DRIFT ISSUE  
  
I  
mport matplotlib.pyplot as plt

import numpy as np

from scipy.interpolate import CubicSpline

import math

import random

# ——— Configuration —————————————————————————————————————————

TURN\_RATE\_PER\_CM = 20 # degrees per cm of turn capability

DISTANCE\_PER\_STEP = 0.2 # cm traveled per simulation step

LOOKAHEAD\_STEPS = 5 # baseline lookahead for drift recovery

# ——— Helpers ———————————————————————————————————————————————

def generate\_farm\_rows(num\_rows=2, row\_length=20, points\_per\_row=20, row\_spacing=2):

ys = np.linspace(0, row\_length, points\_per\_row)

return [[(i \* row\_spacing, y) for y in ys] for i in range(num\_rows)]

def interpolate\_row(pts, res=0.2):

x, y = zip(\*pts)

t = np.linspace(0, 1, len(x))

csx, csy = CubicSpline(t, x), CubicSpline(t, y)

t2 = np.linspace(0, 1, int(1/res \* len(x)))

return list(zip(csx(t2), csy(t2)))

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

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

def diff\_h(c, t): return (t - c + 540) % 360 - 180

# ——— Drawing ————————————————————————————————————

def draw(ax, paths, pts, pos, hdg, title, xlim=(-2, 6), ylim=(-2, 22)):

ax.clear()

for p in paths:

xs, ys = zip(\*p)

ax.plot(xs, ys, 'k--', lw=2, alpha=0.3)

if len(pts) > 1:

xs, ys = zip(\*pts)

ax.plot(xs, ys, 'b-', lw=3, alpha=0.7)

ax.scatter(xs, ys, c='blue', s=60, alpha=0.6)

ax.plot(pos[0], pos[1], 'ro', markersize=12)

ax.quiver(pos[0], pos[1], math.cos(math.radians(hdg)), math.sin(math.radians(hdg)),

scale=5, width=0.005)

ax.set\_title(title, fontsize=14)

ax.set\_aspect('equal')

ax.set\_autoscale\_on(False)

ax.set\_xlim(\*xlim)

ax.set\_ylim(\*ylim)

ax.grid(True)

plt.draw(); plt.pause(0.05)

# ——— Traverse one row with drift & recovery ——————————————————————

def traverse\_row(ax, paths, row\_idx, pos, hdg, start\_idx=0):

path = paths[row\_idx]

idx = start\_idx

trigger = start\_idx + (len(path) - start\_idx) // 3

lookahead = LOOKAHEAD\_STEPS

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

# unit tangent vector

dx, dy = path[end\_idx][0] - path[trigger][0], path[end\_idx][1] - path[trigger][1]

mag = math.hypot(dx, dy)

ux, uy = (dx/mag, dy/mag) if mag != 0 else (1.0, 0.0)

# drift at ±45° relative to path heading

angle = 45 if random.choice([True, False]) else -45

rad = math.radians(angle)

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

rx = ux \* c + uy \* s

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

# determine recovery index based on drift length and turn rate

turn\_dist = abs(angle) / TURN\_RATE\_PER\_CM # cm needed to turn back

extra\_skip = int(turn\_dist / DISTANCE\_PER\_STEP) # steps traveled during turn

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

recovery\_target = path[recovery\_idx]

drift\_len, steps = 1.0, 20

drift\_delta = (rx \* drift\_len/steps, ry \* drift\_len/steps)

drifted = False

drifting = False

recovering = False

step = 0

while idx < len(path):

# trigger drift

if idx == trigger and not drifted:

drifting = True

drifted = True

step = 0

# DRIFT PHASE

if drifting:

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

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

step += 1

draw(ax, paths, [pos, recovery\_target], pos, hdg, f"Row{row\_idx+1} Drifting")

if step >= steps:

drifting = False

recovering = True

continue

# RECOVERY PHASE

if recovering:

d = dist(pos, recovery\_target)

desired = head(pos, recovery\_target)

dh = diff\_h(hdg, desired)

turn\_step = TURN\_RATE\_PER\_CM \* DISTANCE\_PER\_STEP

if abs(dh) > 5:

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

else:

pos[0] += DISTANCE\_PER\_STEP \* math.cos(math.radians(hdg))

pos[1] += DISTANCE\_PER\_STEP \* math.sin(math.radians(hdg))

draw(ax, paths, [pos, recovery\_target], pos, hdg, f"Row{row\_idx+1} Recovering")

if d < 0.5:

pos[:] = recovery\_target

idx = recovery\_idx + 1

recovering = False

continue

# NORMAL TRAVERSE

target = path[idx]

d = dist(pos, target)

desired = head(pos, target)

dh = diff\_h(hdg, desired)

turn\_step = TURN\_RATE\_PER\_CM \* DISTANCE\_PER\_STEP

if abs(dh) > 5:

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

else:

pos[0] += DISTANCE\_PER\_STEP \* math.cos(math.radians(hdg))

pos[1] += DISTANCE\_PER\_STEP \* math.sin(math.radians(hdg))

draw(ax, paths, [target], pos, hdg, f"Row{row\_idx+1} Traversing")

if d < 0.5:

idx += 1

# finish row

pos[:] = list(path[-1])

draw(ax, paths, path, pos, hdg, f"Row{row\_idx+1} Completed")

plt.pause(1)

return pos, hdg, 0

# ——— Transition & traverse —————————————————————————————

def transition\_and\_traverse(ax, paths, current\_pos, hdg, next\_row\_idx):

path = paths[next\_row\_idx]

d0 = dist(current\_pos, path[0])

dn = dist(current\_pos, path[-1])

if dn < d0:

paths[next\_row\_idx] = list(reversed(path))

path = paths[next\_row\_idx]

target = path[0]

while dist(current\_pos, target) > 0.5:

desired = head(current\_pos, target)

dh = diff\_h(hdg, desired)

turn\_step = TURN\_RATE\_PER\_CM \* DISTANCE\_PER\_STEP

if abs(dh) > 5:

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

else:

current\_pos[0] += DISTANCE\_PER\_STEP \* math.cos(math.radians(hdg))

current\_pos[1] += DISTANCE\_PER\_STEP \* math.sin(math.radians(hdg))

draw(ax, paths, [current\_pos, target], current\_pos, hdg,

f"Transition to Row{next\_row\_idx+1}")

return traverse\_row(ax, paths, next\_row\_idx, current\_pos, hdg, start\_idx=0)

# ——— Main Simulation —————————————————————————————

def simulate\_full():

rows = generate\_farm\_rows(num\_rows=2, row\_length=20, points\_per\_row=30, row\_spacing=3)

paths = [interpolate\_row(r) for r in rows]

fig, ax = plt.subplots(figsize=(12, 9))

plt.ion()

pos = list(paths[0][0])

hdg = 90

pos, hdg, \_ = traverse\_row(ax, paths, 0, pos, hdg)

for next\_idx in range(1, len(paths)):

pos, hdg, \_ = transition\_and\_traverse(ax, paths, pos, hdg, next\_idx)

plt.ioff()

plt.show()

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

simulate\_full()