ROW NAVIGATION

import matplotlib.pyplot as plt

import numpy as np

from scipy.interpolate import CubicSpline

import math

import time

from matplotlib.patches import Polygon

import matplotlib.transforms as transforms

from farm\_safety import SafetyModule

safety = SafetyModule()

STEP = 0.4 # Reduced back to original value for smoother movement (was 0.8)

TOLERANCE = 0.5 # Slightly reduced for better precision

MAX\_ATTEMPTS = 200 # Kept the same

DEBUG = False

ANIMATION\_SPEED = 0.05 # Increased for slowerment

# Moved from farm\_entry.py

class Rover:

def \_\_init\_\_(self):

self.x = 0.0

self.y = 0.0

self.heading = 0.0

self.last\_heading = 0.0 # Added memory for last heading

self.history = []

self.geofence = None

self.entry\_point = None

self.inside\_fence = False

self.command\_count = 0 # Add command count tracking

self.blocked\_directions = set() # Set of blocked directions

# Add to Rover class

def log\_movement(self, movement\_type, distance=None, angle=None):

"""Log movement commands to the terminal"""

if movement\_type == "forward":

print(f"⬆️ COMMAND: Move forward {distance:.2f}m")

elif movement\_type == "backward":

print(f"⬇️ COMMAND: Move backward {distance:.2f}m")

elif movement\_type == "turn\_left":

print(f"↩️ COMMAND: Turn left {angle:.1f}°")

elif movement\_type == "turn\_right":

print(f"↪️ COMMAND: Turn right {angle:.1f}°")

elif movement\_type == "stop":

print("🛑 COMMAND: Stop")

self.command\_count += 1

def calculate\_heading\_to(self, tx, ty):

dx, dy = tx - self.x, ty - self.y

if abs(dx)<1e-6 and abs(dy)<1e-6:

return self.heading

ang = math.degrees(math.atan2(dy, dx))

return ang if ang>=0 else ang+360

def set\_position(self, x, y, heading=None, force=False, add\_to\_history=True):

if self.geofence and not force:

in\_fence = self.is\_inside\_farm(x, y)

if not in\_fence:

return False

# Proceed with setting position

if self.geofence and not force:

if self.entry\_point and self.distance\_to(\*self.entry\_point) <= 0.8:

self.inside\_fence = True

in\_fence = self.is\_inside\_farm(x, y)

if (in\_fence and not self.inside\_fence) or (not in\_fence and self.inside\_fence):

print("⚠️ Movement blocked: would cross fence boundary")

return False

# Store previous position and heading before updating

prev\_x, prev\_y = self.x, self.y

prev\_heading = self.heading

# Update position with limited decimal places (point 2)

self.x = round(x, 2)

self.y = round(y, 2)

if heading is not None:

self.heading = round(heading % 360, 1)

# Update last\_heading when heading changes

self.last\_heading = self.heading

if add\_to\_history:

self.history.append((self.x, self.y))

self.command\_count += 1 # Increment command count

# Add GPS-like position reporting (point 5)

print(f"📍 GPS: Position [{self.x:.2f}, {self.y:.2f}], Heading: {self.heading:.1f}°")

return True

def set\_geofence(self, vertices, entry\_point):

self.geofence = vertices

self.entry\_point = entry\_point

self.inside\_fence = self.is\_inside\_farm(self.x, self.y)

def distance\_to(self, tx, ty):

return math.hypot(tx - self.x, ty - self.y)

def is\_inside\_farm(self, x, y):

if not self.geofence:

return True

# Simple boundary check

min\_x = min(v[0] for v in self.geofence)

max\_x = max(v[0] for v in self.geofence)

min\_y = min(v[1] for v in self.geofence)

max\_y = max(v[1] for v in self.geofence)

return min\_x <= x <= max\_x and min\_y <= y <= max\_y

def move\_forward(self, distance, ax=None, fig=None, rover\_patch=None):

# Log the movement command to terminal

self.log\_movement("forward", distance=distance)

rad = math.radians(self.heading)

target\_x = self.x + distance \* math.cos(rad)

target\_y = self.y + distance \* math.sin(rad)

success = self.set\_position(target\_x, target\_y)

if success and ax and fig and rover\_patch:

update\_rover\_visualization(self, ax, fig, rover\_patch)

plt.pause(0.05) # Add pause for smoother animation

return success

# Added method to detect if rover is stuck

def detect\_and\_resolve\_stuck(self):

# If this method is called, we assume potential stuck situation

# Return recommendation for new heading if needed

if len(self.blocked\_directions) > 3:

# Clear blocked directions and suggest a completely new heading

self.blocked\_directions.clear()

# Try perpendicular to current heading

new\_heading = (self.heading + 90) % 360

return new\_heading

return None

# Moved from farm\_entry.py

def update\_rover\_visualization(rover, ax, fig, rover\_patch=None):

if rover\_patch is None:

verts = np.array([[0.7, 0], [-0.3, 0.4], [-0.3, -0.4]])

rover\_patch = Polygon(verts, closed=True, fc='blue', ec='black')

ax.add\_patch(rover\_patch)

tr = transforms.Affine2D().rotate\_deg(rover.heading).translate(rover.x, rover.y)

rover\_patch.set\_transform(tr + ax.transData)

if hasattr(ax, 'path\_line') and len(rover.history) > 1:

ax.path\_line.set\_data(\*zip(\*rover.history))

fig.canvas.draw\_idle()

plt.pause(0.01)

return rover\_patch

# Moved from farm\_entry.py

import matplotlib.pyplot as plt

def visualize\_turn(rover, target\_heading, ax, fig, rover\_patch=None, rotation\_speed\_factor=1):

"""

Turn the rover to face a new heading using the shortest possible rotation.

"""

# Calculate turn direction and angle

delta\_angle = ((target\_heading - rover.heading) + 180) % 360 - 180

direction = "turn\_right" if delta\_angle > 0 else "turn\_left"

abs\_angle = abs(delta\_angle)

# Log the turning command

rover.log\_movement(direction, angle=abs\_angle)

current = rover.heading

# Normalize headings to [0, 360)

current = current % 360

target\_heading = target\_heading % 360

# Calculate clockwise and counterclockwise angles

clockwise = (target\_heading - current) % 360

counterclockwise = (current - target\_heading) % 360

# Choose the shortest direction

if clockwise <= counterclockwise:

diff = clockwise

else:

diff = -counterclockwise

# If already aligned or very close, snap to new heading without animation

if abs(diff) < 2:

rover.heading = target\_heading

return update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Calculate steps for smooth animation (fewer steps for small angles)

steps = max(3, int(abs(diff) / 10)) # Fewer steps for faster turns

step\_ang = diff / steps

# Adjust pause duration based on angle size and speed factor

pause\_duration = min(0.05, 0.03 \* (180 / max(1, abs(diff)))) / rotation\_speed\_factor

# Animate the turn

for i in range(1, steps + 1):

rover.heading = (current + step\_ang \* i) % 360

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

plt.pause(pause\_duration)

# Snap to exact final heading to avoid floating-point errors

rover.heading = target\_heading

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Store the last navigated heading

rover.last\_heading = rover.heading

return rover\_patch

class RowNavigator:

def \_\_init\_\_(self, rover):

self.rover = rover

self.interpolated\_path = []

self.current\_waypoint\_index = 0

self.waypoint\_threshold = 0.3

self.column\_spacing = 1.5

self.column\_height = 15

self.movement\_speed = 0.3

self.current\_row = 0 # Track which row we're on

self.zigzag\_pattern = True # Enable zigzag pattern by default

self.rows\_data = [] # Store information about each row

def generate\_rows(self, start\_x, start\_y, num\_strips=5, strip\_length=None, spacing=None):

if spacing is None:

spacing = self.column\_spacing

if strip\_length is None:

strip\_length = self.column\_height

bottom\_y = start\_y

top\_y = start\_y + strip\_length

self.interpolated\_path = []

self.rows\_data = [] # Reset rows data

print(f"\n🌾 Generating {num\_strips} rows with spacing {spacing:.2f}m")

print(f"🌾 Row length: {strip\_length:.2f}m")

for i in range(num\_strips):

x = start\_x + i \* spacing

# Determine row direction based on zigzag pattern

going\_up = (i % 2 == 0)

direction\_str = "↑ UP" if going\_up else "↓ DOWN"

