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

import csv

import os

from datetime import datetime

import random

from collections import OrderedDict

safety = SafetyModule()

STEP = 1.6 # 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 = 0.001 # 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

self.navigator = None

# 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

self.x = round(x, 2)

self.y = round(y, 2)

if hasattr(self, 'failsafe'):

self.failsafe.update\_gps\_status(has\_fix=True, satellites=10, hdop=1.0)

self.failsafe.update\_internet\_status(connected=True, latency=0.1)

self.failsafe.update\_module\_communication()

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

# Get compass direction - using standard compass conversion

compass\_direction = self.get\_compass\_direction(self.heading)

# Calculate standard compass bearing for display

standard\_bearing = (90 - self.heading) % 360

# Add GPS-like position reporting with compass direction and standard bearing

print(f"📍 GPS: Position [{self.x:.2f}, {self.y:.2f}], Heading: {self.heading:.1f}° (Compass: {standard\_bearing:.1f}° {compass\_direction})")

# Use standard compass heading for logging

compass\_heading = self.get\_compass\_direction(self.heading)

# Get or create a run ID

if not hasattr(self, 'run\_id'):

# First time initialization

self.run\_id = self.get\_next\_run\_id()

# Calculate bearing angle (standard compass bearing)

bearing = (90 - self.heading) % 360 # Convert to standard compass bearing

log\_data = {

'timestamp': datetime.now().isoformat(),

'run\_id': self.run\_id,

'x': self.x,

'y': self.y,

'heading': self.heading,

'bearing': bearing, # Add standard bearing

'compass\_heading': compass\_heading,

'fix\_quality': '3D Fix', # Simulated fix quality

'satellite\_count': random.randint(8, 12), # Simulated satellite count

'deviation': (self.navigator.calculate\_deviation(self.x, self.y)

if hasattr(self, 'navigator') and self.navigator else 0)

}

log\_file\_path = r'F:\GPS\task\_2\_waypoints\rover\_log.csv'

with open(log\_file\_path, 'a', newline='') as csvfile:

# Add 'bearing' to fieldnames

fieldnames = ['timestamp', 'run\_id', 'x', 'y', 'heading', 'bearing', 'compass\_heading',

'fix\_quality', 'satellite\_count', 'deviation']

writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

if not os.path.exists(log\_file\_path) or os.path.getsize(log\_file\_path) == 0:

writer.writeheader()

writer.writerow(log\_data)

return True

def get\_compass\_direction(self, heading):

"""

Convert rover's heading (0° = East, 90° = North) to standard compass direction.

Standard compass: North = 0°, East = 90°, South = 180°, West = 270°

"""

# Convert from rover heading to standard compass bearing

compass\_heading = (90 - heading) % 360

# Define standard compass bearings based on converted heading

if 348.75 <= compass\_heading or compass\_heading < 11.25:

return "North"

elif 11.25 <= compass\_heading < 33.75:

return "North-Northeast"

elif 33.75 <= compass\_heading < 56.25:

return "Northeast"

elif 56.25 <= compass\_heading < 78.75:

return "East-Northeast"

elif 78.75 <= compass\_heading < 101.25:

return "East"

elif 101.25 <= compass\_heading < 123.75:

return "East-Southeast"

elif 123.75 <= compass\_heading < 146.25:

return "Southeast"

elif 146.25 <= compass\_heading < 168.75:

return "South-Southeast"

elif 168.75 <= compass\_heading < 191.25:

return "South"

elif 191.25 <= compass\_heading < 213.75:

return "South-Southwest"

elif 213.75 <= compass\_heading < 236.25:

return "Southwest"

elif 236.25 <= compass\_heading < 258.75:

return "West-Southwest"

elif 258.75 <= compass\_heading < 281.25:

return "West"

elif 281.25 <= compass\_heading < 303.75:

return "West-Northwest"

elif 303.75 <= compass\_heading < 326.25:

return "Northwest"

elif 326.25 <= compass\_heading < 348.75:

return "North-Northwest"

else:

return "Unknown" # Should never reach here

def get\_next\_run\_id(self):

"""Determine the next run ID based on existing data in the log file."""

log\_file\_path = r'F:\GPS\task\_2\_waypoints\rover\_log.csv'

# If file doesn't exist, start with run 1

if not os.path.exists(log\_file\_path):

return 1

try:

# Read the existing file to find the highest run\_id

max\_run\_id = 0

with open(log\_file\_path, 'r', newline='') as csvfile:

reader = csv.DictReader(csvfile)

for row in reader:

if 'run\_id' in row:

try:

run\_id = int(row['run\_id'])

max\_run\_id = max(max\_run\_id, run\_id)

except (ValueError, TypeError):

pass

# Return the next run ID

return max\_run\_id + 1

except Exception as e:

print(f"Error determining run ID: {e}")

return 1 # Default to 1 if there's an error

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)

if self.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, cannot move forward.")

