import numpy as np

import matplotlib.pyplot as plt

from matplotlib.animation import FuncAnimation

from matplotlib.patches import Rectangle, Circle

from matplotlib.lines import Line2D

import csv

from datetime import datetime

import os

import math

# --- Helper Functions ---

SOWN\_SEGMENTS\_LOG = set()

def \_commit\_point\_to\_path(points\_list\_lanes, sow\_flags\_list, new\_lane\_point, sow\_flag\_requested, context=""):

# new\_lane\_point is (lane\_x, lane\_y)

if not points\_list\_lanes:

points\_list\_lanes.append(new\_lane\_point)

return

if points\_list\_lanes[-1] == new\_lane\_point:

return

previous\_lane\_point = points\_list\_lanes[-1]

current\_segment = frozenset({previous\_lane\_point, new\_lane\_point})

actual\_sow\_flag\_for\_this\_segment = False

if sow\_flag\_requested:

if current\_segment not in SOWN\_SEGMENTS\_LOG:

actual\_sow\_flag\_for\_this\_segment = True

SOWN\_SEGMENTS\_LOG.add(current\_segment)

points\_list\_lanes.append(new\_lane\_point)

sow\_flags\_list.append(actual\_sow\_flag\_for\_this\_segment)

def \_add\_headland\_segment\_custom\_exit(current\_lane\_x, current\_lane\_y,

target\_lane\_x, target\_lane\_y,

exit\_point\_lanes,

\_points\_list\_lanes, \_sow\_flags\_list,

segment\_label="",

is\_designated\_unsown\_positioning\_leg=False,

gap\_size=1):

ex\_lane\_x, ey\_lane\_y = exit\_point\_lanes

if (current\_lane\_x, current\_lane\_y) == exit\_point\_lanes:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), False, f"AHLCE {segment\_label} Case 1")

return target\_lane\_x, target\_lane\_y

on\_segment\_path = False

if current\_lane\_x == target\_lane\_x == ex\_lane\_x and min(current\_lane\_y, target\_lane\_y) <= ey\_lane\_y <= max(current\_lane\_y, target\_lane\_y):

on\_segment\_path = True

elif current\_lane\_y == target\_lane\_y == ey\_lane\_y and min(current\_lane\_x, target\_lane\_x) <= ex\_lane\_x <= max(current\_lane\_x, target\_lane\_x):

on\_segment\_path = True

if on\_segment\_path:

# Calculate stop point before exit to leave gap

if current\_lane\_x == target\_lane\_x == ex\_lane\_x: # Vertical movement

if current\_lane\_y < ey\_lane\_y: # Moving upward

sow\_stop\_y = max(current\_lane\_y, ey\_lane\_y - gap\_size)

else: # Moving downward

sow\_stop\_y = min(current\_lane\_y, ey\_lane\_y + gap\_size)

# Sow until stop point

if sow\_stop\_y != current\_lane\_y:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (ex\_lane\_x, sow\_stop\_y), True, f"AHLCE {segment\_label} Case 2 SowToGap")

# Unsown movement to exit

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (ex\_lane\_x, ey\_lane\_y), False, f"AHLCE {segment\_label} Case 2 ToExit")

# Continue unsown to target if needed

if (ex\_lane\_x, ey\_lane\_y) != (target\_lane\_x, target\_lane\_y):

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), False, f"AHLCE {segment\_label} Case 2 PastExit")

elif current\_lane\_y == target\_lane\_y == ey\_lane\_y: # Horizontal movement

if current\_lane\_x < ex\_lane\_x: # Moving rightward

sow\_stop\_x = max(current\_lane\_x, ex\_lane\_x - gap\_size)

else: # Moving leftward

sow\_stop\_x = min(current\_lane\_x, ex\_lane\_x + gap\_size)

# Sow until stop point

if sow\_stop\_x != current\_lane\_x:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (sow\_stop\_x, ey\_lane\_y), True, f"AHLCE {segment\_label} Case 2 SowToGap")

# Unsown movement to exit

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (ex\_lane\_x, ey\_lane\_y), False, f"AHLCE {segment\_label} Case 2 ToExit")

# Continue unsown to target if needed

if (ex\_lane\_x, ey\_lane\_y) != (target\_lane\_x, target\_lane\_y):

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), False, f"AHLCE {segment\_label} Case 2 PastExit")

else:

sow\_request = not is\_designated\_unsown\_positioning\_leg

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), sow\_request, f"AHLCE {segment\_label} Case 3")

return target\_lane\_x, target\_lane\_y

def get\_user\_choice\_corner\_lanes(max\_lx\_idx, max\_ly\_idx):

print(f"\nChoose exit corner (0-indexed lanes: X up to {max\_lx\_idx}, Y up to {max\_ly\_idx}):")

print(f"1. Top-Left Lane (0, {max\_ly\_idx})")

print(f"2. Top-Right Lane ({max\_lx\_idx}, {max\_ly\_idx})")

print(f"3. Bottom-Left Lane (0, 0)")

print(f"4. Bottom-Right Lane ({max\_lx\_idx}, 0)")

while True:

try:

choice = int(input("Enter choice (1-4): "))

if choice == 1: return (0, max\_ly\_idx)

if choice == 2: return (max\_lx\_idx, max\_ly\_idx)

if choice == 3: return (0, 0)

if choice == 4: return (max\_lx\_idx, 0)

except ValueError: print("Invalid input.")

def get\_user\_defined\_exit\_lanes(max\_lx\_idx, max\_ly\_idx):

print(f"\nDefine custom exit lane (X: 0-{max\_lx\_idx}, Y: 0-{max\_ly\_idx}):")

print(f"1. Top boundary (Y lane = {max\_ly\_idx})")

print(f"2. Bottom boundary (Y lane = 0)")

print(f"3. Left boundary (X lane = 0)")

print(f"4. Right boundary (X lane = {max\_lx\_idx})")

while True:

try:

b\_choice = int(input("Choose boundary for exit (1-4): "))

if 1 <= b\_choice <= 4: break

except ValueError: print("Invalid input.")

ex\_l, ey\_l = -1, -1

while True:

try:

if b\_choice == 1: ey\_l = max\_ly\_idx; ex\_l = int(input(f"Enter X-lane (0-{max\_lx\_idx}): ")); assert 0 <= ex\_l <= max\_lx\_idx; break

if b\_choice == 2: ey\_l = 0; ex\_l = int(input(f"Enter X-lane (0-{max\_lx\_idx}): ")); assert 0 <= ex\_l <= max\_lx\_idx; break

if b\_choice == 3: ex\_l = 0; ey\_l = int(input(f"Enter Y-lane (0-{max\_ly\_idx}): ")); assert 0 <= ey\_l <= max\_ly\_idx; break

if b\_choice == 4: ex\_l = max\_lx\_idx; ey\_l = int(input(f"Enter Y-lane (0-{max\_ly\_idx}): ")); assert 0 <= ey\_l <= max\_ly\_idx; break

except (ValueError, AssertionError): print("Invalid lane index.")

return (ex\_l, ey\_l)

def \_commit\_partial\_vertical\_sweep(points\_list, sow\_flags\_list, curr\_x, start\_y, end\_y, gap\_size=1):

"""

Creates a partial vertical sweep with gaps at both ends.

Args:

points\_list: List of lane points

sow\_flags\_list: List of sowing flags

curr\_x: Current X lane position

start\_y: Starting Y lane position

end\_y: Ending Y lane position

gap\_size: Size of gap to leave at each end (in lane units)

"""

if start\_y == end\_y:

return end\_y

# Determine direction and calculate total distance

direction = 1 if end\_y > start\_y else -1

total\_distance = abs(end\_y - start\_y)

# If total distance is too small for meaningful gaps, do full sown movement

if total\_distance <= 2 \* gap\_size:

\_commit\_point\_to\_path(points\_list, sow\_flags\_list, (curr\_x, end\_y), True, "PartialSweep\_FullSown")

return end\_y

# Calculate sowing start and end positions

# Leave gap\_size at the beginning and end

sow\_start\_y = start\_y + (gap\_size \* direction)

sow\_end\_y = end\_y - (gap\_size \* direction)

# Phase 1: Unsown movement to sowing start position (creating initial gap)

if sow\_start\_y != start\_y:

\_commit\_point\_to\_path(points\_list, sow\_flags\_list, (curr\_x, sow\_start\_y), False, "PartialSweep\_Phase1\_Gap")

# Phase 2: Sown movement (main productive sweep)

\_commit\_point\_to\_path(points\_list, sow\_flags\_list, (curr\_x, sow\_end\_y), True, "PartialSweep\_Phase2\_Sown")

# Phase 3: Unsown movement to final position (creating final gap)

if sow\_end\_y != end\_y:

\_commit\_point\_to\_path(points\_list, sow\_flags\_list, (curr\_x, end\_y), False, "PartialSweep\_Phase3\_Gap")

return end\_y

# --- Path Generation (Operates in 0-indexed Lane Numbers) ---

# Updated helper function to properly handle gap\_size parameter

def \_add\_headland\_segment\_custom\_exit\_with\_gaps(current\_lane\_x, current\_lane\_y,

target\_lane\_x, target\_lane\_y,

exit\_point\_lanes,

\_points\_list\_lanes, \_sow\_flags\_list,

segment\_label="",

is\_designated\_unsown\_positioning\_leg=False,

gap\_size=1):

"""

Modified version of \_add\_headland\_segment\_custom\_exit that always considers gap\_size

for consistent behavior with inner sweeps.