# Create row points

if going\_up:

y\_points = np.linspace(bottom\_y, top\_y, 5)

row\_start = (x, bottom\_y)

row\_end = (x, top\_y)

else:

y\_points = np.linspace(top\_y, bottom\_y, 5)

row\_start = (x, top\_y)

row\_end = (x, bottom\_y)

# Store row data for reporting

self.rows\_data.append({

'index': i,

'x\_pos': x,

'direction': direction\_str,

'start': row\_start,

'end': row\_end

})

# Add waypoints for this row

for y in y\_points:

self.interpolated\_path.append((x, y))

# Add transition to next row if not the last row

if i < num\_strips - 1:

next\_x = start\_x + (i + 1) \* spacing

transition\_y = top\_y if going\_up else bottom\_y

self.interpolated\_path.append((next\_x, transition\_y))

# Print row information (point 4)

print("\n📋 Row Navigation Plan:")

for row in self.rows\_data:

print(f" Row {row['index']+1}: X-position {row['x\_pos']:.2f}m, Direction {row['direction']}")

return self.interpolated\_path

def distance(self, p1, p2):

return math.hypot(p2[0] - p1[0], p2[1] - p1[1])

def calculate\_heading(self, p1, p2):

return math.degrees(math.atan2(p2[1] - p1[1], p2[0] - p1[0])) % 360

def heading\_difference(self, current, target):

diff = (target - current + 540) % 360 - 180

return diff

def smooth\_turn(self, target\_heading, ax=None, fig=None, rover\_patch=None):

heading\_diff = self.heading\_difference(self.rover.heading, target\_heading)

if abs(heading\_diff) < 1:

self.rover.heading = target\_heading

self.rover.last\_heading = target\_heading # Remember heading

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

return rover\_patch

# Choose shortest turn direction and log the movement

direction = "turn\_right" if heading\_diff > 0 else "turn\_left"

self.rover.log\_movement(direction, angle=abs(heading\_diff))

# Choose shortest turn direction

steps = max(5, int(abs(heading\_diff) / 15))

angle\_step = heading\_diff / steps

for i in range(steps):

self.rover.heading = (self.rover.heading + angle\_step) % 360

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

plt.pause(0.02)

self.rover.heading = round(target\_heading, 1) # Ensure exact heading with limited decimals

self.rover.last\_heading = self.rover.heading # Remember heading

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

return rover\_patch

def move\_precisely\_to\_point(self, target\_point, ax=None, fig=None, rover\_patch=None):

max\_attempts = 40

attempts = 0

print(f"🎯 Moving to point: ({target\_point[0]:.2f}, {target\_point[1]:.2f})")

while attempts < max\_attempts:

current\_pos = (self.rover.x, self.rover.y)

dist\_to\_target = self.distance(current\_pos, target\_point)

if attempts % 5 == 0:

print(f" Distance to target: {dist\_to\_target:.2f}m")

if dist\_to\_target <= self.waypoint\_threshold:

print(f"✅ Reached target within {self.waypoint\_threshold}m threshold")

# Force exact position to avoid accumulation errors

self.rover.set\_position(target\_point[0], target\_point[1], force=True)

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

return True

desired\_heading = self.calculate\_heading(current\_pos, target\_point)

# Only turn if heading is significantly different

if abs(self.heading\_difference(self.rover.heading, desired\_heading)) > 5:

self.smooth\_turn(desired\_heading, ax, fig, rover\_patch)

move\_dist = min(self.movement\_speed, dist\_to\_target \* 0.8)

# Log the forward movement

self.rover.log\_movement("forward", distance=move\_dist)

path = [(self.rover.x, self.rover.y),

(self.rover.x + move\_dist \* math.cos(math.radians(self.rover.heading)),

self.rover.y + move\_dist \* math.sin(math.radians(self.rover.heading)))]

status, \_ = safety.check\_safety([self.rover.x, self.rover.y], self.rover.heading, path)

if status == 'safe':

success = self.rover.move\_forward(move\_dist, ax, fig, rover\_patch)

if not success:

print("⚠️ Movement failed - obstacle detected")

return False

else:

print(f"⚠️ Safety check failed: {status}")

return False

attempts += 1

plt.pause(0.01)

print("⚠️ Max attempts reached")

return False

def navigate\_to\_starting\_point(self, ax=None, fig=None, rover\_patch=None):

if not self.interpolated\_path:

return False

starting\_point = self.interpolated\_path[0]

# Enhanced terminal output (point 4.i)

print(f"\n🚩 Starting point: ({starting\_point[0]:.2f}, {starting\_point[1]:.2f})")

if self.rows\_data:

print(f"🌾 This is the beginning of Row 1, moving {self.rows\_data[0]['direction']}")

# Navigate to starting point

result = self.move\_precisely\_to\_point(starting\_point, ax, fig, rover\_patch)

if result:

self.current\_waypoint\_index = 0

self.current\_row = 0

print(f"✅ Successfully reached Row 1 starting point")

return result

def determine\_next\_task(self):

if not self.interpolated\_path or self.current\_waypoint\_index >= len(self.interpolated\_path) - 1:

return None

next\_idx = self.current\_waypoint\_index + 1

return self.interpolated\_path[next\_idx]

def align\_to\_next\_task(self, ax=None, fig=None, rover\_patch=None):

next\_point = self.determine\_next\_task()

if not next\_point:

return False

current\_pos = (self.rover.x, self.rover.y)

desired\_heading = self.calculate\_heading(current\_pos, next\_point)

# Check if we already have the right heading (avoid unnecessary turns)

current\_heading = self.rover.heading

if abs(self.heading\_difference(current\_heading, desired\_heading)) < 5:

print(f"✓ Already aligned to correct heading: {current\_heading:.1f}°")

return True

print(f"🧭 Aligning from {current\_heading:.1f}° to {desired\_heading:.1f}°")

if ax and fig and rover\_patch:

rover\_patch = self.smooth\_turn(desired\_heading, ax, fig, rover\_patch)

else:

self.smooth\_turn(desired\_heading)

print(f"✓ Aligned to heading: {self.rover.heading:.1f}°")

return True

def align\_to\_next\_task(self, ax=None, fig=None, rover\_patch=None):

next\_point = self.determine\_next\_task()

if not next\_point:

return False

desired\_heading = self.calculate\_heading((self.rover.x, self.rover.y), next\_point)

if ax and fig and rover\_patch:

rover\_patch = self.smooth\_turn(desired\_heading, ax, fig, rover\_patch)

else:

self.smooth\_turn(desired\_heading)

return True

def navigate\_all\_rows(self, ax=None, fig=None, rover\_patch=None):

if not self.interpolated\_path:

print("⚠️ No path generated - call generate\_rows first")

return False

if not self.navigate\_to\_starting\_point(ax, fig, rover\_patch):

return False

while self.current\_waypoint\_index < len(self.interpolated\_path) - 1:

next\_point = self.determine\_next\_task()

if not next\_point:

break

if not self.align\_to\_next\_task(ax, fig, rover\_patch):

continue

if self.move\_precisely\_to\_point(next\_point, ax, fig, rover\_patch):

self.current\_waypoint\_index += 1

return True

def navigate\_path(self, ax=None, fig=None, rover\_patch=None):

"""

Navigate through all waypoints in the interpolated path with enhanced reporting.