return False

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(0.001) # 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):

rover\_scale = 3.5 # Scale factor to make rover appear larger

if rover\_patch is None:

base\_verts = np.array([[0.7, 0], [-0.3, 0.4], [-0.3, -0.4]])

scaled\_verts = base\_verts \* rover\_scale # Apply scaling

rover\_patch = Polygon(scaled\_verts, closed=True, fc='blue', ec='black') # Correct: uses scaled vertices

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(0.001)

return rover\_patch

# Moved from farm\_entry.py

def visualize\_turn(rover, target\_heading, ax, fig, rover\_patch=None, rotation\_speed\_factor=4):

"""

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

Includes logging for turn diagnostics.

"""

if rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, cannot turn.")

return rover\_patch

# Normalize headings to [0, 360)

current = rover.heading % 360

target\_heading = target\_heading % 360

# Calculate the absolute angle difference (shortest path)

clockwise\_diff = (target\_heading - current) % 360

counterclockwise\_diff = (current - target\_heading) % 360

# Choose the smallest rotation direction

# FIXED: Swapped direction labels to match standard compass directions

if clockwise\_diff <= counterclockwise\_diff:

angle\_diff = clockwise\_diff

direction = "turn\_left" # Changed from "turn\_right" because clockwise is left in standard compass

else:

angle\_diff = -counterclockwise\_diff

direction = "turn\_right" # Changed from "turn\_left" because counterclockwise is right in standard compass

# Log turn details for debugging

print(f"🔄 TURN: From {current:.1f}° to {target\_heading:.1f}°, Direction: {direction}, Angle: {abs(angle\_diff):.1f}°")

# Skip small turns (threshold 5 degrees)

if abs(angle\_diff) < 5:

rover.heading = target\_heading

rover.last\_heading = target\_heading

print(f"✓ Skipped small turn (<5°), set heading to {target\_heading:.1f}°")

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

# Log the turning command

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

# Calculate steps for smooth animation

steps = max(3, min(int(abs(angle\_diff) / 10), 18))

step\_ang = angle\_diff / steps

# Calculate pause duration

pause\_duration = min(0.01, 0.005 \* (180 / max(1, abs(angle\_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)

# Ensure exact final heading

rover.heading = target\_heading % 360

rover.last\_heading = rover.heading

return update\_rover\_visualization(rover, ax, fig, 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 = 2.0

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 load\_waypoints\_from\_csv(filename):

"""

Load waypoints from a CSV file

Returns a list of (x,y) tuples representing the waypoints path

"""

if not os.path.exists(filename):

print(f"⚠️ Waypoints file not found: {filename}")

return []

waypoints = []

try:

with open(filename, 'r') as csvfile:

reader = csv.DictReader(csvfile)

for row in reader:

x = float(row['x'])

y = float(row['y'])

waypoints.append((x, y))

print(f"✅ Loaded {len(waypoints)} waypoints from {filename}")

return waypoints

except Exception as e:

print(f"❌ Error loading waypoints: {e}")

return []

def load\_rows\_from\_csv(self, csv\_filename):

"""

Load waypoints from CSV file (with columns x, y, row\_index)

and process them into rows\_data and interpolated\_path.

"""

if not os.path.exists(csv\_filename):

print(f"⚠️ Waypoints file not found: {csv\_filename}")

return False

raw\_points = [] # will hold tuples (row\_idx, x, y)

xs, ys = [], []

# 1) Read CSV and collect raw points plus min/max for normalization

with open(csv\_filename, 'r') as csvfile:

reader = csv.DictReader(csvfile)

for row in reader:

idx = int(row['row\_index'])

x = float(row['x'])

y = float(row['y'])

raw\_points.append((idx, x, y))

xs.append(x)

ys.append(y)

if not raw\_points:

print("⚠️ No data in CSV.")

return False

# 2) Compute normalization offsets

min\_x, min\_y = min(xs), min(ys)

# 3) Build the full interpolated\_path (normalized)

self.interpolated\_path = [

(x - min\_x, y - min\_y)

for (\_, x, y) in raw\_points

]

# 4) Group by row\_index in original order

grouped = OrderedDict()

for idx, x, y in raw\_points:

grouped.setdefault(idx, []).append((x - min\_x, y - min\_y))

