FARM ENTRY:

import math

import time

import matplotlib.pyplot as plt

import numpy as np

from matplotlib.patches import Polygon

import matplotlib.transforms as transforms

from farm\_safety import SafetyModule

safety = SafetyModule()

# --- Constants ---

STEP = 0.4 # Increased for faster navigation

TOLERANCE = 0.8 # Increased tolerance for reaching targets

MAX\_ATTEMPTS = 200 # Increased maximum attempts

DEBUG = False

ANIMATION\_SPEED = 0.05 # Rotation animation speed (lower is faster)

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.")

class Rover:

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

self.x = 0.0

self.y = 0.0

self.heading = 0.0

self.history = []

self.command\_count = 0

self.waypoint = None

self.geofence = None

self.inside\_fence = False

self.entry\_point = None

self.blocked\_directions = set()

self.stuck\_count = 0 # Counter for detecting when rover is stuck

self.position\_history = [] # To detect cycles and stuck conditions

self.rover\_patch = None # Visual representation of the rover

self.fence\_locked = False # Track if rover is locked within the fence

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

"""Set rover position with optional heading"""

if self.geofence and not force:

# 1) Check if we're at (or within tolerance of) the entry point

if self.entry\_point and self.is\_entry\_point(x, y):

# Entering through entry point: allow and lock inside

self.inside\_fence = True

self.fence\_locked = True

print(f"🔓 Entering farm at entry point — rover locked inside.")

else:

# 2) Perform standard polygon check

in\_fence = self.is\_point\_in\_polygon(x, y, self.geofence)

# 3) Prevent exit once locked inside

if self.fence\_locked and self.inside\_fence and not in\_fence:

print(f"🔒 Movement blocked: Rover is locked inside the farm")

return False

# 4) Block any crossing unless it is the first entry

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

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

return False

# Apply new position/heading

self.x = x

self.y = y

if heading is not None:

self.heading = heading % 360

if add\_to\_history:

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

return True

def set\_waypoint(self, x, y):

"""Set target waypoint"""

self.waypoint = (x, y)

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

"""Set geofence polygon and entry point"""

self.geofence = vertices

self.entry\_point = entry\_point

self.inside\_fence = self.is\_point\_in\_polygon(self.x, self.y, vertices)

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

"""Move rover forward in current heading direction"""

if distance <= 0:

print("⚠️ Invalid distance value <= 0")

return False

rad = math.radians(self.heading)

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

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

# Smooth intermediate animation

if ax and fig and rover\_patch and distance > STEP:

steps = min(int(distance / (STEP/2)), 5)

if steps > 1:

step\_x = (target\_x - self.x) / steps

step\_y = (target\_y - self.y) / steps

for \_ in range(steps-1):

nx, ny = self.x + step\_x, self.y + step\_y

if not self.set\_position(nx, ny, add\_to\_history=False):

return False

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

plt.pause(ANIMATION\_SPEED/2)

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

if success:

self.command\_count += 1

print(f"Cmd #{self.command\_count}: MOVE\_FWD {distance:.2f}m → ({self.x:.3f}, {self.y:.3f})")

if self.waypoint:

print(f" 📏 Distance to waypoint: {self.distance\_to(\*self.waypoint):.2f}m")

if self.entry\_point and not self.inside\_fence:

print(f" 📏 Distance to entry point: {self.distance\_to(\*self.entry\_point):.2f}m")

return True

else:

rounded = int(self.heading/10)\*10

self.blocked\_directions.add(rounded)

print(f"⚠️ Movement in direction {rounded}° blocked")

return False

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 distance\_to(self, tx, ty):

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

def is\_point\_in\_polygon(self, x, y, verts):

n = len(verts)

inside = False

p1x, p1y = verts[0]

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

p2x, p2y = verts[i%n]

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

if p1y!=p2y:

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

if p1x==p2x or x<=xin:

inside = not inside

p1x,p1y = p2x,p2y

return inside

def is\_entry\_point(self, x, y, tolerance=TOLERANCE):

if self.entry\_point:

return self.distance\_to(\*self.entry\_point) <= tolerance

return False

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

self.position\_history.append((self.x,self.y))

if len(self.position\_history)>10:

self.position\_history.pop(0)

if len(self.position\_history)>=5:

maxd = max(math.hypot(self.x-px,self.y-py) for px,py in self.position\_history)

if maxd < TOLERANCE/2:

self.stuck\_count += 1

print(f"⚠️ Possible stuck condition detected ({self.stuck\_count}/3)")

if self.stuck\_count>=3:

print("🔄 Taking recovery action - making a significant turn")

self.blocked\_directions.clear()

self.stuck\_count=0

return (self.heading+120)%360

else:

self.stuck\_count=0

return None

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

"""