"""

print(f"🚜 COMMAND: Navigate zigzag path with {len(self.interpolated\_path)} points")

# Add command logging for each path segment

for i in range(self.current\_waypoint\_index, len(self.interpolated\_path)-1):

current\_point = self.interpolated\_path[i]

next\_point = self.interpolated\_path[i+1]

print(f"➡️ COMMAND: Move from ({current\_point[0]:.2f}, {current\_point[1]:.2f}) to ({next\_point[0]:.2f}, {next\_point[1]:.2f})")

if not self.interpolated\_path:

print("⚠️ No path generated - call generate\_rows first")

return False

# Check if we're already at a waypoint

if self.current\_waypoint\_index >= len(self.interpolated\_path):

print("⚠️ Navigation complete - already at end of path")

return True

# Track current row during navigation

current\_row\_index = 0

for i, row\_data in enumerate(self.rows\_data):

if self.current\_waypoint\_index >= i\*5: # Rough estimation

current\_row\_index = i

print(f"\n🌾 Currently at Row {current\_row\_index+1}, Waypoint {self.current\_waypoint\_index}")

print(f"🧭 Row direction: {self.rows\_data[current\_row\_index]['direction']}")

success = True

while self.current\_waypoint\_index < len(self.interpolated\_path) - 1:

next\_waypoint\_index = self.current\_waypoint\_index + 1

next\_point = self.interpolated\_path[next\_waypoint\_index]

# Determine if we're changing rows

new\_row\_index = current\_row\_index

for i, row\_data in enumerate(self.rows\_data):

if abs(next\_point[0] - row\_data['x\_pos']) < 0.1: # Close to this row's x-position

new\_row\_index = i

# Report row transition if applicable

if new\_row\_index != current\_row\_index:

print(f"\n🔄 Transitioning from Row {current\_row\_index+1} to Row {new\_row\_index+1}")

print(f"🧭 New row direction: {self.rows\_data[new\_row\_index]['direction']}")

current\_row\_index = new\_row\_index

# Navigate to next point with detailed reporting

print(f"\n🚗 Navigating to waypoint {next\_waypoint\_index} at ({next\_point[0]:.2f}, {next\_point[1]:.2f})...")

# First align to the next point

if not self.align\_to\_next\_task(ax, fig, rover\_patch):

print("⚠️ Failed to align to next waypoint")

success = False

break

# Then move to the next point

if self.move\_precisely\_to\_point(next\_point, ax, fig, rover\_patch):

print(f"✅ Reached waypoint {next\_waypoint\_index} in Row {current\_row\_index+1}")

self.current\_waypoint\_index = next\_waypoint\_index

else:

print(f"⚠️ Failed to reach waypoint {next\_waypoint\_index}")

success = False

break

# Pause briefly for visualization

if ax and fig:

plt.pause(0.05)

# Special case for the last waypoint

if success and self.current\_waypoint\_index == len(self.interpolated\_path) - 1:

print("\n✅ Successfully navigated entire path")

# Report completion of final row

final\_row = len(self.rows\_data) - 1

print(f"🎉 Completed Row {final\_row+1} - All rows navigated!")

# Make sure we're exactly at the final point

final\_point = self.interpolated\_path[-1]

if self.rover.distance\_to(\*final\_point) > 0.1: # Small tolerance

print(f"📍 Final adjustment to end point ({final\_point[0]:.2f}, {final\_point[1]:.2f})...")

self.rover.set\_position(final\_point[0], final\_point[1], force=True)

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

plt.pause(0.5) # Extended pause at the end

return success

def find\_best\_path\_angle(rover, tx, ty, blocked\_angles=None):

direct = math.degrees(math.atan2(ty - rover.y, tx - rover.x)) % 360

if not blocked\_angles or int(direct/10)\*10 not in blocked\_angles:

return direct

for off in range(10,360,10):

for sign in (1,-1):

ta = (direct+sign\*off)%360

if int(ta/10)\*10 not in blocked\_angles:

return ta

import random; return random.randint(0,359)

def get\_float(prompt):

while True:

try:

return float(input(prompt))

except ValueError:

print("⚠️ Please enter a valid number.")

def navigate\_to\_point(rover, tx, ty, ax, fig, rover\_patch=None, step\_size=STEP, tolerance=TOLERANCE):

print(f"\n🚗 Navigating to point ({tx:.3f}, {ty:.3f})...\n")

dist = rover.distance\_to(tx,ty)

attempts=0; last\_dist=float('inf'); alt=False; blocked=0

while dist>tolerance and attempts<MAX\_ATTEMPTS:

attempts+=1

rec = rover.detect\_and\_resolve\_stuck()

if rec is not None:

rover\_patch = visualize\_turn(rover, rec, ax, fig, rover\_patch)

alt=True; continue

if attempts%5==0:

if dist>last\_dist\*0.95 and not alt:

print("⚠️ Limited progress detected, trying alternative approach...")

rover.blocked\_directions.clear()

angle=(rover.heading+90+attempts%90)%360

rover\_patch=visualize\_turn(rover,angle,ax,fig,rover\_patch)

step\_size=min(step\_size\*2,dist/2); alt=True

else:

alt=False; step\_size=min(STEP,dist/2)

last\_dist=dist

if blocked>2:

tgt=find\_best\_path\_angle(rover,tx,ty,rover.blocked\_directions)

blocked=0

else:

tgt=rover.calculate\_heading\_to(tx,ty)

diff=(tgt-rover.heading+180)%360-180

if abs(diff)>5:

rover\_patch=visualize\_turn(rover,tgt,ax,fig,rover\_patch)

step=min(step\_size,dist)

# Set up movement parameters

target\_x = rover.x + step \* math.cos(math.radians(rover.heading))

target\_y = rover.y + step \* math.sin(math.radians(rover.heading))

path = [(rover.x, rover.y), (target\_x, target\_y)]

# Safety check before movement

status, recovery\_data = safety.check\_safety([rover.x, rover.y], rover.heading, path)

ok = False # Default to unsuccessful movement

if status == 'safe':

# Safe to proceed with normal movement

ok = rover.move\_forward(step, ax, fig, rover\_patch)

elif status == 'drift':

# Handle drift scenario

pos, heading, drift\_status, updated\_data = safety.handle\_drift(

[rover.x, rover.y], rover.heading, recovery\_data)

# Update rover position and visualize

rover.set\_position(pos[0], pos[1], heading, add\_to\_history=True)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Update drift data or clear it if recovered

if drift\_status == 'recovered':

ok = True

else:

recovery\_data = updated\_data

ok = False

blocked += 1

elif status in ['no-go', 'outside']:

# Handle no-go zone or boundary violation

pos, heading, violation\_status = safety.handle\_no\_go\_violation(

[rover.x, rover.y], rover.heading, recovery\_data)

# Update rover position and visualize

rover.set\_position(pos[0], pos[1], heading, add\_to\_history=True)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

if violation\_status == 'recovered':

ok = True

else:

ok = False

blocked += 1

# Update visualization

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

dist = rover.distance\_to(tx, ty)

if not ok:

blocked += 1

if blocked >= 2:

ch = 45 + blocked \* 15

ch = min(ch, 180)

rover\_patch = visualize\_turn(rover, (rover.heading + ch) % 360, ax, fig, rover\_patch)

else:

blocked = 0

if dist <= tolerance:

print(f"✅ Reached target point ({rover.x:.3f}, {rover.y:.3f})")

return True, rover\_patch

print("🔄 Making final approach attempt with larger step size...")

direct = rover.calculate\_heading\_to(tx, ty)

rover\_patch = visualize\_turn(rover, direct, ax, fig, rover\_patch)

rover.move\_forward(dist \* 0.9, ax, fig, rover\_patch)

fd = rover.distance\_to(tx, ty)

if fd <= tolerance \* 1.5:

print(f"✅ Reached target point on final attempt ({rover.x:.3f}, {rover.y:.3f})")

return True, rover\_patch

print(f"⚠️ Could not reach target point. Current position: ({rover.x:.3f}, {rover.y:.3f})")

print(f" Distance to target: {fd:.3f}")

return False, rover\_patch

def follow\_path\_precisely(rover, waypoints, ax, fig, rover\_patch):

"""

Follows the planned path with ultra-high precision by enforcing strict path adherence

Args:

rover: Rover instance

waypoints: List of (x,y) points to follow

ax: Matplotlib axis

fig: Matplotlib figure

rover\_patch: Visual representation of rover

Returns:

bool: True if path followed successfully, False otherwise

rover\_patch: Updated rover patch

"""

print(f"🛣️ COMMAND: Follow path with {len(waypoints)} waypoints")

if not waypoints or len(waypoints) < 2:

print("⚠️ Path too short or empty")

return False, rover\_patch

print(f"\n🛣️ Following planned path with {len(waypoints)} waypoints...")