# 5) Build rows\_data

self.rows\_data = []

for idx, pts in grouped.items():

start\_pt = pts[0]

end\_pt = pts[-1]

direction = "↑ UP" if end\_pt[1] > start\_pt[1] else "↓ DOWN"

self.rows\_data.append({

'index': idx,

'x\_pos': start\_pt[0],

'direction': direction,

'start': start\_pt,

'end': end\_pt

})

# 6) Print a clean plan

print("\n📋 CSV-based Navigation Plan:")

print(f" Total waypoints: {len(self.interpolated\_path)}")

print(f" Estimated rows: {len(self.rows\_data)}")

for row in self.rows\_data:

print(f" Row {row['index']+1}: "

f"X-position {row['x\_pos']:.2f}m, "

f"Direction {row['direction']}")

return True

# Add this method to your RowNavigator class

def calculate\_deviation(self, x, y):

"""Calculate how far the rover is from the current path segment."""

if not self.interpolated\_path or len(self.interpolated\_path) < 2:

return 0

# Find current path segment (between current and next waypoint)

current\_index = min(self.current\_waypoint\_index, len(self.interpolated\_path) - 2)

next\_index = current\_index + 1

if current\_index < 0 or next\_index >= len(self.interpolated\_path):

return 0

p1 = self.interpolated\_path[current\_index]

p2 = self.interpolated\_path[next\_index]

# Calculate perpendicular distance to line segment

# Line equation: Ax + By + C = 0

A = p2[1] - p1[1]

B = p1[0] - p2[0]

C = p2[0]\*p1[1] - p1[0]\*p2[1]

# Distance formula: |Ax + By + C| / sqrt(A² + B²)

distance = abs(A\*x + B\*y + C) / math.sqrt(A\*A + B\*B) if (A\*A + B\*B) > 0 else 0

return distance

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 with more intermediate points for smoother movement

if going\_up:

# More points for smoother movement (was 5, now more)

y\_points = np.linspace(bottom\_y, top\_y, 8)

row\_start = (x, bottom\_y)

row\_end = (x, top\_y)

else:

y\_points = np.linspace(top\_y, bottom\_y, 8)

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 improved 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

# Add curved transition points between rows instead of just one point

# This creates a smoother path for the rover to follow

num\_transition\_points = 5 # More points for smoother curve

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

t = j / num\_transition\_points

# Create a slight curve for the transition

trans\_x = x + t \* (next\_x - x)

trans\_y = transition\_y

self.interpolated\_path.append((trans\_x, trans\_y))

# Print row information

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):

"""

Calculate heading between two points with tolerance for small differences.

Modified to handle position rounding precision.

"""

dx = p2[0] - p1[0]

dy = p2[1] - p1[1]

tolerance = 0.01 # Matches position rounding to 2 decimal places

if abs(dx) < tolerance:

print(f"📍 Vertical movement detected: dy={dy:.4f}, setting heading to {'90°' if dy > 0 else '270°'}")

return 90.0 if dy > 0 else 270.0

elif abs(dy) < tolerance:

print(f"📍 Horizontal movement detected: dx={dx:.4f}, setting heading to {'0°' if dx > 0 else '180°'}")

return 0.0 if dx > 0 else 180.0

else:

heading = math.degrees(math.atan2(dy, dx)) % 360

print(f"📍 Diagonal movement: dx={dx:.4f}, dy={dy:.4f}, heading={heading:.1f}°")

return heading

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):

"""

Perform a smooth turn to the target heading.

"""

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

if abs(heading\_diff) < 10:

self.rover.heading = target\_heading

self.rover.last\_heading = target\_heading

if ax and fig and rover\_patch:

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

print(f"✓ Skipped small turn (<10°), set heading to {target\_heading:.1f}°")

return rover\_patch

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

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

steps = max(5, int(abs(heading\_diff) / 30))

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(0.001)

self.rover.heading = round(target\_heading, 1)

self.rover.last\_heading = self.rover.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):

"""

Move rover to target point with precision at constant speed.

Includes logging for turn diagnostics.

"""

max\_attempts = 40

attempts = 0

last\_time = time.time() # Initialize time tracking

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

while attempts < max\_attempts:

if self.rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping movement.")

return False

current\_time = time.time()

time\_elapsed = current\_time - last\_time

last\_time = current\_time

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")

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 difference is significant

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

if abs(heading\_diff) > 10:

print(f"🧭 Initiating turn: Current {self.rover.heading:.1f}°, Desired {desired\_heading:.1f}°, Diff {heading\_diff:.1f}°")

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

else:

print(f"✓ Heading diff {abs(heading\_diff):.1f}° < 10°, no turn needed")

# Calculate move\_dist based on constant speed and elapsed time

move\_dist = self.movement\_speed \* time\_elapsed

if move\_dist > dist\_to\_target:

move\_dist = dist\_to\_target # Prevent overshooting

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(0.001)

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):

"""

Align rover to the next waypoint.