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

PATH\_STEP = 0.2 # Use smaller steps for greater precision

PATH\_TOLERANCE = 0.05 # Much smaller tolerance to enforce strict adherence

# Skip the first waypoint if it's the current position

start\_idx = 1 if rover.distance\_to(\*waypoints[0]) < TOLERANCE/3 else 0

# For path visualization

actual\_path = []

path\_line = None

# Enforce initial position to be exactly on path

if start\_idx > 0:

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

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

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

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

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

# Before starting segment, ensure we're exactly at the start point

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

rover.set\_position(current\_wp[0], current\_wp[1], force=True)

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

# Interpolate points along the segment for precise movement

interp\_points = []

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

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

t = j / num\_interp

interp\_points.append((

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

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

))

# Strictly follow each interpolated point

for j, point in enumerate(interp\_points):

# Always turn precisely toward the point

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

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

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

# Calculate exact distance to move

move\_dist = rover.distance\_to(\*point)

# Move directly to the point (never using deviation-based corrections)

if move\_dist > PATH\_TOLERANCE:

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

# If movement fails, try directly setting position as last resort

if not success:

print(f"⚠️ Direct movement failed - force setting position to stay on path")

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

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(0.01)

# Safety check

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

if status != 'safe':

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

return False, rover\_patch

# For the final waypoint, use exact positioning

last\_wp = waypoints[-1]

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

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

# Move directly to last waypoint

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

if final\_dist > PATH\_TOLERANCE:

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

if not success:

print("⚠️ Failed to reach final waypoint via direct movement")

# Force position to complete the path

rover.set\_position(last\_wp[0], last\_wp[1], force=True)

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

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

return True, rover\_patch

# Visualization helpers (unchanged)

def safe\_remove(element):

if element:

try:

element.remove()

return True

except:

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

return False

def create\_rover\_patch():

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

return Polygon(verts, closed=True, fc='blue', ec='black')

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

if rover\_patch is None:

rover\_patch = create\_rover\_patch()

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

def visualize\_turn(rover, new\_heading, ax, fig, rover\_patch=None):

current = rover.heading

diff = (new\_heading-current+180)%360 - 180

if abs(diff)<5:

rover.heading = new\_heading

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

rover.command\_count += 1

print(f"Cmd #{rover.command\_count}: ROTATE\_TO {new\_heading:.1f}° ({diff:.1f}° turn)")

if rover.waypoint:

print(f" 📏 Distance to waypoint: {rover.distance\_to(\*rover.waypoint):.2f}m")

steps = max(5, min(int(abs(diff)/5), 36))

step\_ang = diff/steps

try:

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(ANIMATION\_SPEED)

except Exception as e:

if DEBUG: print(f"Turn viz error: {e}")

rover.heading = new\_heading

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

# Path planning (unchanged)

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

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

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 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 main():

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

rover = Rover()

print("🔧 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: ")

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

# Allow user to manually input entry point on farm border

print("\n🚪 Enter farm entry point coordinates (must be on farm border):")

print(f" Valid X range: {min\_x} to {max\_x}")

print(f" Valid Y range: {min\_y} to {max\_y}")

print(" The point must be exactly on one of the farm boundary lines.")

while True:

entry\_x = get\_float(" Entry point X: ")

entry\_y = get\_float(" Entry point Y: ")

# Check if point is on the border of the farm

on\_border = False

# Check if point is on horizontal borders

if (min\_x <= entry\_x <= max\_x) and (entry\_y == min\_y or entry\_y == max\_y):

on\_border = True

# Check if point is on vertical borders

elif (min\_y <= entry\_y <= max\_y) and (entry\_x == min\_x or entry\_x == max\_x):

on\_border = True

if on\_border:

print(f"✅ Valid entry point: ({entry\_x:.2f}, {entry\_y:.2f})")

break

else:

print("⚠️ Entry point must be exactly on the farm border. Please try again.")

entry = (entry\_x, entry\_y)

rover.set\_geofence(verts, entry)

print("🔧 Enter starting position:")

while True:

x1=get\_float(" x1: "); y1=get\_float(" y1: ")

if rover.is\_point\_in\_polygon(x1,y1,verts):

print("⚠️ Starting point must be outside the farm. Please enter new coordinates.")

else: break

print(f"✅ Valid starting position: ({x1:.2f}, {y1:.2f})")

print("🔧 Enter waypoint coords:")

while True:

wx=get\_float(" wx: "); wy=get\_float(" wy: ")

if not rover.is\_point\_in\_polygon(wx,wy,verts):

print("⚠️ Waypoint must be inside the farm. Please enter new coordinates.")

else: break

print(f"✅ Valid waypoint: ({wx:.2f}, {wy:.2f})")

try:

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

xs=[v[0] for v in verts]+[x1,wx,entry[0]]; ys=[v[1] for v in verts]+[y1,wy,entry[1]]

mxx, Mxx = min(xs)-2, max(xs)+2; myy, Myy = min(ys)-2, max(ys)+2

ax.set\_xlim(mxx,Mxx); ax.set\_ylim(myy,Myy); ax.grid(True)

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

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

ax.add\_patch(fence)

# Place entry point marker on top (zorder=10)

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

ax.scatter(x1,y1,c='green',s=80,label='Start (Outside)')

ax.scatter(wx,wy,c='red',s=120,marker='\*',label='Waypoint')

pl,=ax.plot([],[], 'b-',alpha=0.5,label='Path'); ax.path\_line=pl; ax.legend(loc='upper left')

fig.canvas.draw\_idle(); plt.pause(0.5)

except Exception as e:

print(f"Error in plot setup: {e}")

fig, ax = plt.subplots(figsize=(8,6)); pl,=ax.plot([]); ax.path\_line=pl; ax.set\_title("Rover Navigation (Limited View)")

rover.set\_position(x1,y1,force=True,add\_to\_history=False)

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

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

print("\n🚜 Moving rover from outside farm to entry point...\n")

print(f"📏 Initial distance to entry point: {rover.distance\_to(\*entry):.2f}m")

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

try:

main()

except KeyboardInterrupt:

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

except Exception as e:

print(f"\n❌ Simulation error: {e}")

if DEBUG:

import traceback; traceback.print\_exc()

ROW NAVIGATION

import matplotlib.pyplot as plt

import numpy as np

from scipy.interpolate import CubicSpline

import math

import time

from farm\_safety import SafetyModule

safety = SafetyModule()

class RowNavigator:

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

self.rover = rover

self.current\_waypoint\_index = 0

self.interpolated\_path = [] # Path of discrete waypoints

self.waypoint\_threshold = 0.1 # Keep the precise threshold

self.column\_spacing = 1.5

self.column\_height = 15

self.zigzag\_pattern = True # This only affects waypoint generation pattern

self.movement\_speed = 0.10 # Increased from 0.05 to 0.2 for faster movement

def identify\_current\_row(self):

"""Task 2.ii: Identify which row the closest point belongs to"""

if not self.interpolated\_path or self.current\_waypoint\_index < 0:

print("⚠️ No path or closest point identified yet")

return None

# Calculate which row this belongs to based on X position

closest\_point = self.interpolated\_path[self.current\_waypoint\_index]

row\_index = int((closest\_point[0] - self.interpolated\_path[0][0]) / self.column\_spacing)

print(f"\n📋 Task Current point belongs to Row #{row\_index + 1}")

print(f" Row X-position: {closest\_point[0]:.3f}")

# Determine if we're closer to the top or bottom of the row

if row\_index % 2 == 0:

# Even rows go bottom to top

position = "bottom" if closest\_point[1] < (self.interpolated\_path[0][1] + self.column\_height/2) else "top"

else:

# Odd rows go top to bottom

position = "top" if closest\_point[1] > (self.interpolated\_path[0][1] + self.column\_height/2) else "bottom"

print(f" Position in row: Near {position}")

print(f" ✓ Defining Row #{row\_index + 1} as the first row task")

self.first\_row = row\_index

return row\_index

def determine\_next\_row\_direction(self):

"""Task Determine direction to the next row"""

current\_row = self.identify\_current\_row()

if current\_row is None:

return None

# In our zigzag pattern:

# - If at even row and near top → go right

# - If at odd row and near bottom → go right

# - Otherwise we're in the middle of a row, so we continue in the current direction

closest\_point = self.interpolated\_path[self.current\_waypoint\_index]

y\_threshold = self.interpolated\_path[0][1] + self.column\_height \* 0.9 # Near top

y\_bottom\_threshold = self.interpolated\_path[0][1] + self.column\_height \* 0.1 # Near bottom

if (current\_row % 2 == 0 and closest\_point[1] > y\_threshold) or \

(current\_row % 2 == 1 and closest\_point[1] < y\_bottom\_threshold):

next\_row\_direction = "right"

next\_row = current\_row + 1

elif current\_row % 2 == 0:

next\_row\_direction = "up"

next\_row = current\_row

else:

next\_row\_direction = "down"

next\_row = current\_row

print(f"\n📋 Task Next row direction: {next\_row\_direction}")

if next\_row != current\_row:

print(f" Next row will be Row #{next\_row + 1}")

return next\_row\_direction

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 = []

for i in range(num\_strips):

x = start\_x + i \* spacing

# Generate waypoints for the vertical segment

if i % 2 == 0:

# Even columns: bottom to top

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

else:

# Odd columns: top to bottom

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

# Add all waypoints for this column

for y in y\_points:

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

# Add horizontal transition to the next column (except for the last column)

if i < num\_strips - 1:

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

transition\_y = top\_y if i % 2 == 0 else bottom\_y

# Use fewer intermediate points for horizontal transitions

intermediate\_points = 3 # Reduced from 5 for faster movement

for j in range(1, intermediate\_points):

interp\_x = x + (next\_x - x) \* j / intermediate\_points

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

# Add the transition point explicitly

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

return self.interpolated\_path

def distance(self, p1, p2):

"""Calculate Euclidean distance between two points"""

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

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

"""Calculate heading angle between two points (in degrees)"""

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

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

"""Calculate the difference between current and target heading"""

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

return diff

def find\_closest\_waypoint(self):

"""Find the closest waypoint in the path"""

if not self.interpolated\_path:

print("⚠️ No valid path")

return None

min\_dist = float('inf')

closest\_idx = 0

for i, point in enumerate(self.interpolated\_path):

dist = self.distance((self.rover.x, self.rover.y), point)

if dist < min\_dist:

min\_dist = dist

closest\_idx = i

self.current\_waypoint\_index = closest\_idx

print(f"🔍 Found closest waypoint: #{closest\_idx}")

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

print(f" Waypoint position: ({self.interpolated\_path[closest\_idx][0]:.3f}, {self.interpolated\_path[closest\_idx][1]:.3f})")

print(f" Distance: {min\_dist:.3f}m")

return self.interpolated\_path[closest\_idx]

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

"""Align rover heading to the direction of the next waypoint"""

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

print("⚠️ No active path or at end of path")

return False

# Get current position and next waypoint

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

next\_pt = self.interpolated\_path[self.current\_waypoint\_index + 1]

# Calculate desired heading to next waypoint

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

if ax and fig and rover\_patch:

from farm\_entry import visualize\_turn

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

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

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

else:

self.rover.heading = desired\_heading

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

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

return True

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

"""Move precisely to a specific point using larger, faster steps"""

from farm\_entry import update\_rover\_visualization, visualize\_turn

max\_attempts = 100 # Reduced from 200 for faster execution

attempts = 0

step\_size = self.movement\_speed # Larger step size for faster movement

while attempts < max\_attempts:

# Calculate distance to target

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

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

# Check if we've reached the target

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

# Force exact position to ensure precision

self.rover.x = target\_point[0]

self.rover.y = target\_point[1]

if ax and fig and rover\_patch:

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

plt.pause(0.01) # Faster visualization update

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

return True

# Align heading precisely to target

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

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

if abs(heading\_diff) > 1: # Keep the strict heading alignment

if ax and fig and rover\_patch:

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

else:

self.rover.heading = desired\_heading

# Move a larger step toward the target

move\_dist = min(step\_size, dist\_to\_target)

# Calculate the target position based on heading and move distance

target\_x = self.rover.x + move\_dist \* math.cos(math.radians(self.rover.heading))

target\_y = self.rover.y + move\_dist \* math.sin(math.radians(self.rover.heading))

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

# Check safety before moving

from farm\_safety import safety # Make sure this is imported at the top of the file

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

if status == 'safe':

if ax and fig and rover\_patch:

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

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

plt.pause(0.01) # Faster updates for quicker animation

else:

success = self.rover.move\_forward(move\_dist)

elif status == 'drift':

# Handle drift scenario

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

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

# Update rover position and heading

self.rover.x, self.rover.y = pos

self.rover.heading = heading

success = True

if ax and fig and rover\_patch:

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

plt.pause(0.01)

print(f"🔄 Drift corrected: New heading {heading:.1f}°")

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

# Handle no-go zone or boundary violation

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

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

# Update rover position and heading

self.rover.x, self.rover.y = pos

self.rover.heading = heading

success = False # Mark as unsuccessful to trigger alternative path finding

# Update visualization if available

if ax and fig and rover\_patch:

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

plt.pause(0.01)

print(f"⚠️ Safety violation handled: {violation\_status}")

if not success:

print("⚠️ Movement blocked, attempting to adjust")

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

# Try a very small sidestep and continue

sidestep\_angle = (desired\_heading + 90) % 360

if ax and fig and rover\_patch:

rover\_patch = visualize\_turn(self.rover, sidestep\_angle, ax, fig, rover\_patch)

self.rover.move\_forward(0.1, ax, fig, rover\_patch)

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

else:

self.rover.heading = sidestep\_angle

self.rover.move\_forward(0.1)

self.rover.heading = desired\_heading

attempts += 1

# Only print status updates every 5 steps to reduce console output

if attempts % 5 == 0:

print(f" Moving: distance remaining = {dist\_to\_target:.3f}m")

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

print(f"⚠️ Failed to reach point after {max\_attempts} attempts")

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

return False

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

"""Navigate directly to the starting point (first waypoint) of the path"""

if not self.interpolated\_path:

print("⚠️ No path generated")

return False

# Get the starting point (first waypoint)

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

print(f"\n🚜 Navigating to starting point: ({starting\_point[0]:.3f}, {starting\_point[1]:.3f})")

