import matplotlib.pyplot as plt

import datetime

import time

import serial

import numpy as np

import random

import logging\_100mm

# Import our modules

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

from row\_navigation import RowNavigator

from farm\_safety import SafetyModule

from sleep\_mode import FailsafeModule, GPSFailsafeReason, DriftSeverity, DriftAction

# Import the health check module from the second file

from rover\_health\_check import RoverHealthCheck, HealthCheckFailure

# Import coordinate converter

from coordinate\_converter import CoordinateConverter

import threading

from ntrip\_client import NTRIPClient

from emlid\_gps\_integration import EmlidGPSReader,update\_rover\_from\_emlid, setup\_emlid\_integration

from gps\_system\_monitor import GPSSystemMonitor

import sys

import logging

import os

import csv

# Ensure port is available before starting

global\_serial\_connection = None

# At the top of the file, after imports

direct\_test\_completed = False # Add this flag

def direct\_serial\_test():

"""Test direct serial connection like in test.py and keep it open if successful"""

global global\_serial\_connection, direct\_test\_completed

# Only run the test once

if direct\_test\_completed:

print("Direct serial test already completed, reusing connection")

return global\_serial\_connection is not None

print("\n🔬 Testing direct serial connection to COM12...")

# First ensure port is released

comprehensive\_port\_release('COM12')

try:

# Use EXACTLY the same code as your working test.py

ser = serial.Serial(

port='COM12',

baudrate=115200,

bytesize=serial.EIGHTBITS,

parity=serial.PARITY\_NONE,

stopbits=serial.STOPBITS\_ONE,

timeout=1

)

print("✅ Connected to COM12")

# Try to read some data

print("📡 Reading data...")

data\_received = False

for \_ in range(10):

line = ser.readline().decode('utf-8', errors='ignore').strip()

if line:

print(f"📊 Received: {line}")

data\_received = True

time.sleep(0.1)

if data\_received:

print("✅ Successfully received data from COM12")

# IMPORTANT: Don't close the connection, save it for later use

global\_serial\_connection = ser

direct\_test\_completed = True # Mark test as completed

return True

else:

print("⚠️ Connected but no data received")

ser.close()

direct\_test\_completed = True # Mark test as completed

return False

except Exception as e:

print(f"❌ Direct serial test failed: {e}")

direct\_test\_completed = True # Mark test as completed

return False

logging.basicConfig(level=logging.INFO,

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

def global\_error\_handler(type, value, traceback):

print(f"❌ Uncaught error: {value}")

cleanup\_resources()

sys.\_\_excepthook\_\_(type, value, traceback)

sys.excepthook = global\_error\_handler

def ensure\_port\_available(port='COM12'):

"""Ensure the port is available before attempting connection"""

print(f"🧹 Ensuring {port} is available...")

# First try to close any existing connections

try:

import serial

temp\_ser = serial.Serial(port)

temp\_ser.close()

print(f"✅ Closed existing connection to {port}")

except:

pass

# Kill any Python processes that might be using serial ports

try:

import psutil

import os

current\_pid = os.getpid()

killed = False

for proc in psutil.process\_iter(['pid', 'name']):

try:

if proc.pid != current\_pid and proc.name().lower() in ['python.exe', 'pythonw.exe']:

print(f"🔫 Terminating {proc.name()} (PID: {proc.pid})")

proc.kill()

killed = True

except:

pass

if killed:

time.sleep(2) # Wait for processes to terminate

except Exception as e:

print(f"⚠️ Process cleanup warning: {e}")

# Use mode command to reset port

try:

import os

os.system(f"mode {port} BAUD=115200 PARITY=N DATA=8 STOP=1")

print(f"✅ Reset {port} settings")

except:

pass

time.sleep(1) # Give OS time to release

return True

def comprehensive\_port\_release(port='COM12'):

"""Comprehensive approach to release a COM port"""

print(f"🔓 Attempting comprehensive release of {port}...")

# 1. Try to close any existing connections

try:

import serial

temp\_ser = serial.Serial(port)

temp\_ser.close()

print("✅ Closed existing connection")

except:

pass

# 2. Reset port using mode command

try:

import os

os.system(f"mode {port} BAUD=115200 PARITY=N DATA=8 STOP=1")

print("✅ Reset port settings")

except:

pass

# 3. Kill potential blocking processes

try:

import psutil

import os

current\_pid = os.getpid()

for proc in psutil.process\_iter(['pid', 'name']):

try:

if proc.pid != current\_pid and proc.name().lower() in [

'python.exe', 'pythonw.exe', 'reachview.exe',

'putty.exe', 'terraterm.exe'

]:

print(f"🔫 Terminating {proc.name()} (PID: {proc.pid})")

proc.kill()

except:

pass

print("✅ Cleaned up processes")

except:

pass

# 4. Wait for OS to fully release the port

import time

time.sleep(3)

print("✅ Port release complete")

return True

def robust\_emlid\_connection(port='COM12'):

"""More robust Emlid connection method"""

print(f"🔌 Attempting robust connection to {port}...")

# First force release the port

comprehensive\_port\_release(port)

# Try with direct file access first

try:

import serial

import time

# Open with exclusive access and minimal settings

ser = serial.Serial(

port=port,

baudrate=115200,

timeout=1,

write\_timeout=1,

exclusive=True

)

# Test if we can read data

print("📡 Testing data reception...")

time.sleep(1)

if ser.in\_waiting > 0:

data = ser.read(ser.in\_waiting)

print(f"✅ Received {len(data)} bytes")

ser.close()

time.sleep(1)

return True

ser.close()

time.sleep(1)

except Exception as e:

print(f"❌ Connection error: {e}")

return False

# Call this before attempting to connect

def is\_port\_permanently\_blocked(port='COM12'):

"""

Performs comprehensive checks to determine if a port is permanently blocked

and cannot be accessed programmatically.

Returns:

tuple: (blocked, reason)

"""

import serial

import subprocess

import os

print(f"🔍 Checking if {port} is permanently blocked...")