"""

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)

current\_heading = self.rover.heading

heading\_diff = self.heading\_difference(current\_heading, desired\_heading)

if abs(heading\_diff) < 10:

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:

rover\_patch = visualize\_turn(self.rover, desired\_heading, ax, fig, rover\_patch, rotation\_speed\_factor=6)

else:

self.rover.heading = desired\_heading

self.rover.last\_heading = desired\_heading

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

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 with improved turning and reporting.

"""

# 1) Precompute total points

total\_pts = len(self.interpolated\_path)

print(f"🚜 COMMAND: Navigate zigzag path with {total\_pts} points")

if total\_pts == 0:

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

return False

if self.current\_waypoint\_index >= total\_pts:

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

return True

# 2) Compute a row‐matching tolerance halfway between your actual row y‐positions

# Rows are differentiated by their y-coordinates in zigzag pattern

ys = sorted({pt[1] for pt in self.interpolated\_path})

unique\_ys = []

# Filter out very close y values (within 0.1)

for y in ys:

if not unique\_ys or abs(y - unique\_ys[-1]) > 0.1:

unique\_ys.append(y)

if len(unique\_ys) > 1:

tol = min(abs(unique\_ys[i+1] - unique\_ys[i]) for i in range(len(unique\_ys)-1)) / 2

else:

tol = self.column\_spacing / 2

# 3) Helper to find which row a point belongs to - using y-coordinate and REVERSE indexing

def find\_row(point):

# Match point to row based on y-coordinate with REVERSED indexing

for idx, row\_y in enumerate(unique\_ys):

if abs(point[1] - row\_y) <= tol:

# Return reversed index (if we have n rows, index should be n-idx-1)

return len(unique\_ys) - idx - 1

return len(unique\_ys) - 1 # Default to last row if no match

# 4) Log planned moves once, not in the loop

current\_pos = self.interpolated\_path[self.current\_waypoint\_index]

current\_row\_idx = find\_row(current\_pos)

print(f"\n🌾 Starting in Row {current\_row\_idx+1}")

# Set initial row direction

if current\_row\_idx < len(self.rows\_data):

current\_direction = self.rows\_data[current\_row\_idx]['direction']

else:

# Infer direction from path if row data is incomplete

next\_pt = self.interpolated\_path[self.current\_waypoint\_index + 1] if self.current\_waypoint\_index + 1 < total\_pts else None

if next\_pt:

if next\_pt[0] > current\_pos[0]:

current\_direction = "→ East"

elif next\_pt[0] < current\_pos[0]:

current\_direction = "← West"

else:

current\_direction = "Unknown"

else:

current\_direction = "Unknown"

print(f"🧭 Current direction: {current\_direction}")

success = True

last\_command\_point = None

last\_target\_point = None # Track the last target point to avoid duplicate commands

# 6) Main loop

while self.current\_waypoint\_index < total\_pts - 1:

if self.rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping navigation.")

return False

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

next\_pt = self.interpolated\_path[next\_idx]

current\_pt = self.interpolated\_path[self.current\_waypoint\_index]

# Only print movement command if it's a new target point (avoid duplicates)

if next\_pt != last\_target\_point:

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

f"to ({next\_pt[0]:.2f}, {next\_pt[1]:.2f})")

last\_target\_point = next\_pt

last\_command\_point = current\_pt

# Detect if row changes by comparing y-coordinates

new\_row\_idx = find\_row(next\_pt)

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

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

# Get or infer new direction

if new\_row\_idx < len(self.rows\_data):

new\_direction = self.rows\_data[new\_row\_idx]['direction']

else:

# Infer direction from next points in path

future\_idx = next\_idx + 1

if future\_idx < total\_pts:

future\_pt = self.interpolated\_path[future\_idx]

if future\_pt[0] > next\_pt[0]:

new\_direction = "→ East"

elif future\_pt[0] < next\_pt[0]:

new\_direction = "← West"

else:

new\_direction = "Unknown"

else:

new\_direction = "Unknown"

print(f"🧭 New row direction: {new\_direction}")

# Plan turn for the entire row

if new\_row\_idx < len(self.rows\_data):

start, end = self.rows\_data[new\_row\_idx]['start'], self.rows\_data[new\_row\_idx]['end']

if self.rows\_data[new\_row\_idx]['direction'].startswith("↑") or self.rows\_data[new\_row\_idx]['direction'].startswith("→"):

optimal = self.calculate\_heading(start, end)

else:

optimal = self.calculate\_heading(end, start)

print(f"🧭 Planning efficient turn from {self.rover.heading:.1f}° to {optimal:.1f}°")