"""

ex\_lane\_x, ey\_lane\_y = exit\_point\_lanes

if (current\_lane\_x, current\_lane\_y) == exit\_point\_lanes:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), False, f"AHLCE {segment\_label} Case 1")

return target\_lane\_x, target\_lane\_y

on\_segment\_path = False

if current\_lane\_x == target\_lane\_x == ex\_lane\_x and min(current\_lane\_y, target\_lane\_y) <= ey\_lane\_y <= max(current\_lane\_y, target\_lane\_y):

on\_segment\_path = True

elif current\_lane\_y == target\_lane\_y == ey\_lane\_y and min(current\_lane\_x, target\_lane\_x) <= ex\_lane\_x <= max(current\_lane\_x, target\_lane\_x):

on\_segment\_path = True

if on\_segment\_path:

# Calculate stop point before exit to leave gap

if current\_lane\_x == target\_lane\_x == ex\_lane\_x: # Vertical movement

if current\_lane\_y < ey\_lane\_y: # Moving upward

sow\_stop\_y = max(current\_lane\_y, ey\_lane\_y - gap\_size)

else: # Moving downward

sow\_stop\_y = min(current\_lane\_y, ey\_lane\_y + gap\_size)

# Sow until stop point

if sow\_stop\_y != current\_lane\_y:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (ex\_lane\_x, sow\_stop\_y), True, f"AHLCE {segment\_label} Case 2 SowToGap")

# Unsown movement to exit

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (ex\_lane\_x, ey\_lane\_y), False, f"AHLCE {segment\_label} Case 2 ToExit")

# Continue unsown to target if needed

if (ex\_lane\_x, ey\_lane\_y) != (target\_lane\_x, target\_lane\_y):

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), False, f"AHLCE {segment\_label} Case 2 PastExit")

elif current\_lane\_y == target\_lane\_y == ey\_lane\_y: # Horizontal movement

if current\_lane\_x < ex\_lane\_x: # Moving rightward

sow\_stop\_x = max(current\_lane\_x, ex\_lane\_x - gap\_size)

else: # Moving leftward

sow\_stop\_x = min(current\_lane\_x, ex\_lane\_x + gap\_size)

# Sow until stop point

if sow\_stop\_x != current\_lane\_x:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (sow\_stop\_x, ey\_lane\_y), True, f"AHLCE {segment\_label} Case 2 SowToGap")

# Unsown movement to exit

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (ex\_lane\_x, ey\_lane\_y), False, f"AHLCE {segment\_label} Case 2 ToExit")

# Continue unsown to target if needed

if (ex\_lane\_x, ey\_lane\_y) != (target\_lane\_x, target\_lane\_y):

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), False, f"AHLCE {segment\_label} Case 2 PastExit")

else:

# Exit not on this segment - use partial sweep logic for gap consistency when coming from inner sweeps

if current\_lane\_x == target\_lane\_x and current\_lane\_y != target\_lane\_y:

# This is a vertical movement - apply partial sweep logic for gap consistency

if gap\_size > 0:

# Use partial vertical sweep logic

direction = 1 if target\_lane\_y > current\_lane\_y else -1

total\_distance = abs(target\_lane\_y - current\_lane\_y)

# If total distance is too small for meaningful gaps, do full sown movement

if total\_distance <= 2 \* gap\_size:

sow\_request = not is\_designated\_unsown\_positioning\_leg

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), sow\_request, f"AHLCE {segment\_label} Case 3 FullSown")

else:

# Apply gap logic similar to \_commit\_partial\_vertical\_sweep

sow\_start\_y = current\_lane\_y + (gap\_size \* direction)

sow\_end\_y = target\_lane\_y - (gap\_size \* direction)

# Phase 1: Unsown movement to sowing start position (creating initial gap)

if sow\_start\_y != current\_lane\_y:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (current\_lane\_x, sow\_start\_y), False, f"AHLCE {segment\_label} Case 3 InitialGap")

# Phase 2: Sown movement (main productive sweep)

if not is\_designated\_unsown\_positioning\_leg:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (current\_lane\_x, sow\_end\_y), True, f"AHLCE {segment\_label} Case 3 SownMiddle")

else:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (current\_lane\_x, sow\_end\_y), False, f"AHLCE {segment\_label} Case 3 UnsownMiddle")

# Phase 3: Unsown movement to final position (creating final gap)

if sow\_end\_y != target\_lane\_y:

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), False, f"AHLCE {segment\_label} Case 3 FinalGap")

else:

# No gap\_size specified, use original logic

sow\_request = not is\_designated\_unsown\_positioning\_leg

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), sow\_request, f"AHLCE {segment\_label} Case 3 NoGap")

else:

# Horizontal movement or other cases - use original logic

sow\_request = not is\_designated\_unsown\_positioning\_leg

\_commit\_point\_to\_path(\_points\_list\_lanes, \_sow\_flags\_list, (target\_lane\_x, target\_lane\_y), sow\_request, f"AHLCE {segment\_label} Case 3 Other")

return target\_lane\_x, target\_lane\_y

def generate\_fixed\_path(n\_inner\_x\_sweeps, max\_lane\_idx\_x, max\_lane\_idx\_y, exit\_point\_lanes, is\_corner\_exit, gap\_size=1):

global SOWN\_SEGMENTS\_LOG

SOWN\_SEGMENTS\_LOG.clear()

\_points\_lanes = [] # Stores (lane\_x, lane\_y)

\_sow\_flags = []

ex\_ln\_x, ex\_ln\_y = exit\_point\_lanes # Exit point in lane indices

def \_commit(new\_ln\_pt, sow\_req, ctx=""):

\_commit\_point\_to\_path(\_points\_lanes, \_sow\_flags, new\_ln\_pt, sow\_req, ctx)

# Determine start\_lane\_y for boustrophedon (0 or max\_lane\_idx\_y)

natural\_start\_ln\_y = 0

if ex\_ln\_y == max\_lane\_idx\_y: natural\_start\_ln\_y = max\_lane\_idx\_y

elif ex\_ln\_y == 0: natural\_start\_ln\_y = 0

else:

natural\_start\_ln\_y = max\_lane\_idx\_y if abs(ex\_ln\_y - max\_lane\_idx\_y) < abs(ex\_ln\_y - 0) else 0

b\_start\_lane\_y = natural\_start\_ln\_y

if n\_inner\_x\_sweeps % 2 != 0: # If odd number of inner sweeps, flip start Y

b\_start\_lane\_y = 0 if natural\_start\_ln\_y == max\_lane\_idx\_y else max\_lane\_idx\_y

# Determine start\_lane\_x for boustrophedon (inner lanes are 1 to max\_lane\_idx\_x - 1)

start\_sweep\_lane\_x = -1

lanes\_to\_sweep\_x = [] # List of inner lane X-indices (1 to max\_lane\_idx\_x - 1)

first\_inner\_lane\_x = 1

last\_inner\_lane\_x = max\_lane\_idx\_x - 1

if n\_inner\_x\_sweeps == 0:

# If no inner sweeps, boustrophedon part is skipped.

# Set start\_sweep\_lane\_x to a headland lane for initial positioning.

# This ensures initial\_pos\_lane\_x is valid.

start\_sweep\_lane\_x = 0 if ex\_ln\_x <= max\_lane\_idx\_x / 2.0 else max\_lane\_idx\_x

if is\_corner\_exit:

if n\_inner\_x\_sweeps > 0: # Only determine for actual sweeps

start\_sweep\_lane\_x = last\_inner\_lane\_x if ex\_ln\_x <= max\_lane\_idx\_x / 2.0 else first\_inner\_lane\_x