# Check 1: Basic port existence

try:

ports = list(serial.tools.list\_ports.comports())

port\_exists = any(p.device == port for p in ports)

if not port\_exists:

return True, f"{port} does not exist in the system"

except:

pass

# Check 2: Try with different access modes

for mode in ['r', 'r+', 'w', 'w+']:

try:

# Try to open the port as a file (low-level)

with open(f"\\\\.\\{port}", mode) as f:

return False, "Port can be accessed at file level"

except:

pass

# Check 3: Try with different serial settings

for baudrate in [9600, 115200, 57600]:

for timeout in [0.1, 1.0]:

try:

ser = serial.Serial(port, baudrate=baudrate, timeout=timeout)

ser.close()

return False, "Port can be accessed with serial library"

except:

pass

# Check 4: Check if it's a system-reserved port

try:

result = subprocess.run(["powershell", "-Command",

f"Get-WmiObject Win32\_SerialPort | Where-Object {{$\_.DeviceID -eq '{port}'}} | Select-Object Name,PNPDeviceID"],

capture\_output=True, text=True)

if "System" in result.stdout:

return True, f"{port} is reserved by the system"

except:

pass

return True, f"{port} appears to be permanently blocked"

# Add after the imports

def display\_gps\_info(emlid\_data):

"""Display current GPS information"""

print("\n=== GPS Status ===")

print(f"Fix Type: {emlid\_data.get('solution\_status', 'Unknown')}")

print(f"Satellites: {emlid\_data.get('satellites', 0)}")

print(f"HDOP: {emlid\_data.get('hdop', 0.0):.2f}")

print(f"Position: {emlid\_data.get('latitude', 0.0):.8f}°N, "

f"{emlid\_data.get('longitude', 0.0):.8f}°E")

print("================\n")

def degrees\_to\_cardinal\_16(degrees):

"""Convert degrees to 16-point compass direction"""

if degrees == 'N/A' or degrees is None:

return 'N/A'

# Normalize degrees to 0-360

degrees = degrees % 360

# 16-point compass with 22.5 degree intervals - full names

directions = [

'North', 'North-Northeast', 'Northeast', 'East-Northeast',

'East', 'East-Southeast', 'Southeast', 'South-Southeast',

'South', 'South-Southwest', 'Southwest', 'West-Southwest',

'West', 'West-Northwest', 'Northwest', 'North-Northwest'

]

# Calculate index (each direction covers 22.5 degrees)

index = int((degrees + 11.25) / 22.5) % 16

return directions[index]

def display\_gps\_status():

"""Display GPS status, current position, and navigation information every 100 milliseconds"""

from datetime import datetime

import csv

import time

import os

# Create CSV header if needed - inline header creation

headers = [

'Timestamp',

'GPS\_Fix\_Quality',

'Satellites\_Visible',

'Satellites\_Used',

'HDOP',

'PDOP',

'VDOP',

'Latitude',

'Longitude',

'Altitude\_m',

'Lat\_Error\_m',

'Lon\_Error\_m',

'Alt\_Error\_m',

'UTM\_X',

'UTM\_Y',

'Heading\_Degrees',

'Direction\_16\_Point'

]

# Check if file exists and has content

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

with open(csv\_file, 'w', newline='') as csvfile:

gps\_writer = csv.writer(csvfile)

gps\_writer.writerow(headers)

print(f"Created CSV header in {csv\_file}")

log\_interval = 0.1 # Log every 0.1 second

last\_log\_time = time.time()

while True:

try:

current\_time = time.time()

# Always log to CSV every 0.1 seconds, even if no GPS connection

if current\_time - last\_log\_time >= log\_interval:

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

gps\_writer = csv.writer(csvfile)

timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]

# Check if rover and GPS are available

if rover and hasattr(rover, 'gps\_reader') and rover.gps\_reader:

position = rover.gps\_reader.last\_position

if position:

gps\_fix = position.get('fix\_quality', 'Unknown')

satellites = position.get('satellites', 0)

satellites\_used = len(position.get('satellites\_used', []))

hdop = position.get('hdop', 99.9)

pdop = position.get('pdop', 99.9)

vdop = position.get('vdop', 99.9)

latitude = position.get('latitude', 0.0)

longitude = position.get('longitude', 0.0)

altitude = position.get('altitude', 0.0)

lat\_error = position.get('lat\_error', 0.0)

lon\_error = position.get('lon\_error', 0.0)

alt\_error = position.get('alt\_error', 0.0)

utm\_x = rover.x

utm\_y = rover.y

heading = position.get('heading', 'N/A')

direction\_16 = degrees\_to\_cardinal\_16(heading)

else:

# GPS reader exists but no position data

gps\_fix = 'No Fix'

satellites = 0

satellites\_used = 0

hdop = 'N/A'

pdop = 'N/A'

vdop = 'N/A'

latitude = 'N/A'

longitude = 'N/A'

altitude = 'N/A'

lat\_error = 'N/A'

lon\_error = 'N/A'

alt\_error = 'N/A'

utm\_x = 'N/A'

utm\_y = 'N/A'

heading = 'N/A'

direction\_16 = 'N/A'

else:

# No rover or GPS reader available

gps\_fix = 'No Connection'

satellites = 'N/A'

satellites\_used = 'N/A'

hdop = 'N/A'

pdop = 'N/A'

vdop = 'N/A'

latitude = 'N/A'

longitude = 'N/A'

altitude = 'N/A'

lat\_error = 'N/A'

lon\_error = 'N/A'

alt\_error = 'N/A'

utm\_x = 'N/A'

utm\_y = 'N/A'

heading = 'N/A'

direction\_16 = 'N/A'

# Write row to CSV

gps\_writer.writerow([

timestamp, gps\_fix, satellites, satellites\_used, hdop, pdop, vdop,

latitude, longitude, altitude, lat\_error, lon\_error, alt\_error,

utm\_x, utm\_y, heading, direction\_16

])

# Debug message - show more detailed info

status\_msg = 'Data logged'

if rover and hasattr(rover, 'gps\_reader') and rover.gps\_reader:

if rover.gps\_reader.last\_position:

pos = rover.gps\_reader.last\_position

fix\_quality = pos.get('fix\_quality', 'Unknown')

sats\_visible = pos.get('satellites', 0)

sats\_used = len(pos.get('satellites\_used', []))

pdop\_val = pos.get('pdop', 99.9)

vdop\_val = pos.get('vdop', 99.9)

status\_msg += f' (GPS: {fix\_quality}, Sats: {sats\_visible}/{sats\_used}, PDOP: {pdop\_val:.1f}, VDOP: {vdop\_val:.1f})'

else:

status\_msg += ' (GPS No Fix)'

else:

status\_msg += ' (GPS Disconnected)'

print(f"CSV Log [{timestamp}]: {status\_msg}")

last\_log\_time = current\_time

# Enhanced real-time display with better formatting

print(f"\n=== ROVER STATUS [{datetime.now().strftime('%H:%M:%S.%f')[:-3]}] ===")

if rover and hasattr(rover, 'gps\_reader') and rover.gps\_reader:

position = rover.gps\_reader.last\_position

if position:

fix\_quality = position.get('fix\_quality', 'Unknown')

satellites\_visible = position.get('satellites', 0)

satellites\_used\_list = position.get('satellites\_used', [])

satellites\_used\_count = len(satellites\_used\_list)

hdop = position.get('hdop', 99.9)

pdop = position.get('pdop', 99.9)

vdop = position.get('vdop', 99.9)