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

print(f" Distance: {self.distance((self.rover.x, self.rover.y), starting\_point):.3f}m")

# Move precisely to the starting point

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

if result:

# Update current waypoint index to the starting point

self.current\_waypoint\_index = 0

print(f"✅ Successfully reached the starting point")

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

else:

print(f"⚠️ Failed to reach starting point")

return result

def determine\_next\_task(self):

"""Task 2.i: Determine the next task (next waypoint in the row)"""

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

print("⚠️ No more waypoints available")

return None

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

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

print(f"\n📋 Determined next task - waypoint #{next\_idx}")

print(f" Next waypoint: ({next\_point[0]:.3f}, {next\_point[1]:.3f})")

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

print(f" Distance: {self.distance((self.rover.x, self.rover.y), next\_point):.3f}m")

return next\_point

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

""" Align rover direction to the orientation of next task"""

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

if not next\_point:

return False

# Get current position

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

# Calculate desired heading to next waypoint

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

print(f"\n🔄 Aligning rover to next task orientation")

if ax and fig and rover\_patch:

from farm\_entry import visualize\_turn

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

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

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

else:

self.rover.heading = desired\_heading

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

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

return True

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

"""Navigate through the path with the updated task structure"""

print(f"\n🚜 Starting navigation of path...\n")

if not self.interpolated\_path:

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

return False

# Task 1: Navigate directly to the starting point of the path

print("\n📋 Task 1: Navigate directly to starting point of path")

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

print("⚠️ Failed to reach the starting point. Aborting.")

return False

# Now at starting point, proceed with next tasks

# Task 2.i: Determine next task

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

if not next\_point:

print("⚠️ No next task available. Ending navigation.")

return False

# Task 2.ii: Align to next task orientation

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

print("⚠️ Failed to align to next task. Attempting to continue anyway.")

# Task 3: Navigate through the identified row

print("\n📋 Task 3: Start navigating in the identified row")

return self.navigate\_path(ax, fig, rover\_patch)

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

"""Task 3: Navigate through the path, strictly following all waypoints"""

from farm\_entry import update\_rover\_visualization, visualize\_turn

import matplotlib.pyplot as plt

print(f"\n🚜 Task Navigating path with {len(self.interpolated\_path) - self.current\_waypoint\_index - 1} remaining waypoints...\n")

current\_row = self.identify\_current\_row()

print(f"\n🚜 Starting navigation of Row #{current\_row + 1} (first task)\n")

# Start from current waypoint and visit all remaining waypoints

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

# Task 3.ii: Read the current location

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

print(f"\n📍 Task Current location: ({self.rover.x:.3f}, {self.rover.y:.3f})")

# Task 3.iii: Find the closest next waypoint in the row

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

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

dist\_to\_next = self.distance(current\_pos, next\_point)

print(f"🎯 Task Found next waypoint #{next\_idx}: ({next\_point[0]:.3f}, {next\_point[1]:.3f})")

print(f" Distance to next waypoint: {dist\_to\_next:.3f}m")

# Use the precise movement method to reach the exact waypoint

reached = self.move\_precisely\_to\_point(next\_point, ax, fig, rover\_patch)

if reached:

self.current\_waypoint\_index = next\_idx

print(f"✅ Reached waypoint #{next\_idx} exactly")

new\_row = self.identify\_current\_row()

if new\_row != current\_row:

current\_row = new\_row

next\_direction = self.determine\_next\_row\_direction()

print(f"\n🚜 Now navigating Row #{current\_row + 1}")

print(f" Next direction: {next\_direction}")

else:

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

# Even if we can't reach it perfectly, we still advance to the next waypoint

self.current\_waypoint\_index = next\_idx

print(f"✅ Reached end of path")

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

return True

FARM SIMULATION

import matplotlib.pyplot as plt

import numpy as np

from astar\_algo import AStarPlanner

# Import our modules

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

from row\_navigation import RowNavigator

from farm\_safety import SafetyModule

safety = SafetyModule()

# Set farm boundary

# Add any no-go zones

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 run\_simulation():

print("🚜 Farm Rover Navigation Simulation 🚜")

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

# Create the rover

rover = Rover()

# Setup the geofence (farm boundaries)

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)]

# Allow user to manually input entry point on farm border

print("\n🚪 Enter farm entry point coordinates (must be on farm border):")

print(f" Valid X range: {min\_x} to {max\_x}")

print(f" Valid Y range: {min\_y} to {max\_y}")

print(" The point must be exactly on one of the farm boundary lines.")

while True:

entry\_x = get\_float(" Entry point X: ")

entry\_y = get\_float(" Entry point Y: ")

# Check if point is on the border of the farm

on\_border = False

# Check if point is on horizontal borders

if (min\_x <= entry\_x <= max\_x) and (entry\_y == min\_y or entry\_y == max\_y):

on\_border = True

# Check if point is on vertical borders

elif (min\_y <= entry\_y <= max\_y) and (entry\_x == min\_x or entry\_x == max\_x):

on\_border = True

if on\_border:

print(f"✅ Valid entry point: ({entry\_x:.2f}, {entry\_y:.2f})")

break

else:

print("⚠️ Entry point must be exactly on the farm border. Please try again.")