# If n\_inner\_x\_sweeps is 0, start\_sweep\_lane\_x remains as set above or its default -1 (which then gets corrected for initial\_pos\_lane\_x)

initial\_sweep\_dir = -1 if start\_sweep\_lane\_x == last\_inner\_lane\_x else 1

end\_col\_for\_range = (first\_inner\_lane\_x -1) if initial\_sweep\_dir == -1 else (last\_inner\_lane\_x + 1)

if n\_inner\_x\_sweeps > 0:

lanes\_to\_sweep\_x = list(range(start\_sweep\_lane\_x, end\_col\_for\_range, initial\_sweep\_dir))

else: # Custom Exit

if n\_inner\_x\_sweeps == 0:

start\_sweep\_lane\_x = 0 # Default if no inner sweeps

elif n\_inner\_x\_sweeps == 1:

start\_sweep\_lane\_x = first\_inner\_lane\_x

elif n\_inner\_x\_sweeps > 1:

min\_dist\_to\_ex = float('inf'); best\_start\_lx = -1

for inner\_lx\_candidate in range(first\_inner\_lane\_x, last\_inner\_lane\_x + 1):

dist = abs(inner\_lx\_candidate - ex\_ln\_x)

if dist < min\_dist\_to\_ex:

min\_dist\_to\_ex = dist; best\_start\_lx = inner\_lx\_candidate

start\_sweep\_lane\_x = best\_start\_lx if best\_start\_lx !=-1 else first\_inner\_lane\_x

if n\_inner\_x\_sweeps == 1 and start\_sweep\_lane\_x != -1 : lanes\_to\_sweep\_x = [start\_sweep\_lane\_x]

elif n\_inner\_x\_sweeps > 1:

all\_inner\_lanes\_x = list(range(first\_inner\_lane\_x, last\_inner\_lane\_x + 1))

initial\_sweep\_dir = 1 if start\_sweep\_lane\_x <= (first\_inner\_lane\_x + last\_inner\_lane\_x) / 2.0 else -1

part1 = []; curr\_lx\_sweep = start\_sweep\_lane\_x; visited\_p1 = set()

while first\_inner\_lane\_x <= curr\_lx\_sweep <= last\_inner\_lane\_x and curr\_lx\_sweep not in visited\_p1:

part1.append(curr\_lx\_sweep); visited\_p1.add(curr\_lx\_sweep)

curr\_lx\_sweep += initial\_sweep\_dir

part2 = [lx for lx in all\_inner\_lanes\_x if lx not in visited\_p1]

if part2: part2 = sorted(part2, reverse=(initial\_sweep\_dir == 1))

lanes\_to\_sweep\_x = part1 + part2

initial\_pos\_lane\_x = start\_sweep\_lane\_x if n\_inner\_x\_sweeps > 0 else (0 if ex\_ln\_x <= max\_lane\_idx\_x / 2.0 else max\_lane\_idx\_x)

\_points\_lanes.append((initial\_pos\_lane\_x, b\_start\_lane\_y))

curr\_ln\_x, curr\_ln\_y = initial\_pos\_lane\_x, b\_start\_lane\_y

# MODIFIED: Use partial vertical sweeps for inner lanes

for i, sweep\_ln\_x in enumerate(lanes\_to\_sweep\_x):

curr\_ln\_x = sweep\_ln\_x

target\_ln\_y = 0 if curr\_ln\_y == max\_lane\_idx\_y else max\_lane\_idx\_y

# Use partial sweep instead of full sweep

curr\_ln\_y = \_commit\_partial\_vertical\_sweep(\_points\_lanes, \_sow\_flags, curr\_ln\_x, curr\_ln\_y, target\_ln\_y, gap\_size)

if i < len(lanes\_to\_sweep\_x) - 1:

next\_sweep\_ln\_x = lanes\_to\_sweep\_x[i+1]

\_commit((next\_sweep\_ln\_x, curr\_ln\_y), False, f"InnerSweep H-Turn{i+1}")

curr\_ln\_x = next\_sweep\_ln\_x

target\_headland\_ln\_x = 0

if n\_inner\_x\_sweeps == 0:

target\_headland\_ln\_x = curr\_ln\_x # Already on a headland, stay or move to other based on exit

elif curr\_ln\_x <= (first\_inner\_lane\_x + last\_inner\_lane\_x) / 2.0 :

target\_headland\_ln\_x = 0

else:

target\_headland\_ln\_x = max\_lane\_idx\_x

if curr\_ln\_x != target\_headland\_ln\_x:

\_commit((target\_headland\_ln\_x, curr\_ln\_y), False, "HeadlandPrep")

curr\_ln\_x, curr\_ln\_y = target\_headland\_ln\_x, curr\_ln\_y

target\_ln\_y1 = 0 if curr\_ln\_y == max\_lane\_idx\_y else max\_lane\_idx\_y

# MODIFIED: Apply partial sweep logic to first headland vertical segment if coming from inner sweeps

if n\_inner\_x\_sweeps > 0:

# Use partial sweep for first headland vertical segment to maintain gap consistency

if not is\_corner\_exit:

# For custom exit, use the enhanced custom exit function with gap support

curr\_ln\_x, curr\_ln\_y = \_add\_headland\_segment\_custom\_exit\_with\_gaps(

curr\_ln\_x, curr\_ln\_y, curr\_ln\_x, target\_ln\_y1,

exit\_point\_lanes, \_points\_lanes, \_sow\_flags,

"HLSeg1\_V\_CustomExit\_WithGaps", gap\_size=gap\_size)

else:

# For corner exit, use partial sweep to maintain consistency with inner sweeps

curr\_ln\_y = \_commit\_partial\_vertical\_sweep(\_points\_lanes, \_sow\_flags, curr\_ln\_x, curr\_ln\_y, target\_ln\_y1, gap\_size)

else:

# No inner sweeps, use regular logic

if not is\_corner\_exit:

curr\_ln\_x, curr\_ln\_y = \_add\_headland\_segment\_custom\_exit(curr\_ln\_x, curr\_ln\_y, curr\_ln\_x, target\_ln\_y1, exit\_point\_lanes, \_points\_lanes, \_sow\_flags, "HLSeg1\_V")

else:

\_commit((curr\_ln\_x, target\_ln\_y1), True, "HLSeg1\_V\_Corner"); curr\_ln\_y = target\_ln\_y1

target\_ln\_x1 = max\_lane\_idx\_x if curr\_ln\_x == 0 else 0

if not is\_corner\_exit: curr\_ln\_x, curr\_ln\_y = \_add\_headland\_segment\_custom\_exit(curr\_ln\_x, curr\_ln\_y, target\_ln\_x1, curr\_ln\_y, exit\_point\_lanes, \_points\_lanes, \_sow\_flags, "HLSeg2\_H")

else: \_commit((target\_ln\_x1, curr\_ln\_y), True, "HLSeg2\_H\_Corner"); curr\_ln\_x = target\_ln\_x1

target\_ln\_y2 = 0 if curr\_ln\_y == max\_lane\_idx\_y else max\_lane\_idx\_y

if not is\_corner\_exit: curr\_ln\_x, curr\_ln\_y = \_add\_headland\_segment\_custom\_exit(curr\_ln\_x, curr\_ln\_y, curr\_ln\_x, target\_ln\_y2, exit\_point\_lanes, \_points\_lanes, \_sow\_flags, "HLSeg3\_V")

else: \_commit((curr\_ln\_x, target\_ln\_y2), True, "HLSeg3\_V\_Corner"); curr\_ln\_y = target\_ln\_y2

target\_ln\_x2 = max\_lane\_idx\_x if curr\_ln\_x == 0 else 0

if not is\_corner\_exit: curr\_ln\_x, curr\_ln\_y = \_add\_headland\_segment\_custom\_exit(curr\_ln\_x, curr\_ln\_y, target\_ln\_x2, curr\_ln\_y, exit\_point\_lanes, \_points\_lanes, \_sow\_flags, "HLSeg4\_H", is\_designated\_unsown\_positioning\_leg=True)

else: \_commit((target\_ln\_x2, curr\_ln\_y), True, "HLSeg4\_H\_Corner"); curr\_ln\_x = target\_ln\_x2

if not \_points\_lanes or \_points\_lanes[-1] != exit\_point\_lanes:

last\_path\_pt\_ln = \_points\_lanes[-1]

if last\_path\_pt\_ln[0] != ex\_ln\_x:

\_commit((ex\_ln\_x, last\_path\_pt\_ln[1]), True, "FinalNav\_AlignX")

if \_points\_lanes[-1] != exit\_point\_lanes:

\_commit((ex\_ln\_x, ex\_ln\_y), True, "FinalNav\_AlignY\_to\_Exit")

return {'points\_lanes': \_points\_lanes, 'sow\_flags': \_sow\_flags}

# --- Path Analysis (operates on lane indices) ---

def analyze\_path\_sequence\_fixed(path\_lanes\_list, n\_inner\_x\_sweeps\_val, max\_lx\_idx\_val, max\_ly\_idx\_val, sow\_flags\_list):

if not path\_lanes\_list or len(path\_lanes\_list) < 2: return []

row\_sequence = []

v\_counter = 1; h\_counter = 1

labeled\_v\_lanes = set(); labeled\_h\_lanes = set()

# CORRECTED: Only count inner vertical lanes as VRows (lanes 1 to max-1)

# Outer lanes (0 and max) are headland boundaries, not counted as separate rows

first\_inner\_lx = 1

last\_inner\_lx = max\_lx\_idx\_val - 1

for i in range(len(path\_lanes\_list) - 1):

lx1, ly1 = path\_lanes\_list[i]; lx2, ly2 = path\_lanes\_list[i+1]

is\_sown = i < len(sow\_flags\_list) and sow\_flags\_list[i]

label = ""; mov\_type = "Other"

if lx1 == lx2 and ly1 != ly2:

mov\_type = "VRow\_Path"