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

print("⚠️ Failed to align for transition")

success = False

break

current\_row\_idx = new\_row\_idx

current\_direction = new\_direction

else:

# Only print "continuing in row" message when we're staying in the same row

# and moving to a new target point (not repeatedly)

if next\_pt != last\_target\_point:

print(f"\n🌾 Continuing in Row {current\_row\_idx+1} moving {current\_direction}")

# Check if we need to adjust heading within the row

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

if abs(self.heading\_difference(self.rover.heading, desired)) > 3:

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

print("⚠️ Failed to align in‐row")

success = False

break

# Drive to the point

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

self.current\_waypoint\_index = next\_idx

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

else:

print(f"⚠️ Failed to reach waypoint {next\_idx+1}")

success = False

break

if ax and fig:

plt.pause(0.05)

# 7) Final summary if we made it

if success and self.current\_waypoint\_index == total\_pts - 1:

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

final\_row = find\_row(self.interpolated\_path[-1])

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

# ensure exact final positioning

fx, fy = self.interpolated\_path[-1]

if self.rover.distance\_to(fx, fy) > self.waypoint\_threshold:

print(f"📍 Final adjustment to ({fx:.2f}, {fy:.2f})...")

self.rover.set\_position(fx, fy, force=True)

if ax and fig and rover\_patch:

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

plt.pause(0.5)

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 normalize\_coordinates(waypoints):

"""

Transform coordinates from large values (like UTM) to simulation coordinate space

"""

if not waypoints:

return []

# Find min values to use as origin

min\_x = min(point[0] for point in waypoints)

min\_y = min(point[1] for point in waypoints)

# Normalize all points relative to this origin

normalized = []

for x, y in waypoints:

normalized.append((x - min\_x, y - min\_y))

print(f"✓ Normalized coordinates from ({min\_x:.1f}, {min\_y:.1f}) origin")

return normalized

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:

if rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping navigation.")

return False, rover\_patch

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 = 2.4 # Increased step size for faster movement (was 0.2)

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

ANIMATION\_SPEED = 0.001 # Faster animation (was 0.01)

ROTATION\_STEP\_FACTOR = 8 # 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):

if rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping path following.")

return False, rover\_patch

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)

rover.navigator = row\_navigator

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

import logging\_100mm

# 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

from sleep\_mode import FailsafeModule

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():

def on\_failsafe\_triggered(reason):

print(f"⚠️ Failsafe triggered: {reason.value}")

rover.log\_movement("stop") # Stop the rover for safety

def on\_recovery\_attempt(reason):

print(f"🔄 Attempting recovery from {reason.value}")

return True # Assume recovery succeeds for simulation

print("🚜 Farm Rover Navigation Simulation 🚜")

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

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

# Create the rover

rover = Rover()

gps\_logger = logging\_100mm.initialize\_gps\_logger(rover)

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

# Create row navigator first to load waypoints

navigator = RowNavigator(rover)

failsafe = FailsafeModule()

failsafe.update\_gps\_status(has\_fix=True, satellites=10, hdop=1.0)

failsafe.update\_internet\_status(connected=True, latency=0.1)

failsafe.update\_module\_communication()

rover.failsafe = failsafe

rover.navigator = navigator

failsafe.set\_callbacks(on\_failsafe\_triggered, on\_recovery\_attempt)

failsafe.start\_monitoring()

navigator.zigzag\_pattern = True

# Load waypoints from CSV file

csv\_loaded = navigator.load\_rows\_from\_csv(r"F:\GPS\task\_2\_waypoints\waypoints\_100mm.csv")

if not csv\_loaded:

print("❌ Failed to load waypoints from CSV. Simulation cannot proceed without waypoints.")

return

# Calculate farm boundaries based on waypoints with margin

margin = 3.0 # Add margin around waypoints

min\_x = min(point[0] for point in navigator.interpolated\_path) - margin

max\_x = max(point[0] for point in navigator.interpolated\_path) + margin

min\_y = min(point[1] for point in navigator.interpolated\_path) - margin

max\_y = max(point[1] for point in navigator.interpolated\_path) + margin

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

# Create vertices for the farm boundary

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

print(f"📏 Dynamic farm boundaries: X [{min\_x:.2f}, {max\_x:.2f}], Y [{min\_y:.2f}, {max\_y:.2f}]")