# Color coding for fix quality

if fix\_quality == 'RTK Fixed':

fix\_display = f"🟢 {fix\_quality}"

elif fix\_quality == 'RTK Float':

fix\_display = f"🟡 {fix\_quality}"

elif fix\_quality in ['GPS', 'DGPS']:

fix\_display = f"🔴 {fix\_quality}"

else:

fix\_display = f"⚫ {fix\_quality}"

print(f"GPS Fix: {fix\_display}")

print(f"Satellites Visible: {satellites\_visible}")

print(f"Satellites Used: {satellites\_used\_count}")

if satellites\_used\_list:

print(f"Satellite PRNs: {satellites\_used\_list}")

# DOP values with quality indicators

print(f"HDOP: {hdop:.2f} {'🟢' if hdop < 2.0 else '🟡' if hdop < 5.0 else '🔴'}")

print(f"PDOP: {pdop:.2f} {'🟢' if pdop < 3.0 else '🟡' if pdop < 6.0 else '🔴'}")

print(f"VDOP: {vdop:.2f} {'🟢' if vdop < 3.0 else '🟡' if vdop < 6.0 else '🔴'}")

print(f"Position (Lat/Lon): {position.get('latitude', 0.0):.8f}, {position.get('longitude', 0.0):.8f}")

print(f"Altitude: {position.get('altitude', 0.0):.3f}m")

# Position errors

lat\_err = position.get('lat\_error', 0.0)

lon\_err = position.get('lon\_error', 0.0)

alt\_err = position.get('alt\_error', 0.0)

print(f"Position Errors - Lat: {lat\_err:.3f}m, Lon: {lon\_err:.3f}m, Alt: {alt\_err:.3f}m")

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

if 'heading' in position and position['heading'] != 'N/A':

heading = position['heading']

direction\_16 = degrees\_to\_cardinal\_16(heading)

print(f"Heading: {heading:.1f}° ({direction\_16})")

else:

print("Heading: N/A")

# Additional RTK-specific info

if 'mode' in position:

print(f"GPS Mode: {position['mode']} (A=Auto, M=Manual)")

if 'fix\_type' in position:

fix\_type\_map = {1: 'No Fix', 2: '2D Fix', 3: '3D Fix'}

print(f"Fix Type: {fix\_type\_map.get(position['fix\_type'], 'Unknown')}")

else:

print("GPS Fix: No Fix Available")

print("Satellites Visible: 0")

print("Satellites Used: 0")

print("HDOP: N/A")

print("PDOP: N/A")

print("VDOP: N/A")

print("Position (Lat/Lon): N/A")

print("Position Errors: N/A")

print("Current Position (UTM): N/A")

print("Heading: N/A")

else:

print("GPS Status: DISCONNECTED")

print("GPS Fix: No Connection")

print("Satellites Visible: N/A")

print("Satellites Used: N/A")

print("HDOP: N/A")

print("PDOP: N/A")

print("VDOP: N/A")

print("Position (Lat/Lon): N/A")

print("Position Errors: N/A")

print("Current Position (UTM): N/A")

print("Heading: N/A")

# Navigation information

if rover and hasattr(rover, 'navigator') and rover.navigator and rover.navigator.interpolated\_path:

if rover.navigator.current\_waypoint\_index < len(rover.navigator.interpolated\_path):

next\_wp = rover.navigator.interpolated\_path[rover.navigator.current\_waypoint\_index]

heading = rover.navigator.calculate\_heading((rover.x, rover.y), next\_wp)

distance = rover.distance\_to(next\_wp[0], next\_wp[1])

direction\_16 = degrees\_to\_cardinal\_16(heading)

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

print(f"Heading to Waypoint: {heading:.1f}° ({direction\_16})")

print(f"Distance to Waypoint: {distance:.3f}m")

else:

print("Navigation complete")

else:

print("No navigation path set")

print("====================\n")

time.sleep(0.1) # Update every 100 milliseconds

except Exception as e:

print(f"Status display error: {e}")

# Still log error state to CSV

try:

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

gps\_writer = csv.writer(csvfile)

timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]

gps\_writer.writerow([

timestamp, 'ERROR', 'N/A', 'N/A', 'N/A', 'N/A', 'N/A',

'N/A', 'N/A', 'N/A', 'N/A', 'N/A', 'N/A',

'N/A', 'N/A', 'N/A', 'N/A'

])

except:

pass

time.sleep(0.1)

# Add after the imports section

rover = None

ntrip\_client = None

failsafe = None

gps\_thread = None

def check\_gps\_status():

"""Check GPS quality and connection status"""

global rover

if not rover or not hasattr(rover, 'gps\_reader'):

return False

position = rover.gps\_reader.last\_position

if not position:

return False

satellites = position.get('satellites', 0)

hdop = position.get('hdop', 99.9)

fix\_type = position.get('solution\_status', 'Unknown')

# Log GPS status

print(f"\nGPS Status:")

print(f"Fix Type: {fix\_type}")

print(f"Satellites: {satellites}")

print(f"HDOP: {hdop:.1f}")

# Check minimum requirements

return (satellites >= 4 and

hdop < 5.0 and

fix\_type in ['GPS', 'DGPS', 'RTK Fixed', 'RTK Float'])

def gps\_reading\_loop():

"""GPS data reading loop for real or simulated data"""

while True:

try:

if hasattr(rover, 'gps\_reader') and rover.gps\_reader: # ADDED CHECK

if rover.gps\_reader.simulate\_gps:

fake = simulate\_emlid\_gps\_reading()

logging\_100mm.update\_rover\_position\_from\_emlid(rover, fake)

display\_gps\_info(fake)

else:

# Real GPS data

if rover.gps\_reader.last\_position:

display\_gps\_info(rover.gps\_reader.last\_position)

time.sleep(1.0)

except Exception as e:

print(f"GPS reading error: {e}")

time.sleep(1.0)

def setup\_ntrip(rover, emlid\_reader):

"""Setup NTRIP connection and corrections"""

try:

NTRIP\_CONFIG = {

'host': 'your.ntrip.server.url',

'port': 2101,

'mountpoint': 'MOUNTPOINT',

'user': 'your\_username',

'password': 'your\_password'

}

ntrip\_client = NTRIPClient(\*\*NTRIP\_CONFIG)

# Test NTRIP connection

if not ntrip\_client.connect():

raise Exception("Failed to connect to NTRIP server")

# Start RTCM corrections thread

def stream\_rtcm():

for rtcm\_data in ntrip\_client.get\_corrections():

if hasattr(rover, 'gps\_reader'):

rover.gps\_reader.send\_rtcm\_data(rtcm\_data)

ntrip\_thread = threading.Thread(target=stream\_rtcm, daemon=True)

ntrip\_thread.start()

return True

except Exception as e:

print(f"NTRIP setup failed: {e}")

return False

def setup\_gps\_with\_optional\_ntrip(rover, use\_ntrip=False):

"""Setup GPS with optional NTRIP - SIMPLIFIED VERSION based on working test script"""

try:

print("\n📡 Setting up Emlid GPS integration...")