# Only label inner vertical lanes as VRows

if is\_sown and first\_inner\_lx <= lx1 <= last\_inner\_lx and lx1 not in labeled\_v\_lanes:

label = f"VRow{v\_counter}"; labeled\_v\_lanes.add(lx1); v\_counter += 1

# Outer vertical lanes get boundary labels instead

elif is\_sown and (lx1 == 0 or lx1 == max\_lx\_idx\_val):

label = f"Boundary\_V{lx1}"

elif ly1 == ly2 and lx1 != lx2:

mov\_type = "HRow\_Path"

if is\_sown and (ly1 == 0 or ly1 == max\_ly\_idx\_val) and ly1 not in labeled\_h\_lanes:

label = f"HRow{h\_counter}"; labeled\_h\_lanes.add(ly1); h\_counter += 1

elif is\_sown:

label = f"H-Turn{i+1}"

row\_sequence.append({'segment\_path\_index': i, 'movement\_type': mov\_type,

'from\_pos\_lanes': (lx1, ly1), 'to\_pos\_lanes': (lx2, ly2),

'label': label, 'is\_sown': is\_sown})

return row\_sequence

def get\_movement\_analysis(path\_lanes\_list, seg\_idx, max\_lx\_idx\_val, max\_ly\_idx\_val, analyzed\_row\_seq, sow\_flags\_all, rover\_width\_m\_val, rover\_length\_m\_val):

if seg\_idx >= len(path\_lanes\_list) - 1 or seg\_idx >= len(sow\_flags\_all): return None

lx1, ly1 = path\_lanes\_list[seg\_idx]; lx2, ly2 = path\_lanes\_list[seg\_idx+1]

# FIXED: Use correct dimensions for lane center calculation

from\_pos\_m\_center = ((lx1 + 0.5) \* rover\_width\_m\_val, (ly1 + 0.5) \* rover\_length\_m\_val)

to\_pos\_m\_center = ((lx2 + 0.5) \* rover\_width\_m\_val, (ly2 + 0.5) \* rover\_length\_m\_val)

# FIXED: Use correct dimensions for distance calculation

distance\_m = abs(lx2 - lx1) \* rover\_width\_m\_val + abs(ly2 - ly1) \* rover\_length\_m\_val

row\_info = next((rs for rs in analyzed\_row\_seq if rs['segment\_path\_index'] == seg\_idx), None)

label = row\_info['label'] if row\_info and row\_info['label'] else f"Segment{seg\_idx + 1}"

analysis = {'from\_pos\_m': from\_pos\_m\_center, 'to\_pos\_m': to\_pos\_m\_center,

'from\_row\_y\_coord\_m': from\_pos\_m\_center[1], 'to\_row\_y\_coord\_m': to\_pos\_m\_center[1],

'distance\_m': distance\_m, 'direction': '', 'action': '',

'farming\_type': '', 'status': '', 'row\_sequence\_label': label}

if lx2 > lx1: analysis['direction'] = 'EAST'

elif lx2 < lx1: analysis['direction'] = 'WEST'

elif ly2 > ly1: analysis['direction'] = 'NORTH'

elif ly2 < ly1: analysis['direction'] = 'SOUTH'

is\_sown = sow\_flags\_all[seg\_idx]

first\_inner\_lx = 1

last\_inner\_lx = max\_lx\_idx\_val - 1

if not is\_sown:

analysis['farming\_type'] = 'NONE'; analysis['action'] = 'NAVIGATION\_UNSOWN'; analysis['status'] = 'TRAVERSING\_NO\_SOW'

else:

if lx1 == lx2 and ly1 != ly2:

analysis['status'] = 'SOWING\_VERTICALLY'

is\_inner\_v\_sweep = (first\_inner\_lx <= lx1 <= last\_inner\_lx)

analysis['action'] = 'INNER\_VERTICAL\_FARMING' if is\_inner\_v\_sweep else 'BOUNDARY\_VERTICAL\_FARMING'

analysis['farming\_type'] = 'CROP\_PLANTING\_V' if is\_inner\_v\_sweep else 'PERIMETER\_SOWING\_V'

elif ly1 == ly2 and lx1 != lx2:

analysis['status'] = 'SOWING\_HORIZONTALLY'

is\_headland\_h = (ly1 == 0 or ly1 == max\_ly\_idx\_val)

analysis['action'] = 'BOUNDARY\_HORIZONTAL\_FARMING' if is\_headland\_h else 'TRANSITION\_SOWING\_H'

analysis['farming\_type'] = 'PERIMETER\_SOWING\_H' if is\_headland\_h else 'TRANSITION\_SOWING\_H'

else:

analysis['action'] = 'DIAGONAL\_SOWING\_ERROR'; analysis['farming\_type'] = 'ERROR\_SOW'; analysis['status'] = 'SOWING\_ERROR\_PATH'

return analysis

# --- Telemetry Logger ---

class LiveTelemetryLogger:

def \_\_init\_\_(self, farm\_w\_m, farm\_b\_m, rover\_lw\_m, exit\_info\_str):

self.farm\_w\_m = farm\_w\_m; self.farm\_b\_m = farm\_b\_m; self.rover\_lw\_m = rover\_lw\_m

self.sown\_v\_segs = 0; self.sown\_h\_segs = 0

self.total\_dist\_m = 0; self.total\_sow\_dist\_m = 0

self.start\_time = datetime.now(); self.csv\_filename = "navigation\_log.csv"

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

try:

with open(self.csv\_filename, 'w', newline='', encoding='utf-8') as f:

w = csv.writer(f); w.writerow(["Timestamp", "Step", "Label", "From (m)", "To (m)", "FromY (m)", "ToY (m)", "Dir", "Action", "FarmType", "Status", "SegDist (m)", "TotalDist (m)", "SownDist (m)", "V\_Sown\_Segs", "H\_Sown\_Segs"])

print(f"💾 CSV created: {self.csv\_filename}")

except IOError as e: print(f"❌ CSV Error: {e}")

else: print(f"📝 Appending to: {self.csv\_filename}")

hdr = "🤖 FARM ROBOT TELEMETRY 🤖"; print(f"\n{hdr}\n{'='\*len(hdr)}\n📊 Farm: {farm\_w\_m}x{farm\_b\_m}m, Rover Lane: {rover\_lw\_m}m\n🎯 Exit: {exit\_info\_str}\n⏰ Start: {self.start\_time:%Y-%m-%d %H:%M:%S}\n{'='\*len(hdr)}\n🔴 LIVE LOG:\n{'='\*len(hdr)}")

def log\_movement(self, step, analysis, time\_now):

if not analysis: return

self.total\_dist\_m += analysis['distance\_m']

if analysis['farming\_type'] != 'NONE':

self.total\_sow\_dist\_m += analysis['distance\_m']

if 'VERTICAL' in analysis['action'] or '\_V' in analysis['farming\_type']: self.sown\_v\_segs += 1

elif 'HORIZONTAL' in analysis['action'] or '\_H' in analysis['farming\_type']: self.sown\_h\_segs += 1

elapsed = (time\_now - self.start\_time).total\_seconds()

display\_label = analysis['row\_sequence\_label']

from\_pos\_str = f"({analysis['from\_pos\_m'][0]:.1f}, {analysis['from\_pos\_m'][1]:.1f})"

to\_pos\_str = f"({analysis['to\_pos\_m'][0]:.1f}, {analysis['to\_pos\_m'][1]:.1f})"

print(f"\n⏱️ {time\_now:%H:%M:%S.%f}"[:-3] + f" [S{step:02d}] (+{elapsed:.1f}s)\n🏷️ {display\_label}\n📍 {from\_pos\_str} → {to\_pos\_str} (D:{analysis['distance\_m']:.1f}m)\n🧭 Act: {analysis['action']} ({analysis['status']}) | Type: {analysis['farming\_type']}\n📊 TD:{self.total\_dist\_m:.1f}m SD:{self.total\_sow\_dist\_m:.1f}m VS:{self.sown\_v\_segs} HS:{self.sown\_h\_segs}\n{'-'\*70}")

row = [time\_now.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3], step, display\_label, str(analysis['from\_pos\_m']), str(analysis['to\_pos\_m']), f"{analysis['from\_row\_y\_coord\_m']:.1f}", f"{analysis['to\_row\_y\_coord\_m']:.1f}", analysis['direction'], analysis['action'], analysis['farming\_type'], analysis['status'], f"{analysis['distance\_m']:.1f}", f"{self.total\_dist\_m:.1f}", f"{self.total\_sow\_dist\_m:.1f}", self.sown\_v\_segs, self.sown\_h\_segs]

try:

with open(self.csv\_filename, 'a', newline='', encoding='utf-8') as f: csv.writer(f).writerow(row)

except IOError as e: print(f"❌ CSV Write Err (S{step}): {e}")

def finalize\_mission(self, final\_pos\_m):

end\_time = datetime.now(); duration = (end\_time - self.start\_time).total\_seconds()

eff = (self.total\_sow\_dist\_m / self.total\_dist\_m \* 100) if self.total\_dist\_m > 0 else 0

final\_pos\_str = f"({final\_pos\_m[0]:.1f}, {final\_pos\_m[1]:.1f})m"

summary = f"\n🏁 MISSION COMPLETE! 🏁\n{'='\*25}\n📍 End: {final\_pos\_str}\n⏰ Time: {duration:.2f}s\n📏 TD: {self.total\_dist\_m:.1f}m\n🌱 SD: {self.total\_sow\_dist\_m:.1f}m ({eff:.1f}%)\n🚜 VS: {self.sown\_v\_segs}\n↔️ HS: {self.sown\_h\_segs}\n{'='\*25}"

print(summary)

row = [end\_time.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3], "FINAL", "End", str(final\_pos\_m), "", "", "", "", "", "", "", f"{duration:.2f}", f"{self.total\_dist\_m:.1f}", f"{self.total\_sow\_dist\_m:.1f}", self.sown\_v\_segs, self.sown\_h\_segs]

try:

with open(self.csv\_filename, 'a', newline='', encoding='utf-8') as f: csv.writer(f).writerow(row)

print(f"💾 Final summary logged to {self.csv\_filename}")

except IOError as e: print(f"❌ CSV Final Err: {e}")