# 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

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

# Generate rows within the farm using zigzag pattern

# Load waypoints from CSV file

csv\_loaded = navigator.load\_rows\_from\_csv(r"F:\GPS\task\_2\_waypoints\waypoints\_100mm.csv")

if navigator.interpolated\_path:

wp\_min\_x = min(point[0] for point in navigator.interpolated\_path)

wp\_max\_x = max(point[0] for point in navigator.interpolated\_path)

wp\_min\_y = min(point[1] for point in navigator.interpolated\_path)

wp\_max\_y = max(point[1] for point in navigator.interpolated\_path)

# Use the wider range between farm boundaries and waypoints

plot\_min\_x = min(min\_x, wp\_min\_x)

plot\_max\_x = max(max\_x, wp\_max\_x)

plot\_min\_y = min(min\_y, wp\_min\_y)

plot\_max\_y = max(max\_y, wp\_max\_y)

# Add a larger margin

margin = max(plot\_max\_x - plot\_min\_x, plot\_max\_y - plot\_min\_y) \* 0.15

ax.set\_xlim(plot\_min\_x - margin, plot\_max\_x + margin)

ax.set\_ylim(plot\_min\_y - margin, plot\_max\_y + margin)

else:

# Fallback to original farm boundaries

margin = 3

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

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

ax.grid(True)

if not csv\_loaded:

print("❌ Failed to load waypoints from CSV. Simulation cannot proceed without waypoints.")

return

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")

def on\_rover\_wakeup():

print("Rover has woken up! Resuming operations...")

# Do whatever you need when rover wakes up

# 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 using direct point-to-point moves

with a larger step size and a slightly more generous tolerance to avoid getting stuck.

"""

print("\n🗺️ Navigating directly to starting point...")

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

rover,

path\_start[0],

path\_start[1],

ax,

fig,

rover\_patch,

step\_size=1.5, # larger increments per move

tolerance=0.8 # accept slightly further from the exact point

)

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)

logging\_100mm.stop\_gps\_logger(rover)

failsafe.stop\_monitoring()

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}")

SLEEP MODE

import time

import threading

import datetime

import logging

from enum import Enum

import math

# Set up logging

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler("rover\_failsafe.log", encoding='utf-8'),

logging.StreamHandler()

]

)

logger = logging.getLogger("RoverFailsafe")

class FailsafeReason(Enum):

"""Enumeration of possible reasons for entering failsafe mode"""

GPS\_DATA\_LOSS = "GPS data loss"

GPS\_STALE\_DATA = "GPS stale data"

INTERNET\_CONNECTION\_LOST = "Internet connection lost"

INTERNET\_CONNECTION\_SLOW = "Internet connection slow"

OVER\_TEMPERATURE = "Over temperature"

MODULE\_COMMUNICATION\_FAILURE = "Module communication failure"

SIGNAL\_INSTABILITY = "Signal instability"

CUSTOM = "Custom reason"

class FailsafeModule:

"""

Module to handle failsafe and sleep mode triggering for rover operations.

This module is separate from the farm\_safety module and handles system-level

safety rather than navigation-specific safety.

"""

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

# Status flags

self.in\_failsafe\_mode = False

self.failsafe\_reason = None

self.recovery\_attempts = 0

self.recovery\_in\_progress = False

# Timestamp tracking

self.last\_gps\_update = None

self.last\_internet\_check = None

self.last\_module\_comm = None

self.last\_recovery\_attempt = None

# Thresholds and settings

self.gps\_stale\_threshold = 5.0 # seconds

self.internet\_timeout = 10.0 # seconds

self.recovery\_interval = 60.0 # seconds between recovery attempts

self.max\_recovery\_attempts = 5 # maximum number of recovery attempts

self.temperature\_threshold = 60.0 # degrees Celsius

# Signal tracking

self.signal\_loss\_events = []

self.signal\_loss\_window = 60.0 # 1 minute window for signal loss events

self.signal\_loss\_threshold = 5 # Number of signal losses to trigger failsafe

# Initialize monitoring threads

self.monitoring\_active = False

self.monitor\_thread = None

# Callback for failsafe activation

self.failsafe\_callback = None

self.recovery\_callback = None

logger.info("Failsafe module initialized")

def set\_callbacks(self, failsafe\_callback, recovery\_callback=None):

"""

Set callbacks for failsafe activation and recovery

Args:

failsafe\_callback: Function to call when failsafe is triggered

Should accept FailsafeReason as an argument

recovery\_callback: Function to call when attempting recovery

Should accept FailsafeReason as an argument

"""

self.failsafe\_callback = failsafe\_callback

self.recovery\_callback = recovery\_callback

logger.info("Failsafe callbacks set")

def configure(self, gps\_stale\_threshold=None, internet\_timeout=None,

recovery\_interval=None, max\_recovery\_attempts=None,

temperature\_threshold=None, signal\_loss\_threshold=None):