# Constants for strict path following - adjusted for speed

PATH\_STEP = 0.6 # Increased step size for faster movement (was 0.2)

PATH\_TOLERANCE = 0.05 # Small tolerance to enforce strict adherence

ANIMATION\_SPEED = 0.005 # Faster animation (was 0.01)

ROTATION\_STEP\_FACTOR = 2 # Rotate faster

# Start with current position

start\_idx = 0

# Find closest waypoint if we're not already at the first one

if rover.distance\_to(\*waypoints[0]) > PATH\_TOLERANCE:

closest\_idx = 0

min\_dist = float('inf')

for i, wp in enumerate(waypoints):

dist = rover.distance\_to(\*wp)

if dist < min\_dist:

min\_dist = dist

closest\_idx = i

# If we're closer to a waypoint further along the path, start from there

if closest\_idx > 0 and min\_dist < PATH\_TOLERANCE:

start\_idx = closest\_idx

print(f"Starting from waypoint {start\_idx} which is closest to current position")

else:

# We need to first move to the first waypoint

print(f"Moving to the first waypoint at ({waypoints[0][0]:.2f}, {waypoints[0][1]:.2f})")

initial\_heading = rover.calculate\_heading\_to(\*waypoints[0])

rover\_patch = visualize\_turn(rover, initial\_heading, ax, fig, rover\_patch, rotation\_speed\_factor=ROTATION\_STEP\_FACTOR)

# Don't teleport - move properly to first waypoint

init\_distance = rover.distance\_to(\*waypoints[0])

if init\_distance > PATH\_TOLERANCE:

segments = max(2, int(init\_distance / PATH\_STEP))

step\_dist = init\_distance / segments

for \_ in range(segments):

success = rover.move\_forward(step\_dist, ax, fig, rover\_patch)

if not success:

# If blocked, try with smaller steps

half\_step = step\_dist / 2

if half\_step > 0.1: # Don't try with too small steps

success = rover.move\_forward(half\_step, ax, fig, rover\_patch)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

plt.pause(ANIMATION\_SPEED)

# For path visualization

actual\_path = []

path\_line = None

# Traverse waypoints

for i in range(start\_idx, len(waypoints)-1):

current\_wp = waypoints[i]

next\_wp = waypoints[i+1]

print(f"\n📍 Moving from waypoint {i} to {i+1}: ({current\_wp[0]:.2f}, {current\_wp[1]:.2f}) → ({next\_wp[0]:.2f}, {next\_wp[1]:.2f})")

# Calculate segment vector and length

segment\_vec = (next\_wp[0] - current\_wp[0], next\_wp[1] - current\_wp[1])

segment\_len = math.hypot(\*segment\_vec)

if segment\_len < 0.01: # Skip tiny segments

continue

# Unit vector along segment

unit\_vec = (segment\_vec[0]/segment\_len, segment\_vec[1]/segment\_len)

# Align precisely to segment direction with faster rotation

segment\_heading = math.degrees(math.atan2(segment\_vec[1], segment\_vec[0])) % 360

rover\_patch = visualize\_turn(rover, segment\_heading, ax, fig, rover\_patch, rotation\_speed\_factor=ROTATION\_STEP\_FACTOR)

# Before starting segment, ensure we're exactly at the start point (if not already there)

if rover.distance\_to(\*current\_wp) > PATH\_TOLERANCE:

# Move to start point without teleporting

remaining\_dist = rover.distance\_to(\*current\_wp)

segments = max(2, int(remaining\_dist / PATH\_STEP))

step\_dist = remaining\_dist / segments

for \_ in range(segments):

if rover.distance\_to(\*current\_wp) <= PATH\_TOLERANCE:

break

success = rover.move\_forward(step\_dist, ax, fig, rover\_patch)

if not success:

# Try with smaller step if blocked

rover.move\_forward(step\_dist/2, ax, fig, rover\_patch)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

plt.pause(ANIMATION\_SPEED)

# Calculate appropriate number of interpolated points for this segment

num\_interp = max(3, int(segment\_len / PATH\_STEP))

# Move along the segment with precise steps

for j in range(1, num\_interp + 1):

t = j / num\_interp

interp\_point = (

current\_wp[0] + t \* segment\_vec[0],

current\_wp[1] + t \* segment\_vec[1]

)

# Always ensure heading is aligned with path

point\_heading = rover.calculate\_heading\_to(\*interp\_point)

if abs((point\_heading - rover.heading + 180) % 360 - 180) > 1:

rover\_patch = visualize\_turn(rover, point\_heading, ax, fig, rover\_patch,

rotation\_speed\_factor=ROTATION\_STEP\_FACTOR)

# Calculate exact distance to move

move\_dist = rover.distance\_to(\*interp\_point)

# Move to interpolated point without teleporting

if move\_dist > PATH\_TOLERANCE:

success = rover.move\_forward(move\_dist, ax, fig, rover\_patch)

# If direct movement fails, try with smaller steps

if not success and move\_dist > PATH\_STEP:

smaller\_step = min(PATH\_STEP, move\_dist/2)

success = rover.move\_forward(smaller\_step, ax, fig, rover\_patch)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Visualize the actual path

actual\_path.append((rover.x, rover.y))

if len(actual\_path) > 1 and path\_line:

safe\_remove(path\_line)

if len(actual\_path) > 1:

path\_x, path\_y = zip(\*actual\_path)

path\_line = ax.plot(path\_x, path\_y, 'g-', linewidth=1, alpha=0.7)[0]

fig.canvas.draw\_idle()

plt.pause(ANIMATION\_SPEED)

# Safety check if available

if hasattr(safety, 'check\_safety'):

status, \_ = safety.check\_safety([rover.x, rover.y], rover.heading, [(rover.x, rover.y), interp\_point])

if status != 'safe':

print("⚠️ Safety violation detected during path following!")

return False, rover\_patch

# For the final waypoint, use exact positioning with proper movement

last\_wp = waypoints[-1]

final\_heading = rover.calculate\_heading\_to(\*last\_wp)

rover\_patch = visualize\_turn(rover, final\_heading, ax, fig, rover\_patch, rotation\_speed\_factor=ROTATION\_STEP\_FACTOR)

# Move directly to last waypoint without teleporting

final\_dist = rover.distance\_to(\*last\_wp)

if final\_dist > PATH\_TOLERANCE:

# Break into smaller steps

segments = max(2, int(final\_dist / PATH\_STEP))

step\_dist = final\_dist / segments

for \_ in range(segments):

if rover.distance\_to(\*last\_wp) <= PATH\_TOLERANCE:

break

success = rover.move\_forward(step\_dist, ax, fig, rover\_patch)

if not success:

# Try smaller step if blocked

rover.move\_forward(step\_dist/2, ax, fig, rover\_patch)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

plt.pause(ANIMATION\_SPEED)

print("✅ Successfully followed the planned path with precision")

return True, rover\_patch

def safe\_remove(element):

if element:

try:

element.remove()

return True

except:

if DEBUG: print(f"Warning: failed to remove {element}")

return False

def run\_simulation():

plt.rcParams['figure.max\_open\_warning'] = 50

rover = Rover()

print("🚜 Farm Rover Navigation Simulation 🚜")

print("=====================================")

farm\_width = get\_float(" Farm width: ")

farm\_height = get\_float(" Farm height: ")

min\_x = -farm\_width / 2

max\_x = farm\_width / 2

min\_y = -farm\_height / 2

max\_y = farm\_height / 2

verts = [(min\_x, min\_y), (max\_x, min\_y), (max\_x, max\_y), (min\_x, max\_y)]

entry\_point = (0, min\_y)

rover.set\_geofence(verts, entry\_point)

safety.set\_geofence(verts)

plt.ion()

fig, ax = plt.subplots(figsize=(10, 8))

ax.set\_title("Rover Farm Navigation Simulation")

margin = max(farm\_width, farm\_height) \* 0.2

ax.set\_xlim(min\_x - margin, max\_x + margin)

ax.set\_ylim(min\_y - margin, max\_y + margin)

ax.grid(True)

fence = Polygon(np.array(verts), closed=True, facecolor='lightgreen', edgecolor='darkgreen', alpha=0.3)

ax.add\_patch(fence)

ax.scatter(entry\_point[0], entry\_point[1], c='purple', s=100, marker='o', label='Farm Entry', zorder=10)

path\_line, = ax.plot([], [], 'b-', alpha=0.5, label='Path')

ax.path\_line = path\_line

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(0.5)

rover.set\_position(entry\_point[0], entry\_point[1], force=True, add\_to\_history=False)

rover.history.append((rover.x, rover.y))

rover\_patch = update\_rover\_visualization(rover, ax, fig)

print(f"\n✅ Starting simulation at farm entry point: ({entry\_point[0]:.2f}, {entry\_point[1]:.2f})")

spacing = 1.5

start\_x = min\_x

start\_y = min\_y

strip\_length = max\_y - min\_y

num\_strips = math.floor((max\_x - min\_x) / spacing) + 1

row\_navigator = RowNavigator(rover)

row\_navigator.generate\_rows(start\_x, start\_y, num\_strips, strip\_length, spacing)

row\_navigator.navigate\_all\_rows(ax, fig, rover\_patch)

plt.ioff()

plt.show(block=True)

if \_\_name\_\_ == "\_\_main\_\_":

try:

run\_simulation()

except KeyboardInterrupt:

print("\n\n🛑 Simulation terminated by user.")

except Exception as e:

print(f"\n❌ Simulation error: {e}")

FARM\_SIMULATION

import matplotlib.pyplot as plt

import numpy as np

import random

from astar\_algo import AStarPlanner

# Import our modules

from row\_navigation import Rover, navigate\_to\_point, TOLERANCE, follow\_path\_precisely, update\_rover\_visualization, visualize\_turn

from row\_navigation import RowNavigator

from farm\_safety import SafetyModule

debug = False

safety = SafetyModule()

def get\_float(prompt):

"""Get a float value from user with error handling"""

while True:

try:

value = float(input(prompt))

return value

except ValueError:

print("⚠️ Please enter a valid number.")

def random\_position\_in\_farm(min\_x, max\_x, min\_y, max\_y, safety\_margin=2.0):

"""Generate a random position inside the farm with a safety margin from boundaries"""

x = random.uniform(min\_x + safety\_margin, max\_x - safety\_margin)

y = random.uniform(min\_y + safety\_margin, max\_y - safety\_margin)

return x, y

def safe\_remove(element):

if element:

try:

element.remove()

return True

except:

if debug: print(f"Warning: failed to remove {element}")

return False

def run\_simulation():

print("🚜 Farm Rover Navigation Simulation 🚜")

print("=====================================")

plt.rcParams['figure.max\_open\_warning'] = 50

# Create the rover

rover = Rover()

# Setup the farm boundaries (only input required from user)

print("\n🔧 Enter farm rectangle coordinates:")

min\_x = get\_float(" Min X: ")

max\_x = get\_float(" Max X: ")

min\_y = get\_float(" Min Y: ")

max\_y = get\_float(" Max Y: ")

# Create vertices for the farm boundary

verts = [(min\_x, min\_y), (max\_x, min\_y), (max\_x, max\_y), (min\_x, max\_y)]

# Generate a random entry point (we'll still set this for compatibility even though not used)

# Choose a random side and position on that side

side = random.randint(0, 3)

if side == 0: # Bottom side

entry\_x = random.uniform(min\_x, max\_x)

entry\_y = min\_y

elif side == 1: # Right side

entry\_x = max\_x

entry\_y = random.uniform(min\_y, max\_y)

elif side == 2: # Top side

entry\_x = random.uniform(min\_x, max\_x)

entry\_y = max\_y

else: # Left side

entry\_x = min\_x

entry\_y = random.uniform(min\_y, max\_y)

entry\_point = (entry\_x, entry\_y)

# Set geofence in rover and safety module

rover.set\_geofence(verts, entry\_point)

safety.set\_geofence(verts)

# Remove no-go zone creation

# The following block is commented out to remove the reddish square

"""

# Add a rectangular no-go zone in the middle of the farm

center\_x = (min\_x + max\_x) / 2

center\_y = (min\_y + max\_y) / 2

size = 1.5 # Size of the no-go zone

safety.add\_no\_go\_zone(center\_x - size, center\_y - size, center\_x + size, center\_y + size)

"""

# Generate random starting position inside the farm

random\_x, random\_y = random\_position\_in\_farm(min\_x, max\_x, min\_y, max\_y)

print(f"🎲 Randomly placing rover inside farm at: ({random\_x:.3f}, {random\_y:.3f})")

# Initialize visualization

plt.ion()

fig, ax = plt.subplots(figsize=(10, 8))

ax.set\_title("Rover Farm Navigation Simulation")

# Draw farm boundary

farm\_polygon = plt.Polygon(np.array(verts), closed=True,

facecolor='lightgreen', edgecolor='darkgreen', alpha=0.3)

ax.add\_patch(farm\_polygon)

# Remove no-go zone visualization

# The following block is commented out to remove the reddish square

"""

# Draw no-go zone

no\_go\_verts = [

(center\_x - size, center\_y - size),

(center\_x + size, center\_y - size),

(center\_x + size, center\_y + size),

(center\_x - size, center\_y + size)

]

no\_go\_polygon = plt.Polygon(np.array(no\_go\_verts), closed=True,

facecolor='red', edgecolor='darkred', alpha=0.3)

ax.add\_patch(no\_go\_polygon)

"""

# Mark random start position

ax.scatter(random\_x, random\_y, c='green', s=80, label='Start (Inside)')

# Setup plot limits and grid

margin = 3

ax.set\_xlim(min\_x - margin, max\_x + margin)

ax.set\_ylim(min\_y - margin, max\_y + margin)

ax.grid(True)

# Setup rover path visualization

path\_line, = ax.plot([], [], 'b-', alpha=0.5, label='Path')

ax.path\_line = path\_line

ax.legend(loc='upper left')

# Set rover starting position (inside farm)

rover.set\_position(random\_x, random\_y, force=True, add\_to\_history=False)

rover.inside\_fence = True # Force the rover to be considered inside the farm

rover.fence\_locked = True # Lock the rover inside the farm

rover.history.append((rover.x, rover.y))

rover\_patch = update\_rover\_visualization(rover, ax, fig)

print("\n🚜 TASK 1: Determining farm navigation plan with zigzag pattern...\n")

# Create row navigator

navigator = RowNavigator(rover)

navigator.zigzag\_pattern = True # Ensure zigzag pattern is enabled

# Generate rows within the farm using zigzag pattern

row\_start\_x = min\_x + 2 # Start rows 2 units from left edge

row\_start\_y = min\_y + 2 # Start rows 2 units from bottom edge

row\_spacing = 1.5

num\_strips = max(3, min(10, int((max\_x - min\_x - 4) / row\_spacing))) # Calculate number of strips based on farm width