# --- Animation ---

def interpolate\_path(path\_pts\_metric\_centers, pts\_per\_seg=25):

if not path\_pts\_metric\_centers or len(path\_pts\_metric\_centers) < 2: return np.array([]), np.array([])

sx\_m, sy\_m = [], []

for i in range(len(path\_pts\_metric\_centers) - 1):

x0,y0 = path\_pts\_metric\_centers[i]; x1,y1 = path\_pts\_metric\_centers[i+1]

sx\_m.extend(np.linspace(x0, x1, pts\_per\_seg)); sy\_m.extend(np.linspace(y0, y1, pts\_per\_seg))

return np.array(sx\_m), np.array(sy\_m)

def animate\_robot(n\_inner\_x\_sweeps\_val, max\_lx\_idx\_val, max\_ly\_idx\_val, title\_suffix\_str,

path\_lanes\_list, sow\_flags\_all\_list,

exit\_vis\_lanes, farm\_w\_m\_val, farm\_b\_m\_val, rover\_width\_m\_val, rover\_length\_m\_val):

if not path\_lanes\_list or len(path\_lanes\_list) < 2: print("❌ Anim Err: Path short."); return None

if len(sow\_flags\_all\_list) != len(path\_lanes\_list) -1 :

print(f"❌ Anim Err: Mismatch sow\_flags ({len(sow\_flags\_all\_list)}) and segments ({len(path\_lanes\_list)-1}).")

sow\_flags\_all\_list.extend([False] \* (max(0, len(path\_lanes\_list) - 1 - len(sow\_flags\_all\_list))))

# FIXED: Use correct dimensions for path metric centers

path\_metric\_centers\_list = [((ln\_x + 0.5) \* rover\_width\_m\_val, (ln\_y + 0.5) \* rover\_length\_m\_val) for ln\_x, ln\_y in path\_lanes\_list]

exit\_vis\_metric\_center = ((exit\_vis\_lanes[0] + 0.5) \* rover\_width\_m\_val, (exit\_vis\_lanes[1] + 0.5) \* rover\_length\_m\_val)

# FIXED: Use the existing analyze\_path\_sequence\_fixed function instead of missing function

row\_labels\_info\_list = analyze\_path\_sequence\_fixed(path\_lanes\_list, n\_inner\_x\_sweeps\_val, max\_lx\_idx\_val, max\_ly\_idx\_val, sow\_flags\_all\_list)

seg\_labels\_info\_list = analyze\_path\_sequence\_fixed(path\_lanes\_list, n\_inner\_x\_sweeps\_val, max\_lx\_idx\_val, max\_ly\_idx\_val, sow\_flags\_all\_list)

logger\_obj = LiveTelemetryLogger(farm\_w\_m\_val, farm\_b\_m\_val, rover\_width\_m\_val, rover\_length\_m\_val, title\_suffix\_str)

smooth\_x\_m\_centers, smooth\_y\_m\_centers = interpolate\_path(path\_metric\_centers\_list)

if smooth\_x\_m\_centers.size == 0: print("❌ Anim Err: Interpolated path empty."); return None

# FIXED: Pass rover\_length\_m\_val to get\_movement\_analysis

analyses\_metric\_list = [get\_movement\_analysis(path\_lanes\_list, i, max\_lx\_idx\_val, max\_ly\_idx\_val, seg\_labels\_info\_list, sow\_flags\_all\_list, rover\_width\_m\_val, rover\_length\_m\_val) for i in range(len(path\_lanes\_list) - 1)]

fig, ax = plt.subplots(figsize=(12, 10)); ax.set\_aspect('equal')

plot\_padding\_m = max(rover\_width\_m\_val, rover\_length\_m\_val) \* 0.5

ax.set\_xlim(-plot\_padding\_m, farm\_w\_m\_val + plot\_padding\_m)

ax.set\_ylim(-plot\_padding\_m, farm\_b\_m\_val + plot\_padding\_m)

ax.grid(True, linestyle=':', alpha=0.6)

ax.set\_title(f'🤖 Farm: {farm\_w\_m\_val}x{farm\_b\_m\_val}m, Rover: {rover\_width\_m\_val}x{rover\_length\_m\_val}m ({title\_suffix\_str})', fontsize=14, pad=20)

ax.add\_patch(Rectangle((0,0), farm\_w\_m\_val, farm\_b\_m\_val, fill=False, edgecolor='darkgray', lw=2, zorder=1))

# FIXED: Draw row labels with proper positioning using correct dimensions

# Get unique columns for VRow labels from vertical movements

vrow\_columns = set()

first\_inner\_lx = 1

last\_inner\_lx = max\_lx\_idx\_val - 1

for i in range(len(path\_lanes\_list) - 1):

lx1, ly1 = path\_lanes\_list[i]

lx2, ly2 = path\_lanes\_list[i + 1]

is\_sown = i < len(sow\_flags\_all\_list) and sow\_flags\_all\_list[i]

# Vertical movement in inner lanes that are sown

if lx1 == lx2 and ly1 != ly2 and is\_sown and first\_inner\_lx <= lx1 <= last\_inner\_lx:

vrow\_columns.add(lx1)

# Sort columns and assign VRow labels (all on bottom side)

sorted\_vrow\_columns = sorted(vrow\_columns)

for i, column\_x in enumerate(sorted\_vrow\_columns):

tx\_m = (column\_x + 0.5) \* rover\_width\_m\_val

ty\_m = -plot\_padding\_m \* 0.7 # All VRow labels on bottom side

label = f'VRow{i+1}'

ax.text(tx\_m, ty\_m, label, fontsize=10, color='navy',

ha='center', va='center', weight='bold',

bbox=dict(boxstyle="round,pad=0.3", fc='lightblue', alpha=0.9, ec='navy', lw=2))

# FIXED: Position HRow labels based on actual horizontal paths in the robot's route

# Find the Y-coordinates of horizontal movements in the path

horizontal\_y\_positions = set()

for i in range(len(path\_lanes\_list) - 1):

x1, y1 = path\_lanes\_list[i]

x2, y2 = path\_lanes\_list[i + 1]

is\_sown = i < len(sow\_flags\_all\_list) and sow\_flags\_all\_list[i]

if y1 == y2 and x1 != x2 and is\_sown: # Horizontal movement that is sown

horizontal\_y\_positions.add(y1)

# Sort horizontal positions and assign labels

sorted\_horizontal\_positions = sorted(horizontal\_y\_positions)

if len(sorted\_horizontal\_positions) >= 1:

# HRow1 (bottom horizontal path) - positioned on left side

hrow1\_y\_lane = sorted\_horizontal\_positions[0]

hrow1\_tx\_m = -plot\_padding\_m \* 0.7

hrow1\_ty\_m = (hrow1\_y\_lane + 0.5) \* rover\_length\_m\_val # FIXED: Use rover\_length\_m\_val

ax.text(hrow1\_tx\_m, hrow1\_ty\_m, 'HRow1', fontsize=10, color='darkgreen',

ha='center', va='center', weight='bold', rotation=0,

bbox=dict(boxstyle="round,pad=0.3", fc='lightgreen', alpha=0.9, ec='darkgreen', lw=2))

if len(sorted\_horizontal\_positions) >= 2:

# HRow2 (top horizontal path) - positioned on left side

hrow2\_y\_lane = sorted\_horizontal\_positions[-1] # Last (topmost) horizontal position

hrow2\_tx\_m = -plot\_padding\_m \* 0.7

hrow2\_ty\_m = (hrow2\_y\_lane + 0.5) \* rover\_length\_m\_val # FIXED: Use rover\_length\_m\_val

ax.text(hrow2\_tx\_m, hrow2\_ty\_m, 'HRow2', fontsize=10, color='darkgreen',

ha='center', va='center', weight='bold', rotation=0,

bbox=dict(boxstyle="round,pad=0.3", fc='lightgreen', alpha=0.9, ec='darkgreen', lw=2))