# Initialize Emlid reader with minimal configuration

emlid\_reader = EmlidGPSReader(port='COM12', baud\_rate=115200, message\_format='nmea')

emlid\_reader.simulate\_gps = False

# Register callback before connection

update\_rover\_from\_emlid(rover, emlid\_reader)

print("✅ Callback registered for Emlid GPS data")

# Try the simple connection approach first (like your test script)

print("\n🔌 Attempting to connect to Emlid GPS...")

if hasattr(emlid\_reader, 'connect\_with\_simple\_approach'):

connection\_success = emlid\_reader.connect\_with\_simple\_approach()

else:

# Fallback to regular connect if the new method isn't available

connection\_success = emlid\_reader.connect()

if connection\_success:

print(f"✅ Connected to Emlid M2 on COM12")

# Start reading with immediate verification

if emlid\_reader.start\_reading():

rover.gps\_reader = emlid\_reader

print("📡 GPS data reading started")

# Verify data reception

print("🔍 Verifying data reception...")

for \_ in range(3): # Check 3 times

time.sleep(1)

if emlid\_reader.last\_position:

print("✅ Live GPS data confirmed")

return True

print("⚠️ No position data detected yet, but connection is active")

return True

else:

print("❌ Failed to start GPS reading")

else:

print("❌ Connection failed")

# If we get here, connection failed - clean up

try:

if hasattr(emlid\_reader, 'stop\_reading'):

emlid\_reader.stop\_reading()

if hasattr(emlid\_reader, 'disconnect'):

emlid\_reader.disconnect()

except:

pass

# Simulation fallback

print("\n📡 GPS Connection Failed")

print("Options:")

print("1) Continue with GPS simulation")

print("2) Abort")

while True:

try:

choice = input("Select option (1/2): ").strip()

if choice == '1':

print("\n🧪 Initializing GPS simulation...")

emlid\_reader = EmlidGPSReader(port='COM12', message\_format='nmea')

emlid\_reader.simulate\_gps = True

rover.gps\_reader = emlid\_reader

update\_rover\_from\_emlid(rover, emlid\_reader)

def simulation\_thread():

while True:

try:

fake\_data = simulate\_emlid\_gps\_reading()

for callback in emlid\_reader.callbacks:

try:

callback(fake\_data)

except Exception as e:

print(f"Callback error: {e}")

time.sleep(0.1)

except Exception as e:

print(f"Simulation error: {e}")

time.sleep(1)

sim\_thread = threading.Thread(target=simulation\_thread, daemon=True)

sim\_thread.start()

print("✅ GPS simulation active")

return False

elif choice == '2':

print("\n❌ Setup aborted by user")

return False

else:

print("Invalid choice - enter 1 or 2")

except Exception as input\_e:

print(f"Input error: {input\_e}")

except Exception as e:

print(f"\n❌ GPS setup error: {e}")

import traceback

traceback.print\_exc()

return False

debug = False

safety = SafetyModule()

def setup\_gps\_direct\_approach(rover):

"""Setup GPS using the already open serial connection"""

global global\_serial\_connection

print("\n📡 Setting up GPS with direct serial approach...")

# Check if we already have an open connection

if not global\_serial\_connection or not global\_serial\_connection.is\_open:

print("❌ No open serial connection available")

return False

try:

# Create a minimal EmlidGPSReader that just uses our working connection

emlid\_reader = EmlidGPSReader(port='COM12', baud\_rate=115200)

emlid\_reader.serial\_connection = global\_serial\_connection

emlid\_reader.simulate\_gps = False

# Register callback

update\_rover\_from\_emlid(rover, emlid\_reader)

# Create a custom reading thread

def custom\_reading\_thread():

error\_count = 0

max\_errors = 10

satellites\_in\_view = {} # Use dict to avoid duplicates, key = PRN

satellites\_used = [] # Track satellites used in solution

constellation\_stats = {} # Track stats per constellation

while True:

try:

if not global\_serial\_connection:

print("❌ No serial connection available")

time.sleep(1)

continue

if not global\_serial\_connection.is\_open:

print("❌ Serial connection is closed, attempting to reopen")

try:

global\_serial\_connection.open()

print("✅ Reopened serial connection")

except Exception as open\_err:

print(f"❌ Failed to reopen connection: {open\_err}")

time.sleep(1)

continue

# Read NMEA data directly

try:

line = global\_serial\_connection.readline().decode('utf-8', errors='ignore').strip()

if line:

current\_time = time.time()

if not hasattr(custom\_reading\_thread, 'last\_print\_time') or current\_time - custom\_reading\_thread.last\_print\_time >= 2.0:

print(f"📡 NMEA: {line}")

custom\_reading\_thread.last\_print\_time = current\_time

# Parse ALL GSV messages - ENHANCED to catch more constellations

if 'GSV' in line and line.startswith('$'):

parts = line.split(',')

if len(parts) >= 4:

try:

constellation = line[:6] # e.g., $GPGSV, $GNGSV, $GLGSV, etc.

total\_messages = int(parts[1]) if parts[1] else 0

message\_number = int(parts[2]) if parts[2] else 0

total\_sats\_reported = int(parts[3]) if parts[3] else 0

# Initialize constellation tracking

if constellation not in constellation\_stats:

constellation\_stats[constellation] = {

'expected\_messages': total\_messages,

'received\_messages': 0,

'satellites\_count': 0

}

# If this is message 1, reset the constellation data

if message\_number == 1:

constellation\_stats[constellation] = {

'expected\_messages': total\_messages,

'received\_messages': 0,

'satellites\_count': total\_sats\_reported

}

# Remove old satellites from this constellation

constellation\_prefixes = {

'$GPGSV': range(1, 33), # GPS: PRN 1-32

'$GLGSV': range(65, 97), # GLONASS: PRN 65-96 (sometimes 1-24)