# Generate rows and visualize them

rows = navigator.generate\_rows(

row\_start\_x, row\_start\_y,

num\_strips=num\_strips,

strip\_length=max\_y - min\_y - 4 # Strip height based on farm height

)

safety.set\_waypoints(navigator.interpolated\_path)

# Visualize zigzag row pattern

x\_coords, y\_coords = zip(\*navigator.interpolated\_path)

ax.plot(x\_coords, y\_coords, 'b-', alpha=0.5, label='Zig-Zag Path')

# Mark start and end points

path\_start = navigator.interpolated\_path[0]

path\_end = navigator.interpolated\_path[-1]

ax.scatter(path\_start[0], path\_start[1], c='orange', s=50, marker='s', label='Path Start')

ax.scatter(path\_end[0], path\_end[1], c='red', s=50, marker='o', label='Path End')

fig.canvas.draw\_idle()

plt.pause(0.5)

# --- TASK 1: Navigate directly to the path start point ---

print("\n🚜 TASK 1: Navigating directly to path start point...\n")

print(f"🎯 Path start point: ({path\_start[0]:.3f}, {path\_start[1]:.3f})")

print(f"📏 Distance to path start: {rover.distance\_to(\*path\_start):.3f}m")

# Navigate to path start

def navigate\_to\_path\_start(rover, safety, path\_start, ax, fig, rover\_patch):

"""

Navigate rover to the starting point of the path

"""

# Save original position

original\_x, original\_y = rover.x, rover.y

original\_heading = rover.heading

# Create A\* planner - temporarily set path\_start as the entry point for planning

temp\_entry = (path\_start[0], path\_start[1])

# Save original entry point

original\_entry = rover.entry\_point

# Temporarily set entry point to path\_start for A\* planning

rover.entry\_point = temp\_entry

print("\n🗺️ Planning path to starting point...")

planner = AStarPlanner(rover, safety, cell\_size=0.2, padding=2.5)

# Restore the original entry point

rover.entry\_point = original\_entry

# Plan the path

waypoints = planner.plan()

if waypoints:

print(f"✅ Found path to starting point with {len(waypoints)} waypoints")

# Visualize the planned path

path\_x, path\_y = zip(\*waypoints)

planned\_path\_line = ax.plot(path\_x, path\_y, 'y--', linewidth=2, alpha=0.7, label='Path to Start')[0]

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(0.5)

# Navigate to path start

reached\_start, rover\_patch = follow\_path\_precisely(rover, waypoints, ax, fig, rover\_patch)

# Clean up after path following

safe\_remove(planned\_path\_line)

# Final approach to exact path start

if not reached\_start or rover.distance\_to(\*path\_start) > TOLERANCE:

reached\_start, rover\_patch = navigate\_to\_point(

rover, path\_start[0], path\_start[1], ax, fig, rover\_patch)

return reached\_start, rover\_patch

else:

# Direct navigation as fallback

print("⚠️ Failed to plan path to starting point. Attempting direct navigation.")

reached\_start, rover\_patch = navigate\_to\_point(

rover, path\_start[0], path\_start[1], ax, fig, rover\_patch)

return reached\_start, rover\_patch

# Use our custom function to navigate to path start

reached\_start, rover\_patch = navigate\_to\_path\_start(rover, safety, path\_start, ax, fig, rover\_patch)

if not reached\_start:

print("\n⚠️ Could not reach path start point after multiple attempts.")

print(" Try adjusting simulation parameters or path positioning.")

return

# Force rover position to exactly match path start

rover.set\_position(path\_start[0], path\_start[1], force=True)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Mark path start reached

ax.scatter(path\_start[0], path\_start[1], c='lime', s=80, marker='\*', label='Start Reached')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

print("\n✅ TASK 1 COMPLETE: Successfully reached path start point")

print(f" Current position: ({rover.x:.3f}, {rover.y:.3f})")

# --- TASK 2: Align to the path direction ---

print("\n🚜 TASK 2: Aligning rover to path direction...\n")

# Find next waypoint (should be index 1 since we're at index 0)

navigator.current\_waypoint\_index = 0 # Force to start at the beginning of the path

next\_point = navigator.interpolated\_path[1]

desired\_heading = navigator.calculate\_heading((rover.x, rover.y), next\_point)

# Align to the path direction

rover\_patch = visualize\_turn(rover, desired\_heading, ax, fig, rover\_patch)

print(f" Aligned rover to heading: {desired\_heading:.1f}°")

print("\n✅ TASK 2 COMPLETE: Successfully aligned to path direction")

# --- TASK 3: Navigate through the path ---

print("\n🚜 TASK 3: Starting path navigation pattern...\n")

# Start navigation from the beginning of the path

navigator.current\_waypoint\_index = 0

path\_success = navigator.navigate\_path(ax, fig, rover\_patch)

if not path\_success:

print("\n⚠️ Failed to navigate path. Simulation halted.")

return

# Mark completion of path

final\_point = navigator.interpolated\_path[-1]

ax.scatter(final\_point[0], final\_point[1], c='green', s=100, marker='\*', label='Mission Complete')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

print("\n🎉 TASK 3 COMPLETE: Successfully navigated the path")

print("\n🏁 SIMULATION COMPLETE! 🏁")

print(f" Total commands executed: {rover.command\_count}")

print(f" Final position: ({rover.x:.3f}, {rover.y:.3f})")

# Keep plot open until closed manually

plt.ioff()

plt.show(block=True)

if \_\_name\_\_ == "\_\_main\_\_":

try:

run\_simulation()

except KeyboardInterrupt:

print("\n\n🛑 Simulation terminated by user.")

except Exception as e:

print(f"\n❌ Simulation error: {e}")

ASTAR ALGO  
  
# A\* Path Planner for Rover Entry Navigation with Safety Integration

# --------------------------------------------------

# This module implements an occupancy-grid A\* planner that finds a collision-free path

# from the rover's current position to the farm entry point, avoiding both the farm boundary

# and any configured no-go zones in the SafetyModule.

import heapq

import math

import numpy as np

class AStarPlanner:

def \_\_init\_\_(self, rover, safety, cell\_size=0.2, padding=2.5):

"""

rover: instance of Rover with .x, .y, .entry\_point, .geofence (list of (x,y) vertices)

safety: instance of SafetyModule containing no-go zones & geofence

cell\_size: resolution of grid in meters (smaller for better accuracy)

padding: extra margin around bounding box

"""

self.rover = rover

self.safety = safety

self.cell = cell\_size

self.margin = padding

self.\_build\_grid()

def \_build\_grid(self):

"""Build the occupancy grid for path planning"""

# Determine bounding box combining rover pos, entry point, geofence, and no-go zones

xs = [self.rover.x, self.rover.entry\_point[0]] + [v[0] for v in self.safety.geofence]

ys = [self.rover.y, self.rover.entry\_point[1]] + [v[1] for v in self.safety.geofence]

# Include no-go zone bounds if they exist

if hasattr(self.safety, 'no\_go\_zones') and self.safety.no\_go\_zones:

for (x0, y0, x1, y1) in self.safety.no\_go\_zones:

xs += [x0, x1]

ys += [y0, y1]

self.x\_min = min(xs) - self.margin

self.x\_max = max(xs) + self.margin

self.y\_min = min(ys) - self.margin

self.y\_max = max(ys) + self.margin

# Compute grid dimensions

self.nx = int(math.ceil((self.x\_max - self.x\_min) / self.cell))

self.ny = int(math.ceil((self.y\_max - self.y\_min) / self.cell))

# Initialize occupancy grid: 0=free, 1=obstacle

self.grid = np.zeros((self.ny, self.nx), dtype=np.uint8)