# FIXED: Draw full path trace using rectangles with proper rover dimensions

single\_brown\_color = '#8B4513' # Consistent brown color

for i in range(len(path\_metric\_centers\_list) - 1):

x1, y1 = path\_metric\_centers\_list[i]

x2, y2 = path\_metric\_centers\_list[i + 1]

if x1 == x2: # Vertical movement

rect\_x = x1 - rover\_width\_m\_val / 2

rect\_y = min(y1, y2) - rover\_length\_m\_val / 2 # FIXED: Use rover\_length\_m\_val

rect\_width = rover\_width\_m\_val

rect\_height = abs(y2 - y1) + rover\_length\_m\_val # FIXED: Use rover\_length\_m\_val

else: # Horizontal movement

rect\_x = min(x1, x2) - rover\_width\_m\_val / 2

rect\_y = y1 - rover\_length\_m\_val / 2

rect\_width = abs(x2 - x1) + rover\_width\_m\_val

rect\_height = rover\_length\_m\_val

# Add rectangle for full path trace

path\_rect = Rectangle((rect\_x, rect\_y), rect\_width, rect\_height,

color=single\_brown\_color, alpha=0.4, zorder=2)

ax.add\_patch(path\_rect)

# FIXED: Pre-create all sown rectangles with proper rover dimensions

sown\_rectangles = []

sown\_progress\_masks = [] # To track partial visibility parameters

for i in range(len(path\_lanes\_list) - 1):

if sow\_flags\_all\_list[i]:

x1, y1 = path\_metric\_centers\_list[i]

x2, y2 = path\_metric\_centers\_list[i + 1]

if x1 == x2: # Vertical movement

rect\_x = x1 - rover\_width\_m\_val / 2

rect\_y = min(y1, y2) - rover\_length\_m\_val / 2 # FIXED: Use rover\_length\_m\_val

rect\_width = rover\_width\_m\_val

rect\_height = abs(y2 - y1) + rover\_length\_m\_val # FIXED: Use rover\_length\_m\_val

else: # Horizontal movement

rect\_x = min(x1, x2) - rover\_width\_m\_val / 2

rect\_y = y1 - rover\_length\_m\_val / 2

rect\_width = abs(x2 - x1) + rover\_width\_m\_val

rect\_height = rover\_length\_m\_val

# Create a clipping rectangle that will grow gradually

# Start with zero size in the movement direction

if x1 == x2: # Vertical - start with zero height

clip\_rect = Rectangle((rect\_x, rect\_y), rect\_width, 0,

color='#006400', alpha=0.8, zorder=3, visible=False)

else: # Horizontal - start with zero width

clip\_rect = Rectangle((rect\_x, rect\_y), 0, rect\_height,

color='#006400', alpha=0.8, zorder=3, visible=False)

ax.add\_patch(clip\_rect)

sown\_rectangles.append((i, clip\_rect))

sown\_progress\_masks.append({

'full\_width': rect\_width,

'full\_height': rect\_height,

'is\_vertical': x1 == x2,

'base\_x': rect\_x,

'base\_y': rect\_y,

'start\_pos': (x1, y1),

'end\_pos': (x2, y2)

})

else:

sown\_rectangles.append((i, None))

sown\_progress\_masks.append(None)

# Create dummy lines for legend with rover dimensions info

full\_path\_trace\_line, = ax.plot([], [], color=single\_brown\_color, lw=10, alpha=0.4, label=f'Unsown Path (Rover:{rover\_width\_m\_val}x{rover\_length\_m\_val}m)')

sown\_segments\_line, = ax.plot([], [], color='#006400', lw=10, alpha=0.8, label=f'Sown Area (Rover:{rover\_width\_m\_val}x{rover\_length\_m\_val}m)')

# UPDATED: Use actual rover dimensions for visual representation

rover\_body\_width\_m = rover\_width\_m\_val

rover\_body\_height\_m = rover\_length\_m\_val # Use rover length for visual height

initial\_rover\_center\_m = path\_metric\_centers\_list[0]

robot\_body\_patch = Rectangle((initial\_rover\_center\_m[0] - rover\_body\_width\_m/2, initial\_rover\_center\_m[1] - rover\_body\_height\_m/2),

rover\_body\_width\_m, rover\_body\_height\_m, color='orange', ec='black', lw=1.5, zorder=4)

ax.add\_patch(robot\_body\_patch)

start\_marker\_center\_m = path\_metric\_centers\_list[0]; marker\_radius\_m = 0.4 \* max(rover\_width\_m\_val, rover\_length\_m\_val)

ax.add\_patch(Circle(start\_marker\_center\_m, marker\_radius\_m, color='blue', fill=True, lw=2.5, zorder=5, alpha=0.5))

ax.add\_patch(Circle(start\_marker\_center\_m, marker\_radius\_m, color='blue', fill=False, lw=2.5, zorder=5, hatch='//'))

# Create gate-style exit marker instead of circle

gate\_width = rover\_width\_m\_val \* 0.8

gate\_height = rover\_length\_m\_val \* 0.3 # FIXED: Use rover\_length\_m\_val for gate height

# Determine which border the exit is on and position gate accordingly

exit\_x, exit\_y = exit\_vis\_lanes[0], exit\_vis\_lanes[1]

if exit\_x == 0: # Left border

gate\_x = -gate\_height/2

gate\_y = exit\_vis\_metric\_center[1] - gate\_width/2

gate\_w, gate\_h = gate\_height, gate\_width

elif exit\_x == max\_lx\_idx\_val: # Right border

gate\_x = farm\_w\_m\_val - gate\_height/2

gate\_y = exit\_vis\_metric\_center[1] - gate\_width/2

gate\_w, gate\_h = gate\_height, gate\_width

elif exit\_y == 0: # Bottom border

gate\_x = exit\_vis\_metric\_center[0] - gate\_width/2

gate\_y = -gate\_height/2

gate\_w, gate\_h = gate\_width, gate\_height

else: # Top border

gate\_x = exit\_vis\_metric\_center[0] - gate\_width/2

gate\_y = farm\_b\_m\_val - gate\_height/2

gate\_w, gate\_h = gate\_width, gate\_height

# Add the gate marker

exit\_gate = Rectangle((gate\_x, gate\_y), gate\_w, gate\_h,

color='red', alpha=0.8, zorder=5,

edgecolor='darkred', linewidth=2)

ax.add\_patch(exit\_gate)

legend\_handles = [

Line2D([0],[0],c=single\_brown\_color,lw=10,alpha=0.4,label=f'Unsown Path (Rover:{rover\_width\_m\_val}x{rover\_length\_m\_val}m)'),

Line2D([0],[0],c='#006400',lw=10,alpha=0.8,label=f'Sown Area (Rover:{rover\_width\_m\_val}x{rover\_length\_m\_val}m)'),

Rectangle((0,0), 1, 1, fc='orange', ec='black', label=f'🤖 Rover ({rover\_width\_m\_val}x{rover\_length\_m\_val}m)'),

Line2D([0],[0],marker='o',mfc='blue',mec='blue',ms=10,ls='None',label=f'🔵 START ({start\_marker\_center\_m[0]:.1f}, {start\_marker\_center\_m[1]:.1f})m'),

Rectangle((0,0), 1, 1, fc='red', ec='darkred', label=f'🚪 EXIT GATE ({exit\_vis\_metric\_center[0]:.1f}, {exit\_vis\_metric\_center[1]:.1f})m')

]

ax.legend(handles=legend\_handles,loc='upper right',bbox\_to\_anchor=(1.28,1.02),fontsize=8); plt.subplots\_adjust(right=0.75)

logged\_segments\_indices = set(); logger\_obj.mission\_finalized = False; total\_animation\_frames = len(smooth\_x\_m\_centers)

num\_orig\_segments = len(path\_metric\_centers\_list) -1

pts\_per\_orig\_segment\_approx = total\_animation\_frames // num\_orig\_segments if num\_orig\_segments > 0 else total\_animation\_frames

def init\_animation\_func():

robot\_body\_patch.set\_xy((smooth\_x\_m\_centers[0]-rover\_body\_width\_m/2, smooth\_y\_m\_centers[0]-rover\_body\_height\_m/2))

return [robot\_body\_patch]

def update\_animation\_func(frame\_idx):

current\_x\_center\_m = smooth\_x\_m\_centers[frame\_idx]

current\_y\_center\_m = smooth\_y\_m\_centers[frame\_idx]

robot\_body\_patch.set\_xy((current\_x\_center\_m - rover\_body\_width\_m/2, current\_y\_center\_m - rover\_body\_height\_m/2))

current\_original\_segment\_idx = frame\_idx // pts\_per\_orig\_segment\_approx if pts\_per\_orig\_segment\_approx > 0 else num\_orig\_segments -1

current\_original\_segment\_idx = min(current\_original\_segment\_idx, num\_orig\_segments - 1)

if current\_original\_segment\_idx != -1 and current\_original\_segment\_idx < len(analyses\_metric\_list) and current\_original\_segment\_idx not in logged\_segments\_indices:

if analyses\_metric\_list[current\_original\_segment\_idx]:

logger\_obj.log\_movement(current\_original\_segment\_idx + 1, analyses\_metric\_list[current\_original\_segment\_idx], datetime.now())

logged\_segments\_indices.add(current\_original\_segment\_idx)

# Show sown rectangles gradually as rover moves - FIXED VERSION

if current\_original\_segment\_idx >= 0:

# Make completed segments fully visible with correct dimensions

for i in range(min(current\_original\_segment\_idx, len(sown\_rectangles))):

if i < len(sown\_rectangles) and sown\_rectangles[i][1] is not None:

rect = sown\_rectangles[i][1]

mask = sown\_progress\_masks[i]

rect.set\_visible(True)

# Set to full dimensions immediately for completed segments

if mask['is\_vertical']:

rect.set\_height(mask['full\_height'])

rect.set\_y(mask['base\_y']) # Ensure correct position

else:

rect.set\_width(mask['full\_width'])

rect.set\_x(mask['base\_x']) # Ensure correct position

# Gradually show current segment based on rover progress

if (current\_original\_segment\_idx < len(sown\_rectangles) and

sown\_rectangles[current\_original\_segment\_idx][1] is not None and

current\_original\_segment\_idx < len(sow\_flags\_all\_list) and

sow\_flags\_all\_list[current\_original\_segment\_idx]):

current\_rect = sown\_rectangles[current\_original\_segment\_idx][1]

current\_mask = sown\_progress\_masks[current\_original\_segment\_idx]

current\_rect.set\_visible(True)

# Calculate progress within current segment

segment\_start\_frame = current\_original\_segment\_idx \* pts\_per\_orig\_segment\_approx

frames\_in\_segment = min(pts\_per\_orig\_segment\_approx, total\_animation\_frames - segment\_start\_frame)

progress\_in\_segment = (frame\_idx - segment\_start\_frame) / frames\_in\_segment if frames\_in\_segment > 0 else 1

progress\_in\_segment = max(0, min(1, progress\_in\_segment))

if current\_mask['is\_vertical']:

# Vertical movement - grow height gradually

new\_height = current\_mask['full\_height'] \* progress\_in\_segment

current\_rect.set\_height(new\_height)

# Determine growth direction based on start/end positions

start\_y, end\_y = current\_mask['start\_pos'][1], current\_mask['end\_pos'][1]

if end\_y < start\_y: # Moving from top to bottom

# Rectangle grows from top, so adjust y position

new\_y = current\_mask['base\_y'] + current\_mask['full\_height'] - new\_height

current\_rect.set\_y(new\_y)

else: # Moving from bottom to top

# Rectangle grows from bottom, keep base\_y

current\_rect.set\_y(current\_mask['base\_y'])

else:

# Horizontal movement - grow width gradually

new\_width = current\_mask['full\_width'] \* progress\_in\_segment

current\_rect.set\_width(new\_width)

# Determine growth direction based on start/end positions

start\_x, end\_x = current\_mask['start\_pos'][0], current\_mask['end\_pos'][0]

if end\_x < start\_x: # Moving from right to left

# Rectangle grows from right, so adjust x position

new\_x = current\_mask['base\_x'] + current\_mask['full\_width'] - new\_width

current\_rect.set\_x(new\_x)

else: # Moving from left to right

# Rectangle grows from left, keep base\_x

current\_rect.set\_x(current\_mask['base\_x'])

# Final frame handling - ensure all sown rectangles are properly displayed

if frame\_idx >= total\_animation\_frames - 1 and not logger\_obj.mission\_finalized:

# Make all remaining sown rectangles fully visible with correct dimensions

for i, (\_, rect) in enumerate(sown\_rectangles):

if rect is not None:

rect.set\_visible(True)

if i < len(sown\_progress\_masks) and sown\_progress\_masks[i]:

mask = sown\_progress\_masks[i]

if mask['is\_vertical']:

rect.set\_height(mask['full\_height'])

rect.set\_y(mask['base\_y'])

else:

rect.set\_width(mask['full\_width'])

rect.set\_x(mask['base\_x'])

for i\_log\_final\_check in range(len(analyses\_metric\_list)):

if i\_log\_final\_check not in logged\_segments\_indices and analyses\_metric\_list[i\_log\_final\_check]:

logger\_obj.log\_movement(i\_log\_final\_check+1, analyses\_metric\_list[i\_log\_final\_check], datetime.now())

logger\_obj.finalize\_mission(path\_metric\_centers\_list[-1])

logger\_obj.mission\_finalized = True

return [robot\_body\_patch] + [rect for \_, rect in sown\_rectangles if rect is not None and rect.get\_visible()]

animation\_obj = FuncAnimation(fig, update\_animation\_func, frames=total\_animation\_frames,

init\_func=init\_animation\_func, blit=True, interval=50, repeat=False)

plt.show()

return animation\_obj

class LiveTelemetryLogger:

def \_\_init\_\_(self, farm\_w\_m, farm\_b\_m, rover\_width\_m, rover\_length\_m, exit\_info\_str):

self.farm\_w\_m = farm\_w\_m; self.farm\_b\_m = farm\_b\_m; self.rover\_width\_m = rover\_width\_m; self.rover\_length\_m = rover\_length\_m

self.sown\_v\_segs = 0; self.sown\_h\_segs = 0

self.total\_dist\_m = 0; self.total\_sow\_dist\_m = 0

self.start\_time = datetime.now(); self.csv\_filename = "navigation\_log.csv"

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

try:

with open(self.csv\_filename, 'w', newline='', encoding='utf-8') as f:

w = csv.writer(f); w.writerow(["Timestamp", "Step", "Label", "From (m)", "To (m)", "FromY (m)", "ToY (m)", "Dir", "Action", "FarmType", "Status", "SegDist (m)", "TotalDist (m)", "SownDist (m)", "V\_Sown\_Segs", "H\_Sown\_Segs"])

print(f"💾 CSV created: {self.csv\_filename}")

except IOError as e: print(f"❌ CSV Error: {e}")

else: print(f"📝 Appending to: {self.csv\_filename}")

hdr = "🤖 FARM ROBOT TELEMETRY 🤖"; print(f"\n{hdr}\n{'='\*len(hdr)}\n📊 Farm: {farm\_w\_m}x{farm\_b\_m}m, Rover: {rover\_width\_m}x{rover\_length\_m}m\n🎯 Exit: {exit\_info\_str}\n⏰ Start: {self.start\_time:%Y-%m-%d %H:%M:%S}\n{'='\*len(hdr)}\n🔴 LIVE LOG:\n{'='\*len(hdr)}")

def log\_movement(self, step, analysis, time\_now):

if not analysis: return

self.total\_dist\_m += analysis['distance\_m']

if analysis['farming\_type'] != 'NONE':

self.total\_sow\_dist\_m += analysis['distance\_m']

if 'VERTICAL' in analysis['action'] or '\_V' in analysis['farming\_type']: self.sown\_v\_segs += 1

elif 'HORIZONTAL' in analysis['action'] or '\_H' in analysis['farming\_type']: self.sown\_h\_segs += 1

elapsed = (time\_now - self.start\_time).total\_seconds()

display\_label = analysis['row\_sequence\_label']

from\_pos\_str = f"({analysis['from\_pos\_m'][0]:.1f}, {analysis['from\_pos\_m'][1]:.1f})"

to\_pos\_str = f"({analysis['to\_pos\_m'][0]:.1f}, {analysis['to\_pos\_m'][1]:.1f})"

print(f"\n⏱️ {time\_now:%H:%M:%S.%f}"[:-3] + f" [S{step:02d}] (+{elapsed:.1f}s)\n🏷️ {display\_label}\n📍 {from\_pos\_str} → {to\_pos\_str} (D:{analysis['distance\_m']:.1f}m)\n🧭 Act: {analysis['action']} ({analysis['status']}) | Type: {analysis['farming\_type']}\n📊 TD:{self.total\_dist\_m:.1f}m SD:{self.total\_sow\_dist\_m:.1f}m VS:{self.sown\_v\_segs} HS:{self.sown\_h\_segs}\n{'-'\*70}")

row = [time\_now.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3], step, display\_label, str(analysis['from\_pos\_m']), str(analysis['to\_pos\_m']), f"{analysis['from\_row\_y\_coord\_m']:.1f}", f"{analysis['to\_row\_y\_coord\_m']:.1f}", analysis['direction'], analysis['action'], analysis['farming\_type'], analysis['status'], f"{analysis['distance\_m']:.1f}", f"{self.total\_dist\_m:.1f}", f"{self.total\_sow\_dist\_m:.1f}", self.sown\_v\_segs, self.sown\_h\_segs]

try:

with open(self.csv\_filename, 'a', newline='', encoding='utf-8') as f: csv.writer(f).writerow(row)

except IOError as e: print(f"❌ CSV Write Err (S{step}): {e}")

def finalize\_mission(self, final\_pos\_m):

end\_time = datetime.now(); duration = (end\_time - self.start\_time).total\_seconds()

eff = (self.total\_sow\_dist\_m / self.total\_dist\_m \* 100) if self.total\_dist\_m > 0 else 0

final\_pos\_str = f"({final\_pos\_m[0]:.1f}, {final\_pos\_m[1]:.1f})m"

summary = f"\n🏁 MISSION COMPLETE! 🏁\n{'='\*25}\n📍 End: {final\_pos\_str}\n⏰ Time: {duration:.2f}s\n📏 TD: {self.total\_dist\_m:.1f}m\n🌱 SD: {self.total\_sow\_dist\_m:.1f}m ({eff:.1f}%)\n🚜 VS: {self.sown\_v\_segs}\n↔️ HS: {self.sown\_h\_segs}\n{'='\*25}"

print(summary)

row = [end\_time.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3], "FINAL", "End", str(final\_pos\_m), "", "", "", "", "", "", "", f"{duration:.2f}", f"{self.total\_dist\_m:.1f}", f"{self.total\_sow\_dist\_m:.1f}", self.sown\_v\_segs, self.sown\_h\_segs]

try:

with open(self.csv\_filename, 'a', newline='', encoding='utf-8') as f: csv.writer(f).writerow(row)

print(f"💾 Final summary logged to {self.csv\_filename}")

except IOError as e: print(f"❌ CSV Final Err: {e}")

def main():

print("=== 🤖 ADVANCED FARM ROBOT TRAVERSAL SYSTEM (v12.9) ===")

farm\_width\_m, farm\_breadth\_m, rover\_width\_m, rover\_length\_m = 0.0, 0.0, 0.0, 0.0

while True:

try: farm\_width\_m = float(input("Enter farm WIDTH (X-axis, e.g., 50 meters): ")); assert farm\_width\_m > 0; break

except (ValueError, AssertionError): print("Invalid input. Must be positive number.")