'$GAGSV': range(301, 337), # Galileo: PRN 301-336 (sometimes 1-36)

'$GBGSV': range(401, 438), # BeiDou: PRN 401-437 (sometimes 1-37)

'$GQGSV': range(201, 237), # QZSS: PRN 201-237 (sometimes 1-10)

'$GIGSV': range(501, 537), # IRNSS: PRN 501-537

'$GNGSV': range(1, 600) # Mixed: could be any

}

# More flexible PRN removal - remove by constellation type

if constellation in constellation\_prefixes:

prn\_range = constellation\_prefixes[constellation]

# Remove satellites that might belong to this constellation

satellites\_in\_view = {k: v for k, v in satellites\_in\_view.items()

if not (v.get('constellation') == constellation)}

# Parse satellite info (up to 4 satellites per GSV message)

satellites\_in\_this\_message = 0

for i in range(4, min(len(parts), 20), 4):

if i + 3 < len(parts):

sat\_prn = parts[i].strip() if parts[i] else None

elevation = parts[i + 1].strip() if parts[i + 1] else None

azimuth = parts[i + 2].strip() if parts[i + 2] else None

snr = parts[i + 3].split('\*')[0].strip() if parts[i + 3] else None

if sat\_prn and sat\_prn != '':

try:

prn = int(sat\_prn)

# Create unique key combining constellation and PRN

sat\_key = f"{constellation}\_{prn}"

sat\_info = {

'prn': prn,

'constellation': constellation,

'elevation': int(elevation) if elevation and elevation != '' else 0,

'azimuth': int(azimuth) if azimuth and azimuth != '' else 0,

'snr': int(snr) if snr and snr != '' else 0,

'key': sat\_key

}

satellites\_in\_view[sat\_key] = sat\_info

satellites\_in\_this\_message += 1

except ValueError as ve:

print(f"⚠️ Error parsing satellite PRN '{sat\_prn}': {ve}")

# Update message tracking

constellation\_stats[constellation]['received\_messages'] = message\_number

# Update position data with total satellite count

if emlid\_reader.last\_position is None:

emlid\_reader.last\_position = {}

total\_sats\_visible = len(satellites\_in\_view)

emlid\_reader.last\_position['satellites'] = total\_sats\_visible

emlid\_reader.last\_position['satellites\_in\_view'] = list(satellites\_in\_view.values())

# Detailed logging every few seconds

if not hasattr(custom\_reading\_thread, 'last\_detailed\_log') or current\_time - custom\_reading\_thread.last\_detailed\_log >= 3.0:

print(f"🛰️ {constellation}: Msg {message\_number}/{total\_messages}, Reported: {total\_sats\_reported}, This msg: {satellites\_in\_this\_message}")

print(f"🛰️ Total satellites visible across all constellations: {total\_sats\_visible}")

# Show breakdown by constellation

constellation\_breakdown = {}

for sat\_key, sat\_info in satellites\_in\_view.items():

const = sat\_info['constellation']

if const not in constellation\_breakdown:

constellation\_breakdown[const] = 0

constellation\_breakdown[const] += 1

for const, count in constellation\_breakdown.items():

print(f" {const}: {count} satellites")

custom\_reading\_thread.last\_detailed\_log = current\_time

except Exception as gsv\_e:

print(f"GSV parsing error for {line}: {gsv\_e}")

# Parse $GPGGA or $GNGGA for position data

elif line.startswith(('$GPGGA', '$GNGGA')):

parts = line.split(',')

if len(parts) >= 15 and parts[6] != '0' and parts[2] and parts[4]:

try:

# Convert DDMM.MMMM to decimal degrees

lat\_raw = parts[2]

lon\_raw = parts[4]

lat\_deg = int(lat\_raw[:2])

lat\_min = float(lat\_raw[2:])

latitude = lat\_deg + lat\_min / 60.0

lon\_deg = int(lon\_raw[:3])

lon\_min = float(lon\_raw[3:])

longitude = lon\_deg + lon\_min / 60.0

if parts[3] == 'S':

latitude = -latitude

if parts[5] == 'W':

longitude = -longitude

# Map fix quality to proper RTK status

fix\_quality\_map = {

'0': 'Invalid',

'1': 'GPS',

'2': 'DGPS',

'3': 'PPS',

'4': 'RTK Fixed',

'5': 'RTK Float',

'6': 'Estimated',

'7': 'Manual',

'8': 'Simulation'

}

position = {

'latitude': latitude,

'longitude': longitude,

'altitude': float(parts[9]) if parts[9] else 0.0,

'satellites': len(satellites\_in\_view), # Total satellites visible

'hdop': float(parts[8]) if parts[8] else 99.9,

'fix\_quality': fix\_quality\_map.get(parts[6], 'Unknown'),

'solution\_status': fix\_quality\_map.get(parts[6], 'Unknown')

}

if emlid\_reader.last\_position is None:

emlid\_reader.last\_position = {}

emlid\_reader.last\_position.update(position)

emlid\_reader.last\_update\_time = time.time()

for callback in emlid\_reader.callbacks:

callback(emlid\_reader.last\_position)

except Exception as parse\_e:

print(f"NMEA parsing error: {parse\_e}")

# Parse $GPRMC or $GNRMC for heading

elif line.startswith(('$GPRMC', '$GNRMC')):

parts = line.split(',')

if len(parts) >= 10 and parts[2] == 'A':

try:

cog = float(parts[8]) if parts[8] else 0.0

if emlid\_reader.last\_position is None:

emlid\_reader.last\_position = {}

emlid\_reader.last\_position['heading'] = cog

emlid\_reader.last\_update\_time = time.time()

for callback in emlid\_reader.callbacks:

callback(emlid\_reader.last\_position)

except Exception as parse\_e:

print(f"$GPRMC parsing error: {parse\_e}")

# Parse ALL GSA messages for satellites used and DOP

elif 'GSA' in line and line.startswith('$'):

parts = line.split(',')

if len(parts) >= 18:

try:

constellation\_gsa = line[:6] # e.g., $GPGSA, $GNGSA, $GLGSA

mode = parts[1] # A = Auto, M = Manual

fix\_type = int(parts[2]) if parts[2] else 0 # 1=No fix, 2=2D, 3=3D

# Extract satellites used (positions 3-14)

current\_satellites\_used = []

for i in range(3, 15): # Positions 3-14 contain satellite PRNs

if i < len(parts) and parts[i] and parts[i].strip():

try:

sat\_prn = int(parts[i].strip())

current\_satellites\_used.append(sat\_prn)

except ValueError:

pass

# For GNGSA (multi-constellation), this gives us the total used

if constellation\_gsa == '$GNGSA':

satellites\_used = current\_satellites\_used

else:

# For single constellation GSA, add to the list

for sat in current\_satellites\_used:

if sat not in satellites\_used:

satellites\_used.append(sat)

# Extract DOP values

pdop = float(parts[15]) if len(parts) >= 16 and parts[15] else 99.9

hdop = float(parts[16]) if len(parts) >= 17 and parts[16] else 99.9

vdop\_str = parts[17].split('\*')[0] if len(parts) >= 18 and parts[17] else '99.9'

vdop = float(vdop\_str) if vdop\_str else 99.9

if emlid\_reader.last\_position is None:

emlid\_reader.last\_position = {}

emlid\_reader.last\_position.update({

'mode': mode,

'fix\_type': fix\_type,

'satellites\_used': satellites\_used,

'pdop': pdop,

'hdop': hdop,

'vdop': vdop

})

# Validation and detailed logging

sats\_visible = len(satellites\_in\_view)

sats\_used = len(satellites\_used)

if sats\_visible < sats\_used:

print(f"⚠️ Warning: Satellites used ({sats\_used}) > visible ({sats\_visible})")

# Enhanced logging for RTK analysis

if not hasattr(custom\_reading\_thread, 'last\_rtk\_log') or current\_time - custom\_reading\_thread.last\_rtk\_log >= 5.0:

fix\_quality = emlid\_reader.last\_position.get('fix\_quality', 'Unknown')

print(f"🛰️ RTK Status: {fix\_quality}")

print(f"🛰️ GSA ({constellation\_gsa}): Mode={mode}, Fix={fix\_type}")

print(f"🛰️ Satellites: Visible={sats\_visible}, Used={sats\_used}")

print(f"🛰️ DOP: PDOP={pdop:.1f}, HDOP={hdop:.1f}, VDOP={vdop:.1f}")

# RTK quality assessment

if fix\_quality == 'RTK Fixed':

print("🟢 Excellent RTK Fixed solution")

elif fix\_quality == 'RTK Float':

print("🟡 Good RTK Float solution")

if sats\_used < 8:

print(" 💡 Consider: More satellites could help achieve RTK Fixed")

elif fix\_quality in ['GPS', 'DGPS']:

print("🔴 Basic GPS solution - RTK corrections may not be working")

print(" 💡 Check NTRIP connection and base station distance")

# Satellite usage analysis

if sats\_visible >= 20 and sats\_used < 10:

print(f" 💡 Many satellites visible ({sats\_visible}) but few used ({sats\_used})")

print(" 💡 This is normal - receiver selects best satellites for solution")

elif sats\_used >= 12:

print(f" ✅ Good satellite usage: {sats\_used} satellites")

custom\_reading\_thread.last\_rtk\_log = current\_time

for callback in emlid\_reader.callbacks:

callback(emlid\_reader.last\_position)

except Exception as parse\_e:

print(f"$GSA parsing error: {parse\_e}")

# Parse $GPGST or $GNGST for position errors

elif line.startswith(('$GPGST', '$GNGST')):

parts = line.split(',')

if len(parts) >= 9:

try:

lat\_error = float(parts[6]) if parts[6] else 0.0

lon\_error = float(parts[7]) if parts[7] else 0.0

alt\_error\_str = parts[8].split('\*')[0] if parts[8] else '0.0'

alt\_error = float(alt\_error\_str) if alt\_error\_str else 0.0

if emlid\_reader.last\_position is None:

emlid\_reader.last\_position = {}

emlid\_reader.last\_position.update({

'lat\_error': lat\_error,

'lon\_error': lon\_error,

'alt\_error': alt\_error

})

for callback in emlid\_reader.callbacks:

callback(emlid\_reader.last\_position)

except Exception as parse\_e:

print(f"$GST parsing error: {parse\_e}")

error\_count = 0

except Exception as read\_err:

error\_count += 1

print(f"Read error ({error\_count}/{max\_errors}): {read\_err}")

if error\_count >= max\_errors:

print("Too many read errors, resetting connection")

try:

global\_serial\_connection.close()

time.sleep(1)

global\_serial\_connection.open()

print("Connection reset complete")

except:

pass

error\_count = 0

time.sleep(0.5)

continue

time.sleep(0.05) # Faster polling for more data

except Exception as e:

print(f"Reading thread error: {e}")

error\_count += 1

if error\_count >= max\_errors:

print("Too many errors in reading thread, resetting")

error\_count = 0

time.sleep(1.0)

# Start our custom reading thread

reading\_thread = threading.Thread(target=custom\_reading\_thread, daemon=True)

reading\_thread.start()

emlid\_reader.reading\_thread = reading\_thread

# Assign to rover

rover.gps\_reader = emlid\_reader

print("✅ GPS setup complete with direct approach")

return True

except Exception as e:

print(f"❌ Direct GPS setup failed: {e}")

return False

def enhanced\_gps\_status\_monitor():

"""Enhanced GPS status monitor with health checks"""

global rover

last\_health\_report = 0

while True:

try:

if rover and hasattr(rover, 'gps\_reader') and rover.gps\_reader: # ADDED CHECK

current\_time = time.time()

# Get health status

health = rover.gps\_reader.check\_health()

# Print detailed status every 10 seconds

if current\_time - last\_health\_report > 10:

print(f"\n=== GPS Health Report [{datetime.datetime.now().strftime('%H:%M:%S')}] ===")

print(f"Connected: {'✅' if health['connected'] else '❌'}")

print(f"Thread Alive: {'✅' if health['thread\_alive'] else '❌'}")

print(f"Simulation Mode: {'🧪' if health['simulation\_mode'] else '📡'}")

print(f"Last Update: {health['last\_update']:.1f}s ago")

if rover.gps\_reader.last\_position:

pos = rover.gps\_reader.last\_position

print(f"Fix Quality: {pos.get('fix\_quality', 'Unknown')}")

print(f"Satellites: {pos.get('satellites', 0)}")

print(f"HDOP: {pos.get('hdop', 99.9):.2f}")

# Quality indicators

if pos.get('fix\_quality') == 'RTK Fixed':

print("🟢 Excellent RTK Fixed")

elif pos.get('fix\_quality') == 'RTK Float':

print("🟡 Good RTK Float")

elif pos.get('fix\_quality') in ['GPS', 'DGPS']:

print("🔴 Basic GPS Fix")

else:

print("⚫ Poor/No Fix")

print("=" \* 50)

last\_health\_report = current\_time

# Check for problems

if health['last\_update'] > 10: # No data for 10 seconds

print("⚠️ WARNING: No GPS data received for 10+ seconds")

if not health['connected'] and not health['simulation\_mode']:

print("⚠️ WARNING: GPS connection lost, attempting recovery...")

try:

rover.gps\_reader.connect(retries=2, retry\_delay=1)

except:

pass

time.sleep(5) # Update every 5 seconds

except Exception as e:

print(f"GPS monitor error: {e}")

time.sleep(5)

def get\_float(prompt):

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

while True:

try:

value = float(input(prompt))

return value

except ValueError:

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

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

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

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

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

return x, y

def safe\_remove(element):

if element:

try:

element.remove()

return True

except:

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

return False

def process\_emlid\_gps\_data(rover, emlid\_data):

"""

Process GPS data from Emlid receiver and update rover position.