# Edge buffer thickness (in cells) for the farm boundary

edge\_buffer = max(1, int(0.3 / self.cell))

# Mark farm boundary and obstacles in grid

for i in range(self.ny):

for j in range(self.nx):

x = self.x\_min + (j + 0.5) \* self.cell

y = self.y\_min + (i + 0.5) \* self.cell

# Mark farm boundary with buffer zone

if self.\_is\_near\_fence\_edge(x, y, self.safety.geofence, edge\_buffer \* self.cell):

self.grid[i, j] = 1

# Mark no-go zones

if hasattr(self.safety, 'no\_go\_zones') and self.safety.no\_go\_zones:

for (x0, y0, x1, y1) in self.safety.no\_go\_zones:

if x0 <= x <= x1 and y0 <= y <= y1:

self.grid[i, j] = 1

# Make entry point traversable along with a small area around it

entry\_cell = self.\_to\_cell(\*self.rover.entry\_point)

if 0 <= entry\_cell[0] < self.ny and 0 <= entry\_cell[1] < self.nx:

# Clear entry point cell

self.grid[entry\_cell[0], entry\_cell[1]] = 0

# Clear cells in small radius around entry point for easier approach

entry\_radius = max(1, int(0.5 / self.cell)) # 0.5m radius

for di in range(-entry\_radius, entry\_radius+1):

for dj in range(-entry\_radius, entry\_radius+1):

ni, nj = entry\_cell[0] + di, entry\_cell[1] + dj

if 0 <= ni < self.ny and 0 <= nj < self.nx:

# Only clear if distance to entry is within radius

if di\*di + dj\*dj <= entry\_radius\*entry\_radius:

self.grid[ni, nj] = 0

# Ensure start cell is free (rover's current position)

self.start = self.\_to\_cell(self.rover.x, self.rover.y)

if 0 <= self.start[0] < self.ny and 0 <= self.start[1] < self.nx:

self.grid[self.start[0], self.start[1]] = 0

# Ensure goal cell is free (entry point)

self.goal = self.\_to\_cell(\*self.rover.entry\_point)

if 0 <= self.goal[0] < self.ny and 0 <= self.goal[1] < self.nx:

self.grid[self.goal[0], self.goal[1]] = 0

# Debug info

print(f"Grid size: {self.ny}x{self.nx} cells, cell size: {self.cell}m")

print(f"Start cell: {self.start}, Goal cell: {self.goal}")

print(f"Entry point: {self.rover.entry\_point}")

print(f"Grid value at goal: {self.grid[self.goal] if 0 <= self.goal[0] < self.ny and 0 <= self.goal[1] < self.nx else 'out of bounds'}")

def \_is\_near\_fence\_edge(self, x, y, vertices, threshold):

"""Check if point (x,y) is near any edge of the polygon defined by vertices"""

n = len(vertices)

for i in range(n):

x1, y1 = vertices[i]

x2, y2 = vertices[(i+1) % n]

# Calculate distance from point to line segment

px, py = x, y

# Vector from point 1 to point 2

dx = x2 - x1

dy = y2 - y1

line\_length\_sq = dx\*dx + dy\*dy

# If segment is a point, calculate distance to that point

if line\_length\_sq < 1e-10:

dist = math.hypot(px - x1, py - y1)

else:

# Project point onto line

t = max(0, min(1, ((px - x1) \* dx + (py - y1) \* dy) / line\_length\_sq))

# Calculate closest point on line segment

closest\_x = x1 + t \* dx

closest\_y = y1 + t \* dy

# Distance from point to closest point on line

dist = math.hypot(px - closest\_x, py - closest\_y)

if dist < threshold:

return True

return False

def \_to\_cell(self, x, y):

"""Convert world coordinates to grid cell indices"""

j = int((x - self.x\_min) / self.cell)

i = int((y - self.y\_min) / self.cell)

# Ensure indices are within grid bounds

i = max(0, min(i, self.ny-1))

j = max(0, min(j, self.nx-1))

return (i, j)

def \_to\_xy(self, i, j):

"""Convert grid cell indices to world coordinates (cell center)"""

x = self.x\_min + (j + 0.5) \* self.cell

y = self.y\_min + (i + 0.5) \* self.cell

return x, y

def plan(self):

"""

Executes A\* search and returns a list of (x, y) waypoints from current position to entry point.

Returns None if no path is found.

"""

if self.start == self.goal:

# Already at goal, return just the goal point

return [self.\_to\_xy(\*self.goal)]

# Heuristic function: Euclidean distance

h = lambda a, b: math.hypot(a[0]-b[0], a[1]-b[1])

# Initialize priority queue

open\_heap = [] # elements: (f\_score, node\_id, (i,j))

node\_id = 0 # To break ties in priority queue

heapq.heappush(open\_heap, (h(self.start, self.goal), node\_id, self.start))

node\_id += 1

# Initialize tracking dictionaries

came\_from = {} # Parent pointers for path reconstruction

g\_score = {} # Cost from start to node

g\_score[self.start] = 0

closed = set() # Set of fully expanded nodes

# Max iterations to prevent infinite loops

max\_iterations = self.nx \* self.ny \* 2

iterations = 0

while open\_heap and iterations < max\_iterations:

iterations += 1

# Get node with lowest f\_score

f, \_, current = heapq.heappop(open\_heap)

# Skip if already processed

if current in closed:

continue

# Goal check

if current == self.goal:

path = self.\_reconstruct(came\_from, current)

return self.\_smooth\_path(path)

# Mark as processed

closed.add(current)

# Generate neighbors in 8 directions

ci, cj = current

for di, dj in [(-1,0),(1,0),(0,-1),(0,1),(-1,-1),(-1,1),(1,-1),(1,1)]:

ni, nj = ci+di, cj+dj

# Check bounds

if not (0 <= ni < self.ny and 0 <= nj < self.nx):

continue

# Check if obstacle

if self.grid[ni, nj] == 1:

continue

# Calculate cost to neighbor (diagonal moves cost more)

move\_cost = math.sqrt(di\*di + dj\*dj)

tentative\_g = g\_score.get(current, float('inf')) + move\_cost

neighbor = (ni, nj)

# Skip if we already found a better path to this neighbor

if neighbor in closed and tentative\_g >= g\_score.get(neighbor, float('inf')):

continue

# Update if this is a better path

if tentative\_g < g\_score.get(neighbor, float('inf')):

g\_score[neighbor] = tentative\_g

f\_score = tentative\_g + h(neighbor, self.goal)

came\_from[neighbor] = current

heapq.heappush(open\_heap, (f\_score, node\_id, neighbor))

node\_id += 1

print(f"A\* search terminated after {iterations} iterations without finding path")

return None

def \_reconstruct(self, came\_from, current):

"""Reconstruct path from goal to start using parent pointers"""

path = [current]

while current in came\_from:

current = came\_from[current]

path.append(current)

path.reverse()

# Convert to real-world coordinates

return [self.\_to\_xy(i, j) for (i, j) in path]

def \_smooth\_path(self, path):

"""Apply path smoothing to reduce unnecessary waypoints"""

if len(path) <= 2:

return path

# Keep start and end points

result = [path[0]]

# Look ahead to find direct paths

i = 0

while i < len(path) - 1:

current = path[i]

# Try to find furthest point we can directly reach

furthest = i + 1

for j in range(i + 2, len(path)):

# Check if direct path from current to path[j] is collision-free

if self.\_is\_line\_free(current, path[j]):

furthest = j

else:

break

# Add the furthest reachable point and continue from there

result.append(path[furthest])

i = furthest

return result

def \_is\_line\_free(self, point1, point2):

"""Check if line between two points is collision-free"""

x1, y1 = point1

x2, y2 = point2

# Get vector and length

dx, dy = x2 - x1, y2 - y1

length = math.hypot(dx, dy)