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

# Set geofence in rover

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

# Set farm boundary in safety module

safety.set\_geofence(verts)

# Rest of the code continues as before...

# Modified A\* integration code that properly handles visualization scopes

# Place this in the run\_simulation function after the farm boundary setup

# and before the "Navigate to entry point" section

def integrate\_astar\_planner(rover, safety, entry\_point, ax, fig, rover\_patch):

"""

Integrates the A\* planner into the simulation with proper visualization scope

Returns: reached\_entry (bool), updated\_rover\_patch

"""

# Create and configure the A\* planner with appropriate parameters

print("\n🗺️ Setting up A\* path planner for navigating to entry point...")

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

print(f"🔍 Grid dimensions: {planner.ny}x{planner.nx} cells")

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

# Optional: Visualize the grid for debugging (in a separate figure)

try:

grid\_fig, grid\_ax = plt.subplots(figsize=(8, 6))

planner.visualize\_grid(grid\_ax)

grid\_fig.canvas.draw\_idle()

plt.pause(0.5)

except Exception as e:

print(f"Note: Grid visualization skipped: {e}")

# Plan the path

print("\n🔄 Computing optimal path to entry point...")

waypoints = planner.plan()

if waypoints:

print(f"✅ A\* found a path with {len(waypoints)} waypoints:")

for i, (wx, wy) in enumerate(waypoints):

if i == 0:

print(f" Start: ({wx:.2f}, {wy:.2f})")

elif i == len(waypoints) - 1:

print(f" Goal: ({wx:.2f}, {wy:.2f})")

elif i < 5 or i > len(waypoints) - 5:

print(f" Waypoint {i}: ({wx:.2f}, {wy:.2f})")

elif i == 5 and len(waypoints) > 10:

print(f" ... {len(waypoints) - 10} more waypoints ...")

# Visualize the planned path on the main figure

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

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

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(0.5)

print("\n🚜 Following A\* path to entry point...")

# Use our new precise path follower instead of waypoint-by-waypoint navigation

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

# Update visualization

fig.canvas.draw\_idle()

plt.pause(0.5)

if not reached:

print(f"⚠️ Failed to follow planned path. Attempting to replan from current position...")

# Reinitialize planner with current position

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

new\_waypoints = planner.plan()

if new\_waypoints and len(new\_waypoints) > 1:

print(f"✅ Found new path with {len(new\_waypoints)} waypoints")

# Update path visualization

safe\_remove(planned\_path\_line)

path\_x, path\_y = zip(\*new\_waypoints)

ax.plot(path\_x, path\_y, 'g--', linewidth=2, alpha=0.7, label='Replanned Path')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(0.5)

# Try again with new path

reached, rover\_patch = follow\_path\_precisely(rover, new\_waypoints, ax, fig, rover\_patch)

# Check if we reached the entry point

if rover.distance\_to(\*entry\_point) <= TOLERANCE:

print("\n🎯 Successfully reached entry point using A\* navigation!")

return True, rover\_patch

else:

print(f"\n⚠️ A\* navigation terminated {rover.distance\_to(\*entry\_point):.2f}m from entry point.")

print("🔄 Attempting direct navigation for final approach...")

# Final direct approach to entry point

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

rover, entry\_point[0], entry\_point[1], ax, fig, rover\_patch)

return reached\_entry, rover\_patch

else:

print("⚠️ A\* failed to find a path—falling back to direct navigate\_to\_point.")

# Use the original direct navigation as fallback

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

rover, entry\_point[0], entry\_point[1], ax, fig, rover\_patch)

return reached\_entry, rover\_patch

# Add no-go zones if needed

# For example, to 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)

# Get rover starting position (outside farm)

print("\n🔧 Enter starting position (must be outside the farm):")

while True:

x1 = get\_float(" x1: ")

y1 = get\_float(" y1: ")

if rover.is\_point\_in\_polygon(x1, y1, verts):

print("⚠️ Starting point must be outside the farm. Please enter new coordinates.")

else:

break

print(f"✅ Valid starting position: ({x1:.3f}, {y1:.3f})")

# Initialize visualization

plt.ion()

fig, ax = plt.subplots(figsize=(12, 10))

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)