This is a standalone version that doesn't rely on class methods.

Args:

rover: The rover instance

emlid\_data: Dictionary with GPS data from Emlid receiver

Returns:

bool: True if position was updated successfully, False otherwise

"""

if not emlid\_data or 'latitude' not in emlid\_data or 'longitude' not in emlid\_data:

print("⚠️ Invalid or missing Emlid GPS data")

return False

try:

# Get the coordinate converter

converter = None

if hasattr(rover, 'coordinate\_converter'):

converter = rover.coordinate\_converter

elif hasattr(rover, 'gps\_logger') and hasattr(rover.gps\_logger, 'converter'):

converter = rover.gps\_logger.converter

else:

print("⚠️ No coordinate converter found")

return False

# Convert lat/lon to UTM

easting, northing = converter.latlon\_to\_utm\_coord(

emlid\_data['latitude'],

emlid\_data['longitude']

)

if easting is None or northing is None:

print("⚠️ Failed to convert lat/lon to UTM")

return False

# Get the correct UTM offsets

utm\_offset\_x = 0

utm\_offset\_y = 0

if hasattr(rover, 'navigator') and hasattr(rover.navigator, 'utm\_offset\_x'):

utm\_offset\_x = rover.navigator.utm\_offset\_x

utm\_offset\_y = rover.navigator.utm\_offset\_y

# Calculate the local coordinates by removing the offsets

local\_x = easting - utm\_offset\_x

local\_y = northing - utm\_offset\_y

# Update rover position

rover.set\_position(local\_x, local\_y)

# Log the update if a logger is available

if hasattr(rover, 'gps\_logger'):

if hasattr(rover.gps\_logger, 'log\_data\_once'):

rover.gps\_logger.log\_data\_once()

return True

except Exception as e:

print(f"Error processing Emlid GPS data: {e}")

return False

def display\_ntrip\_status(ntrip\_client):

print("\n=== NTRIP Status ===")

print(f"Connected: {ntrip\_client.connected}")

print("===================\n")

def handle\_gps\_error(error\_message):

"""Handle GPS errors without disconnecting"""

print(f"⚠️ GPS Error: {error\_message}")

print("Attempting recovery...")

# Just reset the failsafe timers without disconnecting

if hasattr(rover, 'failsafe'):

rover.failsafe.last\_gps\_update = time.time()

if hasattr(rover.failsafe, 'last\_correction\_update'):

rover.failsafe.last\_correction\_update = time.time()

return True # Indicate successful recovery

def setup\_gps\_simple\_approach(rover):

"""Setup GPS using the exact same approach as test.py"""

print("\n📡 Setting up Emlid GPS with simple approach...")

# First ensure all existing connections are closed

try:

from emlid\_gps\_integration import cleanup\_all\_gps\_connections

cleanup\_all\_gps\_connections()

time.sleep(2) # Give OS time to release port

except Exception as e:

print(f"Cleanup error: {e}")

# Force release the port

force\_release\_com\_port('COM12')

# Check if port is permanently blocked

blocked, reason = is\_port\_permanently\_blocked('COM12')

if blocked:

print(f"❌ COM12 is permanently blocked: {reason}")

print(" Cannot proceed with real GPS connection")

return False

try:

# Create a simple serial connection like in test.py

ser = serial.Serial(

port='COM12',

baudrate=115200,

bytesize=serial.EIGHTBITS,

parity=serial.PARITY\_NONE,

stopbits=serial.STOPBITS\_ONE,

timeout=1

)

print("✅ Connected to COM12")

# Initialize the EmlidGPSReader with this connection

emlid\_reader = EmlidGPSReader(port='COM12', baud\_rate=115200)

emlid\_reader.serial\_connection = ser # Use our working connection

# Register callback

update\_rover\_from\_emlid(rover, emlid\_reader)

# Start reading thread

emlid\_reader.start\_reading()

# Assign to rover

rover.gps\_reader = emlid\_reader

return True

except Exception as e:

print(f"❌ Simple GPS setup failed: {e}")

return False

csv\_dir = r'F:\GPS\task\_2\_waypoints'

csv\_file = os.path.join(csv\_dir, 'gps\_status\_log.csv')

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

os.makedirs(csv\_dir)

print(f"✅ Created directory: {csv\_dir}")

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

with open(csv\_file, 'w', newline='') as csvfile:

gps\_writer = csv.writer(csvfile)

gps\_writer.writerow([

'Timestamp', 'GPS Fix', 'Satellites', 'Satellites Used', 'HDOP', 'PDOP', 'VDOP',

'Latitude', 'Longitude', 'Altitude', 'Lat Error', 'Lon Error', 'Alt Error',

'UTM X', 'UTM Y', 'Heading'

])

def run\_simulation():

global rover, ntrip\_client, failsafe, global\_serial\_connection

print("🧹 Initial cleanup of any existing GPS connections...")

try:

from emlid\_gps\_integration import cleanup\_all\_gps\_connections

cleanup\_all\_gps\_connections()

except Exception as e:

print(f"Initial cleanup error: {e}")

direct\_serial\_test()

# Initialize gps\_success variable with a default value

gps\_success = False

# Only run the direct serial test if it hasn't been run yet

if not direct\_test\_completed:

direct\_serial\_test()

def on\_failsafe\_triggered(reason):

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

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

# Update the on\_recovery\_attempt function

def on\_recovery\_attempt(reason):

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

current\_time = time.time()

try:

if reason == GPSFailsafeReason.GPS\_STALE\_DATA or reason == GPSFailsafeReason.GPS\_DATA\_LOSS:

failsafe.last\_gps\_update = current\_time

# Don't disconnect/reconnect, just reset the timer

print("Resetting GPS data timer without disconnecting")

return True

elif reason == GPSFailsafeReason.GPS\_CORRECTION\_STALE:

failsafe.last\_correction\_update = current\_time

print("Resetting GPS correction timer without disconnecting")

return True

elif reason == GPSFailsafeReason.INTERNET\_CONNECTION\_LOST or reason == GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW:

failsafe.last\_internet\_check = current\_time

return True # Continue without internet

elif reason == GPSFailsafeReason.MODULE\_COMMUNICATION\_FAILURE:

failsafe.last\_module\_comm = current\_time

return True

return True

except Exception as e:

print(f"Recovery attempt failed: {e}")

return False

# Add periodic NTRIP status check

print("🚜 Farm Rover Navigation Simulation 🚜")

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

sys.modules['\_\_main\_\_'].global\_serial\_connection = global\_serial\_connection

# -------------------- HEALTH CHECK SECTION --------------------

print("\n🔍 Running rover health checks before simulation...")