"""Configure failsafe thresholds and settings"""

if gps\_stale\_threshold is not None:

self.gps\_stale\_threshold = gps\_stale\_threshold

if internet\_timeout is not None:

self.internet\_timeout = internet\_timeout

if recovery\_interval is not None:

self.recovery\_interval = recovery\_interval

if max\_recovery\_attempts is not None:

self.max\_recovery\_attempts = max\_recovery\_attempts

if temperature\_threshold is not None:

self.temperature\_threshold = temperature\_threshold

if signal\_loss\_threshold is not None:

self.signal\_loss\_threshold = signal\_loss\_threshold

logger.info(f"Failsafe configured: GPS stale threshold={self.gps\_stale\_threshold}s, "

f"Internet timeout={self.internet\_timeout}s, "

f"Recovery interval={self.recovery\_interval}s, "

f"Max recovery attempts={self.max\_recovery\_attempts}, "

f"Temperature threshold={self.temperature\_threshold}°C, "

f"Signal loss threshold={self.signal\_loss\_threshold} events")

def start\_monitoring(self):

"""Start the failsafe monitoring thread"""

if self.monitoring\_active:

logger.warning("Monitoring already active")

return

self.monitoring\_active = True

self.monitor\_thread = threading.Thread(target=self.\_monitor\_loop, daemon=True)

self.monitor\_thread.start()

logger.info("Failsafe monitoring started")

def stop\_monitoring(self):

"""Stop the failsafe monitoring thread"""

if not self.monitoring\_active:

logger.warning("Monitoring not active")

return

self.monitoring\_active = False

if self.monitor\_thread:

self.monitor\_thread.join(timeout=1.0)

logger.info("Failsafe monitoring stopped")

def \_monitor\_loop(self):

"""Main monitoring loop that checks for failsafe conditions"""

while self.monitoring\_active:

try:

# Check various failsafe conditions

self.\_check\_gps\_status()

self.\_check\_internet\_connection()

self.\_check\_module\_communication()

self.\_check\_temperature()

self.\_check\_signal\_stability()

# Check if recovery is needed and if it's time for another attempt

if (self.in\_failsafe\_mode and not self.recovery\_in\_progress and

(self.last\_recovery\_attempt is None or

time.time() - self.last\_recovery\_attempt >= self.recovery\_interval)):

self.\_attempt\_recovery()

time.sleep(1.0) # Check conditions every second

except Exception as e:

logger.error(f"Error in failsafe monitor loop: {e}")

def \_check\_gps\_status(self):

"""Check if GPS data is missing or stale"""

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any GPS data

if self.last\_gps\_update is None:

self.\_trigger\_failsafe(FailsafeReason.GPS\_DATA\_LOSS)

return

# Check if GPS data is stale

if current\_time - self.last\_gps\_update > self.gps\_stale\_threshold:

self.\_trigger\_failsafe(FailsafeReason.GPS\_STALE\_DATA)

def \_check\_internet\_connection(self):

"""Check if internet connection for RTCM corrections is lost or slow"""

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any internet connectivity data

if self.last\_internet\_check is None:

return

# Check if internet connection is lost or too slow

if current\_time - self.last\_internet\_check > self.internet\_timeout:

self.\_trigger\_failsafe(FailsafeReason.INTERNET\_CONNECTION\_LOST)

def \_check\_module\_communication(self):

"""Check if module-to-computer communication is working"""

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any module communication data

if self.last\_module\_comm is None:

return

# Module communication timeout (5 seconds)

if current\_time - self.last\_module\_comm > 5.0:

self.\_trigger\_failsafe(FailsafeReason.MODULE\_COMMUNICATION\_FAILURE)

def \_check\_temperature(self):

"""Check if system temperature is too high"""

if self.in\_failsafe\_mode:

return

# This is a placeholder for actual temperature checking

# In a real system, you would read from temperature sensors

temperature = self.\_get\_system\_temperature()

if temperature > self.temperature\_threshold:

self.\_trigger\_failsafe(FailsafeReason.OVER\_TEMPERATURE)

def \_check\_signal\_stability(self):

"""Check for signal instability (≥ 5 signal-loss events per minute)"""

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Remove signal loss events older than the window

self.signal\_loss\_events = [t for t in self.signal\_loss\_events

if current\_time - t <= self.signal\_loss\_window]

# Check if we have too many signal loss events

if len(self.signal\_loss\_events) >= self.signal\_loss\_threshold:

self.\_trigger\_failsafe(FailsafeReason.SIGNAL\_INSTABILITY)

def \_trigger\_failsafe(self, reason):