while True:

try: farm\_breadth\_m = float(input("Enter farm BREADTH (Y-axis, e.g., 50 meters): ")); assert farm\_breadth\_m > 0; break

except (ValueError, AssertionError): print("Invalid input. Must be positive number.")

while True:

try: rover\_width\_m = float(input("Enter rover WIDTH (for lane spacing, e.g., 10 meters): ")); assert rover\_width\_m > 0; break

except (ValueError, AssertionError): print("Invalid input. Must be positive number.")

while True:

try: rover\_length\_m = float(input("Enter rover LENGTH (for coverage depth, e.g., 8 meters): ")); assert rover\_length\_m > 0; break

except (ValueError, AssertionError): print("Invalid input. Must be positive number.")

# FIXED: Correct grid calculation logic

# Vertical passes (columns) = Farm\_Width / Rover\_Width

num\_lanes\_x = int(farm\_width\_m / rover\_width\_m)

# Horizontal rows = Farm\_Height / Rover\_Height (rover\_length\_m is the rover's height)

num\_lanes\_y = int(farm\_breadth\_m / rover\_length\_m)

# FIXED: Updated minimum requirements with correct logic

# FIXED: Updated minimum requirements with correct logic

if num\_lanes\_x < 3:

print(f"❌ Farm width {farm\_width\_m}m too small for rover width {rover\_width\_m}m.")

print(f" Need at least 3 vertical passes (2 headlands + 1 inner) = {3 \* rover\_width\_m}m minimum farm width.")

return None

if num\_lanes\_y < 1:

print(f"❌ Farm breadth {farm\_breadth\_m}m too small for rover length {rover\_length\_m}m.")

print(f" Need at least 1 horizontal row = {rover\_length\_m}m minimum farm breadth.")

return None

max\_lane\_idx\_x = num\_lanes\_x - 1

max\_lane\_idx\_y = num\_lanes\_y - 1

# FIXED: Inner sweeps calculation remains the same (based on vertical passes)

n\_inner\_x\_sweeps = num\_lanes\_x - 2

if n\_inner\_x\_sweeps < 0: n\_inner\_x\_sweeps = 0

# FIXED: Updated information display with correct grid logic

total\_coverage\_area = num\_lanes\_x \* num\_lanes\_y \* rover\_width\_m \* rover\_length\_m

farm\_area = farm\_width\_m \* farm\_breadth\_m

coverage\_percentage = (total\_coverage\_area / farm\_area \* 100) if farm\_area > 0 else 0

print(f"\n🚀 Grid Analysis for {farm\_width\_m}x{farm\_breadth\_m}m farm, Rover: {rover\_width\_m}x{rover\_length\_m}m")

print(f"📊 Grid Layout: {num\_lanes\_x} vertical passes × {num\_lanes\_y} horizontal rows")

print(f"📊 Calculation:")

print(f" • Vertical passes = Farm\_Width({farm\_width\_m}) / Rover\_Width({rover\_width\_m}) = {num\_lanes\_x}")

print(f" • Horizontal rows = Farm\_Breadth({farm\_breadth\_m}) / Rover\_Length({rover\_length\_m}) = {num\_lanes\_y}")

print(f"📊 Coverage: {total\_coverage\_area:.1f}m² of {farm\_area:.1f}m² ({coverage\_percentage:.1f}%)")

print(f"📊 Productive rows breakdown:")

print(f" • Inner vertical sweeps: {n\_inner\_x\_sweeps} (lanes 1 to {max\_lane\_idx\_x-1})" if n\_inner\_x\_sweeps > 0 else " • No inner vertical sweeps (farm too narrow)")

print(f" • Outer vertical headlands: 2 (lanes 0 & {max\_lane\_idx\_x})")

print(f" • Horizontal headland rows: {min(2, num\_lanes\_y)} (covering {min(2, num\_lanes\_y) \* rover\_length\_m}m breadth)")

print(f"\nℹ️ Lane Index Ranges:")

print(f" • X-lanes (vertical passes): 0 to {max\_lane\_idx\_x}")

print(f" • Y-lanes (horizontal rows): 0 to {max\_lane\_idx\_y}")

exit\_point\_lanes, anim\_title\_suffix\_str = None, ""

print("\nSelect Exit Type:\n1. Corner Exit (select a corner lane)\n2. Custom Boundary Exit (select a boundary lane)")

nav\_choice\_str = ""

while nav\_choice\_str not in ["1", "2"]: nav\_choice\_str = input("Choice (1-2): ").strip()

is\_corner\_exit\_choice = (nav\_choice\_str == "1")

if is\_corner\_exit\_choice:

exit\_point\_lanes = get\_user\_choice\_corner\_lanes(max\_lane\_idx\_x, max\_lane\_idx\_y)

anim\_title\_suffix\_str = f"Corner Exit (Lane: {exit\_point\_lanes})"

else:

exit\_point\_lanes = get\_user\_defined\_exit\_lanes(max\_lane\_idx\_x, max\_lane\_idx\_y)

anim\_title\_suffix\_str = f"Custom Exit (Lane: {exit\_point\_lanes})"

# FIXED: Exit position calculation using correct dimensions

exit\_metric\_center\_display = ((exit\_point\_lanes[0] + 0.5) \* rover\_width\_m, (exit\_point\_lanes[1] + 0.5) \* rover\_length\_m)

print(f"ℹ️ Target Exit Lane {exit\_point\_lanes} (approx. center: {exit\_metric\_center\_display[0]:.1f}m, {exit\_metric\_center\_display[1]:.1f}m)")

path\_data\_dict = generate\_fixed\_path(n\_inner\_x\_sweeps, max\_lane\_idx\_x, max\_lane\_idx\_y, exit\_point\_lanes, is\_corner\_exit\_choice)

path\_lanes\_list\_gen, sow\_flags\_list\_gen = path\_data\_dict['points\_lanes'], path\_data\_dict['sow\_flags']

if not path\_lanes\_list\_gen or len(path\_lanes\_list\_gen) < 2: print("❌ Path generation failed. Exiting."); return None

if len(sow\_flags\_list\_gen) != len(path\_lanes\_list\_gen)-1:

print(f"❌ CRITICAL MISMATCH: Sow\_flags ({len(sow\_flags\_list\_gen)}) vs segments ({len(path\_lanes\_list\_gen)-1}). Exiting.")

return None

# FIXED: Final summary with correct grid understanding

print(f"\n🎬 MISSION SUMMARY:")

print(f"📊 Farm: {farm\_width\_m}x{farm\_breadth\_m}m | Rover: {rover\_width\_m}x{rover\_length\_m}m")

print(f"📊 Grid: {num\_lanes\_x} vertical passes × {num\_lanes\_y} horizontal rows")

print(f"📊 Path: {len(path\_lanes\_list\_gen)} waypoints, {len(sow\_flags\_list\_gen)} segments")

print(f"📊 Coverage: {coverage\_percentage:.1f}% of farm area")

input("\n🎬 Press Enter to start animation & telemetry...")

final\_animation\_object = None

try:

final\_animation\_object = animate\_robot(

n\_inner\_x\_sweeps, max\_lane\_idx\_x, max\_lane\_idx\_y, anim\_title\_suffix\_str,

path\_lanes\_list\_gen, sow\_flags\_list\_gen, exit\_point\_lanes,

farm\_width\_m, farm\_breadth\_m, rover\_width\_m, rover\_length\_m

)

except KeyboardInterrupt: print("\n\n⚠️ Animation aborted by user.")

except Exception as e\_anim: print(f"\n❌ An error occurred during animation: {e\_anim}"); import traceback; traceback.print\_exc()

return final\_animation\_object

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

run\_animation\_main\_obj = main()

if run\_animation\_main\_obj:

print("\n✅ Animation process completed or started successfully.")

print("📁 Check 'navigation\_log.csv' for telemetry data.")

else:

print("\n🔴 Animation did not complete or was not started due to an error.")