

- It explores deeply first
- May miss shorter paths



BFS : A → C → E (2 edges) (shortest path)
DFS might go: A → B → D → E (3 edges).

Dijkstra Algorithm

Core Idea —

- Dijkstra is a shortest-path algorithm for weighted graphs
- Finds the minimum-cost path from a source to all nodes
- Uses a greedy strategy, Always expands the node with smallest known distance
- Maintains distance array dist[]
- Uses a priority queue for best-first expansion
- Once a node is finalized, its shortest distance is guaranteed

Distance Concept

- dist(u) = shortest known cost from start → u
- Initialization: dist(start) = 0 and all others = ∞
- Edge relaxation:
  - If dist(u) + weight(u,v) < dist(v)
  - then update dist(v)

This continuous relaxation slowly reveals the optimal structure of the graph.

Algorithm Flow —

1. Insert start node into priority queue
2. While queue not empty:
    - Extract node u with minimum dist
    - For each neighbor v of u:
        - Compute new distance
        - If better → update dist(v) and push into queue
    - Mark u as visited (finalized)
3. Stop when destination is finalized

Why Dijkstra is Optimal

- Greedy choice is safe because all edge weights ≥ 0
- Once node u has minimum distance, no shorter path to u can exist
- Guarantees:
    a. Correctness
    b. Optimal shortest path
    c. Monotonic improvement of solution

GRID PATH PLANNING MODEL

Grids as Graphs

- Each grid cell (i, j) is a node
- Edges connect neighboring cells
- Obstacles remove nodes
- Pathfinding = shortest path problem in graph

4-Direction Grid Movement:

- Allowed moves: up, down, left, right
- Movement only along grid axes
- Each move cost = 1
- BFS guarantees shortest path

8-Direction Grid Movement:

- Includes diagonal moves
- Straight cost = 1, diagonal cost = √2
- Matches real geometric distance
- Requires weighted path planning

BFS on Grids:

- Explores level by level
- First visit gives minimum distance
- Works only for equal-cost moves
- Explores many unnecessary cells

Why BFS is Not Enough :

- No direction toward goal
- Blind exploration
- Inefficient for large grids
- Need goal-directed search

A* Algorithm

Core Idea-

- A* is a best-first search algorithm
- Uses evaluation function:
- f(n) = g(n) + h(n)
- g(n): cost from start to current node
- h(n): estimated cost from current node to goal
- Node with lowest f(n) is expanded first

Heuristic Function h(n)

Heuristic estimates remaining distance to goal.
It must be fast and easy to compute.

It should not overestimate actual cost (admissible)
and it must guide search toward target efficiently

A* Algorithm — Steps

1. **Insert start node into OPEN**
2. **While OPEN not empty:**
   - Select node with minimum f(n)
   - If goal reached → stop
   - Generate valid neighbors
   - Update g, h, f values
   - Add/update neighbors in OPEN
   - Move current node to CLOSED

Why A* is Optimal

- g(n) ensures correct path cost
- h(n) guides search direction
- OPEN ensures best-first expansion
- CLOSED avoids repeated work
- With admissible heuristic, A* always finds shortest path

GROUP MEMBERS

LOVERAJ BIRDA
240599

Shivanshu Jaiswara
240985

D.sahasra
240313

Smit
241021

THANK YOU ...