"""Trigger failsafe mode and execute callback"""

if self.in\_failsafe\_mode:

return

self.in\_failsafe\_mode = True

self.failsafe\_reason = reason

self.recovery\_attempts = 0

logger.warning(f"⚠️ FAILSAFE MODE ACTIVATED: {reason.value}")

# Execute failsafe callback if provided

if self.failsafe\_callback:

try:

self.failsafe\_callback(reason)

except Exception as e:

logger.error(f"Error in failsafe callback: {e}")

def \_attempt\_recovery(self):

"""Attempt to recover from failsafe mode"""

if not self.in\_failsafe\_mode or self.recovery\_in\_progress:

return

self.recovery\_in\_progress = True

self.recovery\_attempts += 1

self.last\_recovery\_attempt = time.time()

logger.info(f"🔄 Attempting recovery #{self.recovery\_attempts} for {self.failsafe\_reason.value}")

# Execute recovery callback if provided

if self.recovery\_callback:

try:

success = self.recovery\_callback(self.failsafe\_reason)

if success:

self.\_clear\_failsafe()

except Exception as e:

logger.error(f"Error in recovery callback: {e}")

# If too many attempts, give up and stay in failsafe mode

if self.recovery\_attempts >= self.max\_recovery\_attempts:

logger.error(f"❌ Max recovery attempts reached for {self.failsafe\_reason.value}")

# Keep in failsafe mode but allow future recovery attempts

self.recovery\_in\_progress = False

def \_clear\_failsafe(self):

"""Clear failsafe mode after successful recovery"""

if not self.in\_failsafe\_mode:

return

old\_reason = self.failsafe\_reason

self.in\_failsafe\_mode = False

self.failsafe\_reason = None

self.recovery\_attempts = 0

logger.info(f"✅ Recovered from failsafe mode: {old\_reason.value}")

# API Methods for rover interaction

def update\_gps\_status(self, has\_fix=True, satellites=0, hdop=0.0):

"""

Update GPS status information

Args:

has\_fix: Whether GPS has a valid fix

satellites: Number of satellites in view

hdop: Horizontal dilution of precision

"""

if not has\_fix or satellites < 4 or hdop > 5.0:

# Register as signal loss event

self.signal\_loss\_events.append(time.time())

else:

self.last\_gps\_update = time.time()

def update\_internet\_status(self, connected=True, latency=0.0):

"""

Update internet connection status

Args:

connected: Whether internet is connected

latency: Latency in seconds

"""

self.last\_internet\_check = time.time()

# Consider high latency as a slow connection

if not connected or latency > 2.0:

if self.monitoring\_active and not self.in\_failsafe\_mode:

self.\_trigger\_failsafe(

FailsafeReason.INTERNET\_CONNECTION\_LOST if not connected

else FailsafeReason.INTERNET\_CONNECTION\_SLOW

)

def update\_module\_communication(self, timestamp=None):

"""

Update module communication timestamp

Args:

timestamp: Optional timestamp to use, defaults to current time

"""

self.last\_module\_comm = timestamp if timestamp is not None else time.time()

def report\_signal\_loss(self):

"""Report a signal loss event"""

self.signal\_loss\_events.append(time.time())

def trigger\_custom\_failsafe(self, reason\_text):

"""

Manually trigger failsafe mode with a custom reason

Args:

reason\_text: Description of the custom reason

"""

custom\_reason = FailsafeReason.CUSTOM

self.\_trigger\_failsafe(custom\_reason)

logger.warning(f"Custom failsafe reason: {reason\_text}")

def force\_recovery(self):

"""Force a recovery attempt regardless of interval"""

if self.in\_failsafe\_mode:

logger.info("Forcing recovery attempt")

self.\_attempt\_recovery()

def get\_status(self):

"""Get current failsafe status information"""

return {

"in\_failsafe\_mode": self.in\_failsafe\_mode,

"failsafe\_reason": self.failsafe\_reason.value if self.failsafe\_reason else None,

"recovery\_attempts": self.recovery\_attempts,

"recovery\_in\_progress": self.recovery\_in\_progress,

"last\_gps\_update": self.last\_gps\_update,

"last\_internet\_check": self.last\_internet\_check,

"last\_module\_comm": self.last\_module\_comm,

"signal\_loss\_events": len(self.signal\_loss\_events),

"monitoring\_active": self.monitoring\_active

}

def \_get\_system\_temperature(self):

"""

Get the current system temperature

This is a placeholder implementation that should be replaced

with actual hardware temperature readings in a real system

"""

# In a real system, you would read from temperature sensors

# For simulation, we'll return a fixed value below threshold

return 45.0 # Simulated temperature in Celsius