# Mark entry point and start position

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

ax.scatter(x1, y1, c='green', s=80, label='Start (Outside)')

# Setup plot limits and grid

xs = [v[0] for v in verts] + [x1, entry\_point[0]]

ys = [v[1] for v in verts] + [y1, entry\_point[1]]

margin = 3 # Add more margin

mxx, Mxx = min(xs) - margin, max(xs) + margin

myy, Myy = min(ys) - margin, max(ys) + margin

ax.set\_xlim(mxx, Mxx)

ax.set\_ylim(myy, Myy)

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

rover.set\_position(x1, y1, force=True, add\_to\_history=False)

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

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

# --- TASK 1: Enter the farm at entry point ---

print("\n🚜 TASK 1: Moving rover from outside farm to entry point...\n")

print(f"📏 Initial distance to entry point: {rover.distance\_to(\*entry\_point):.3f}m")

reached\_entry, rover\_patch = integrate\_astar\_planner(rover, safety, entry\_point, ax, fig, rover\_patch)

if not reached\_entry:

print("\n⚠️ Could not reach farm entry point after multiple attempts.")

print(" Try adjusting simulation parameters or entry point location.")

return

# Force rover position to exactly match entry point

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

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

# Mark entry point reached

ax.scatter(entry\_point[0], entry\_point[1], c='cyan', s=80, marker='^', label='Entry Reached')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

print("\n✅ TASK 1 COMPLETE: Successfully entered the farm")

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

# Force rover position to exactly match entry point

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

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

# Mark entry point reached

# Mark entry point reached

ax.scatter(entry\_point[0], entry\_point[1], c='cyan', s=80, marker='^', label='Entry Reached')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

print("\n✅ TASK 1 COMPLETE: Successfully entered the farm")

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

print("\n🚜 TASK 2: 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

)

# 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 2.i: Navigate directly to the path start point ---

print("\n🚜 TASK 2.i: 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)

ax.plot(path\_x, path\_y, 'y--', linewidth=2, alpha=0.7, label='Path to Start')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(0.5)

# Navigate to path start

reached\_start = False

for i, (wx, wy) in enumerate(waypoints[1:], 1):

print(f"\n📍 Moving to waypoint {i}/{len(waypoints)-1}: ({wx:.2f}, {wy:.2f})")

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

fig.canvas.draw\_idle()

plt.pause(0.2)

if not reached and i < len(waypoints) - 1:

print(f"⚠️ Failed to reach waypoint {i}. Attempting to continue.")

# Final approach to exact path start

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 2.i COMPLETE: Successfully reached path start point")

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

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

print("\n🚜 TASK 2.ii: 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.ii 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

# 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}")

if 'DEBUG' in globals() and DEBUG:

import traceback

traceback.print\_exc()

A STAR 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)

FARM SAFETY  
  
import math

import random

import numpy as np

class SafetyModule:

"""

A module to handle drift and no-go zone safety checks for farm rover navigation.

Combines logic from both drift recovery and geofence enforcement.

"""

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

# 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

# No-go zone configuration

self.no\_go\_zones = [] # list of no-go zones [(x0, y0, x1, y1), ...]

self.geofence = None # farm boundary polygon vertices

self.violations\_history = [] # track violation incidents

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

def set\_geofence(self, vertices):

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

self.geofence = vertices

def add\_no\_go\_zone(self, x0, y0, x1, y1):

"""Add a rectangular no-go zone"""

self.no\_go\_zones.append((x0, y0, x1, y1))

def clear\_no\_go\_zones(self):

"""Clear all no-go zones"""

self.no\_go\_zones = []

def is\_in\_no\_go\_zone(self, pos):

"""Check if position is in any no-go zone"""

for zone in self.no\_go\_zones:

x0, y0, x1, y1 = zone

if x0 <= pos[0] <= x1 and y0 <= pos[1] <= y1:

return True

return False

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

"""Check if position is outside the geofence"""

if not self.geofence:

return False

# Ray casting algorithm for point-in-polygon test

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):

if 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 current position and heading are safe.

Returns:

- safety\_status: 'safe', 'drift', 'no-go', or 'outside'

- recovery\_data: Data needed for recovery (if applicable)

"""

# Check for no-go zone

if self.is\_in\_no\_go\_zone(pos):

self.violations\_history.append(('no-go', pos.copy(), heading))

return 'no-go', {'violation\_type': 'no-go', 'pos': pos.copy(), 'heading': heading}

# Check for outside geofence

if self.is\_outside\_geofence(pos):

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

return 'outside', {'violation\_type': 'outside', 'pos': pos.copy(), 'heading': heading}

# Check for potential drift (random occurrence)

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

# Find closest point on path for reference

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

# Only drift if we have enough path ahead

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

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

# Calculate drift parameters

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

lookahead = self.lookahead\_steps

# Calculate recovery parameters

trigger\_idx = closest\_idx

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

# Drift direction calculation

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 != 0 else (1.0, 0.0)

# Rotate drift direction

rad = math.radians(drift\_angle)

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

rx = ux \* c + uy \* s

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

# Calculate recovery index

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

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

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

recovery\_target = path[recovery\_idx]

# Return drift data

drift\_data = {

'trigger\_idx': trigger\_idx,

'drift\_angle': drift\_angle,

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

'recovery\_idx': recovery\_idx,

'recovery\_target': recovery\_target,

'path': path

}

return 'drift', drift\_data

# If all checks pass, we're safe

return 'safe', None

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

"""

Handle a drift event by simulating drift and computing recovery path.