# Skip if points are too close

if length < self.cell:

return True

# Sample points along line

steps = max(5, int(length / (self.cell \* 0.5)))

for i in range(steps + 1):

t = i / steps

x = x1 + t \* (x2 - x1)

y = y1 + t \* (y2 - y1)

# Convert to grid cell

cell = self.\_to\_cell(x, y)

# Check if cell is obstacle

if self.grid[cell[0], cell[1]] == 1:

return False

return True

def visualize\_grid(self, ax=None):

"""Visualize the occupancy grid for debugging"""

import matplotlib.pyplot as plt

created\_fig = False

if ax is None:

fig, ax = plt.subplots(figsize=(10, 8))

created\_fig = True

# Create a visualizable grid (transpose for correct orientation)

vis\_grid = np.zeros((self.ny, self.nx, 3), dtype=np.uint8)

# Mark free space as light gray

vis\_grid[self.grid == 0] = [240, 240, 240]

# Mark obstacles as black

vis\_grid[self.grid == 1] = [0, 0, 0]

# Mark start as green

if 0 <= self.start[0] < self.ny and 0 <= self.start[1] < self.nx:

vis\_grid[self.start[0], self.start[1]] = [0, 255, 0]

# Mark goal as red

if 0 <= self.goal[0] < self.ny and 0 <= self.goal[1] < self.nx:

vis\_grid[self.goal[0], self.goal[1]] = [255, 0, 0]

ax.imshow(vis\_grid, origin='lower')

ax.set\_title("A\* Planning Grid")

# Add gridlines

ax.grid(True, which='both', color='lightgray', linewidth=0.5)

ax.set\_xticks(np.arange(-.5, self.nx, 1))

ax.set\_yticks(np.arange(-.5, self.ny, 1))

ax.set\_xticklabels([])

ax.set\_yticklabels([])

# Mark start and goal with text

if 0 <= self.start[0] < self.ny and 0 <= self.start[1] < self.nx:

ax.text(self.start[1], self.start[0], "S", ha='center', va='center', color='white', fontsize=10, fontweight='bold')

if 0 <= self.goal[0] < self.ny and 0 <= self.goal[1] < self.nx:

ax.text(self.goal[1], self.goal[0], "G", ha='center', va='center', color='white', fontsize=10, fontweight='bold')

if created\_fig:

plt.tight\_layout()

return ax

# Usage in main simulation:

# from astar\_algo import AStarPlanner

# planner = AStarPlanner(rover, safety, cell\_size=0.2, padding=2.5)

# waypoints = planner.plan()

# if waypoints:

# for wx, wy in waypoints[1:]: # Skip first waypoint (current position)

# navigate\_to\_point(rover, wx, wy, ax, fig, rover\_patch)

import math

import random

class SafetyModule:

"""

A module to handle drift and enforce no-go rules in farm rover navigation.

No-go now covers:

1) Revisiting a waypoint (in the user-defined waypoint list).

2) Exiting the farm boundary.

Drift logic is unchanged.

"""

def \_\_init\_\_(self, revisit\_threshold=0.2):

# Drift configuration

self.turn\_rate\_per\_cm = 20 # degrees per cm of turn capability

self.distance\_per\_step = 0.2 # cm traveled per simulation step

self.lookahead\_steps = 5 # baseline lookahead for drift recovery

self.drift\_probability = 0.05 # probability of drift occurring per check

# Geofence

self.geofence = None # farm boundary polygon vertices

# Waypoint tracking for revisit prevention

self.waypoints = [] # list of (x,y) waypoints

self.visited\_wp\_indices = set() # indices of waypoints already visited

self.revisit\_threshold = revisit\_threshold

# History logs

self.violations\_history = [] # track no-go incidents

self.drift\_history = [] # track drift incidents

def set\_geofence(self, vertices):

"""Set the farm boundary as a polygon"""

self.geofence = vertices

def set\_waypoints(self, waypoints):

"""Provide the ordered list of navigation waypoints"""

self.waypoints = waypoints

self.visited\_wp\_indices.clear()

def is\_outside\_geofence(self, pos):

"""Check if a position is outside the farm boundary polygon"""

if not self.geofence:

return False

x, y = pos

inside = False

n = len(self.geofence)

p1x, p1y = self.geofence[0]

for i in range(1, n+1):

p2x, p2y = self.geofence[i % n]

if y > min(p1y, p2y):

if y <= max(p1y, p2y) and x <= max(p1x, p2x):

if p1y != p2y:

xinters = (y - p1y) \* (p2x - p1x) / (p2y - p1y) + p1x

if p1x == p2x or x <= xinters:

inside = not inside

p1x, p1y = p2x, p2y

return not inside

def check\_safety(self, pos, heading, path):

"""

Check if a planned move is safe.

Returns:

- status: 'safe', 'drift', or 'no-go'

- data: info for recovery or violation handling

"""

# Candidate next position

next\_pos = path[-1]

# 1) No-go: revisiting an already visited waypoint

for idx, wp in enumerate(self.waypoints):

if math.hypot(next\_pos[0]-wp[0], next\_pos[1]-wp[1]) <= self.revisit\_threshold:

if idx in self.visited\_wp\_indices:

self.violations\_history.append(('no-go-revisit', next\_pos, heading))

return 'no-go', {

'violation\_type': 'revisit',

'pos': next\_pos,

'heading': heading

}

else:

# mark this waypoint as visited now

self.visited\_wp\_indices.add(idx)

break

# 2) No-go: exiting farm boundary

if self.is\_outside\_geofence(next\_pos):

self.violations\_history.append(('no-go-boundary', next\_pos, heading))

return 'no-go', {

'violation\_type': 'boundary',

'pos': next\_pos,

'heading': heading

}

# 3) Potential drift

if path and random.random() < self.drift\_probability:

closest\_idx, \_ = self.find\_closest\_point\_on\_path(pos, path)

if closest\_idx < len(path) - self.lookahead\_steps:

self.drift\_history.append(('drift', pos.copy(), heading))

drift\_angle = 45 if random.choice([True, False]) else -45

trigger\_idx = closest\_idx

end\_idx = min(trigger\_idx + self.lookahead\_steps, len(path)-1)

dx = path[end\_idx][0] - path[trigger\_idx][0]

dy = path[end\_idx][1] - path[trigger\_idx][1]

mag = math.hypot(dx, dy)

ux, uy = (dx/mag, dy/mag) if mag else (1.0, 0.0)

rad = math.radians(drift\_angle)

c, s = math.cos(rad), math.sin(rad)

rx = ux\*c + uy\*s

ry = -ux\*s + uy\*c

turn\_dist = abs(drift\_angle) / self.turn\_rate\_per\_cm

extra\_skip = int(turn\_dist / self.distance\_per\_step)

recovery\_idx = min(trigger\_idx + self.lookahead\_steps + extra\_skip, len(path)-1)

recovery\_target = path[recovery\_idx]

return 'drift', {

'trigger\_idx': trigger\_idx,

'drift\_angle': drift\_angle,

'drift\_vector': (rx, ry),

'recovery\_idx': recovery\_idx,

'recovery\_target': recovery\_target,

'path': path

}

# 4) Safe to proceed

return 'safe', None

def handle\_drift(self, pos, heading, drift\_data):

"""(Unchanged) Simulate drift and guide recovery."""

# ... existing drift handler code ...

raise NotImplementedError

def handle\_no\_go\_violation(self, pos, heading, violation\_data):

"""(Unchanged) Back away from forbidden position."""

# ... existing no-go handler code ...

raise NotImplementedError

def find\_closest\_point\_on\_path(self, pos, path):

"""Find the path index closest to pos."""

min\_dist, min\_idx = float('inf'), 0

for i, p in enumerate(path):

d = math.hypot(p[0]-pos[0], p[1]-pos[1])

if d < min\_dist:

min\_dist, min\_idx = d, i

return min\_idx, path[min\_idx]

def diff\_h(self, c, t):

return (t - c + 540) % 360 - 180