Returns:

- updated\_pos: Position after drift or during recovery

- updated\_heading: Heading after drift or during recovery

- status: 'drifting', 'recovering', or 'recovered'

- recovery\_path: Path to follow for recovery

"""

# Extract drift parameters

drift\_vector = drift\_data['drift\_vector']

recovery\_target = drift\_data['recovery\_target']

drift\_len = 1.0

drift\_steps = 20

# Calculate how far to drift in each step

drift\_delta = (drift\_vector[0] \* drift\_len/drift\_steps,

drift\_vector[1] \* drift\_len/drift\_steps)

# Apply drift to position

pos[0] += drift\_delta[0]

pos[1] += drift\_delta[1]

# Determine if we need to start recovery

drift\_steps\_done = drift\_data.get('drift\_steps\_done', 0) + 1

drift\_data['drift\_steps\_done'] = drift\_steps\_done

if drift\_steps\_done >= drift\_steps:

# Calculate recovery heading

desired = self.head(pos, recovery\_target)

dh = self.diff\_h(heading, desired)

# Apply heading correction

turn\_step = self.turn\_rate\_per\_cm \* self.distance\_per\_step

if abs(dh) > 5:

heading += turn\_step if dh > 0 else -turn\_step

return pos, heading, 'recovering', drift\_data

else:

# Move toward recovery point

pos[0] += self.distance\_per\_step \* math.cos(math.radians(heading))

pos[1] += self.distance\_per\_step \* math.sin(math.radians(heading))

# Check if we've recovered

d = self.dist(pos, recovery\_target)

if d < 0.5:

return recovery\_target, heading, 'recovered', None

return pos, heading, 'recovering', drift\_data

else:

return pos, heading, 'drifting', drift\_data

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

"""

Handle a no-go zone violation by backing away from the zone.

Returns:

- updated\_pos: Position after backing away

- updated\_heading: Heading after backing away

- status: 'backing', 'redirecting', or 'recovered'

"""

# Back up in the opposite direction of current heading

backup\_heading = (heading + 180) % 360

# Back up by a small amount

backup\_dist = self.distance\_per\_step \* 2 # Back up by twice the normal step size

new\_pos = [

pos[0] + backup\_dist \* math.cos(math.radians(backup\_heading)),

pos[1] + backup\_dist \* math.sin(math.radians(backup\_heading))

]

# Check if we're still in a no-go zone

if self.is\_in\_no\_go\_zone(new\_pos) or self.is\_outside\_geofence(new\_pos):

# Try a different direction if backing up doesn't work

for angle\_offset in [45, -45, 90, -90]:

try\_heading = (backup\_heading + angle\_offset) % 360

try\_pos = [

pos[0] + backup\_dist \* math.cos(math.radians(try\_heading)),

pos[1] + backup\_dist \* math.sin(math.radians(try\_heading))

]

if not self.is\_in\_no\_go\_zone(try\_pos) and not self.is\_outside\_geofence(try\_pos):

return try\_pos, try\_heading, 'redirecting'

# If all attempts fail, just back up a tiny bit and hope for the best

return [

pos[0] + 0.1 \* math.cos(math.radians(backup\_heading)),

pos[1] + 0.1 \* math.sin(math.radians(backup\_heading))

], backup\_heading, 'backing'

else:

# Successfully backed out

return new\_pos, backup\_heading, 'recovered'

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

"""Find the closest point on a path to the given position"""

min\_dist = float('inf')

min\_idx = 0

min\_point = None

for i, point in enumerate(path):

d = self.dist(pos, point)

if d < min\_dist:

min\_dist = d

min\_idx = i

min\_point = point

return min\_idx, min\_point

# Utility functions

def dist(self, a, b):

"""Calculate Euclidean distance between points a and b"""

return math.hypot(b[0] - a[0], b[1] - a[1])

def head(self, a, b):

"""Calculate heading from point a to point b in degrees"""

return math.degrees(math.atan2(b[1] - a[1], b[0] - a[0])) % 360

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

"""Calculate the minimal angle difference between headings c and t"""

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

# At the end of your file, add:

safety = SafetyModule()