# Create the rover first (needed by the health checker)

rover = Rover()

# Initialize the coordinate converter

coordinate\_converter = CoordinateConverter()

rover.coordinate\_converter = coordinate\_converter

# Initialize health checker with the rover instance

health\_checker = RoverHealthCheck(rover)

try:

# Run all health checks

health\_status = health\_checker.run\_all\_checks(simulation\_mode=True)

# Generate and display health report

health\_report = health\_checker.generate\_health\_report()

print(health\_report)

# Check if all systems passed

if not all(health\_status.values()):

print("\n⚠️ One or more health checks failed. Aborting simulation.")

print(" Please address the issues and try again.")

return

print("\n✅ All health checks passed! Proceeding with simulation.")

except HealthCheckFailure as e:

print(f"\n❌ Critical health check failure: {e}")

print(" Simulation cannot proceed until this issue is resolved.")

return

# -------------------- END HEALTH CHECK SECTION --------------------

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

failsafe = FailsafeModule()

safety = SafetyModule(failsafe=failsafe)

failsafe.set\_safety\_module(safety)

# Initialize failsafe first

rover.failsafe = failsafe

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

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

failsafe.update\_module\_communication()

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

# Now initialize GPS logger

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

# Inside run\_simulation function, replace the GPS setup section with:

# Enhanced GPS setup with comprehensive error handling

print("\n📡 Setting up Emlid GPS integration...")

# Initialize gps\_success variable with a default value

gps\_success = False

# Try our direct approach using the already open connection

gps\_success = setup\_gps\_direct\_approach(rover)

status\_thread = threading.Thread(target=display\_gps\_status, daemon=True)

status\_thread.start()

# If that fails, fall back to simulation

if not gps\_success:

print("\n⚠️ GPS setup failed. Switching to simulation mode...")

try:

# Clean up any partial setup

if hasattr(rover, 'gps\_reader') and rover.gps\_reader:

rover.gps\_reader.stop\_reading()

rover.gps\_reader.disconnect()

# Setup simulation

emlid\_reader = EmlidGPSReader(port='COM12', message\_format='nmea')

emlid\_reader.simulate\_gps = True

rover.gps\_reader = emlid\_reader

# Register callback for simulation

update\_rover\_from\_emlid(rover, emlid\_reader)

def gps\_simulation\_loop():

while True:

try:

fake = simulate\_emlid\_gps\_reading()

for callback in emlid\_reader.callbacks:

try:

callback(fake)

except Exception as cb\_e:

print(f"Simulation callback error: {cb\_e}")

time.sleep(0.1) # 10Hz simulation

except Exception as sim\_e:

print(f"Simulation error: {sim\_e}")

time.sleep(1.0)

threading.Thread(target=gps\_simulation\_loop, daemon=True).start()

print("🧪 GPS simulation started successfully")

except Exception as sim\_error:

print(f"❌ Even simulation setup failed: {sim\_error}")

print(" This is a critical error - check your imports and dependencies")

# Create row navigator

navigator = RowNavigator(rover)

rover.navigator = navigator

# Start failsafe monitoring

failsafe.start\_monitoring()

navigator.zigzag\_pattern = True

# Load waypoints from CSV file

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

if not csv\_loaded:

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

return

# Calculate farm boundaries based on waypoints with margin

margin = 3.0 # Add margin around waypoints

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

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

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

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

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

# Create vertices for the farm boundary

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

# Generate a random entry point

side = random.randint(0, 3)

if side == 0: # Bottom side

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

entry\_y = min\_y

elif side == 1: # Right side

entry\_x = max\_x

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

elif side == 2: # Top side

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

entry\_y = max\_y

else: # Left side

entry\_x = min\_x

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

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

# Set geofence in rover and safety module

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

safety.set\_geofence(verts)

# Generate random starting position inside the farm

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

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

# Initialize visualization

plt.ion()

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

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

# Draw farm boundary

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

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

ax.add\_patch(farm\_polygon)

# Mark random start position

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

# Setup rover path visualization

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

ax.path\_line = path\_line

ax.legend(loc='upper left')

# Set rover starting position (inside farm)

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

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

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

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

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

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

# Ensure zigzag pattern is enabled

navigator.zigzag\_pattern = True

# Use the waypoints previously loaded from CSV

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

# Determine plot boundaries based on waypoints

if navigator.interpolated\_path:

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

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

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

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

# Use the wider range between farm boundaries and waypoints

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

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

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

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

# Add a larger margin

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

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

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

else:

# Fallback to original farm boundaries

margin = 3

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

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

ax.grid(True)

# Visualize zigzag row pattern

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

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

# Mark start and end points

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

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

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

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

fig.canvas.draw\_idle()

plt.pause(0.5)

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

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

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

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

def on\_rover\_wakeup():

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

# Do whatever you need when rover wakes up

# Navigate to path start

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

"""

Navigate rover to the starting point of the path using direct point-to-point moves

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

"""

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

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

rover,

path\_start[0],

path\_start[1],

ax,

fig,

rover\_patch,

step\_size=1.5, # larger increments per move

tolerance=0.8 # accept slightly further from the exact point

)

return reached\_start, rover\_patch

# Use our custom function to navigate to path start

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

if not reached\_start:

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

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

return

#Force rover position to exactly match path start

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

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

# Mark path start reached

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

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

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

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

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

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

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

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

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

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

# Align to the path direction

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

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

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

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

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

# Start navigation from the beginning of the path

navigator.current\_waypoint\_index = 0

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

if not path\_success:

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

return

# Mark completion of path

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

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

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

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

print("\n🏁 SIMULATION COMPLETE! 🏁")

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

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

# Keep plot open until closed manually

plt.ioff()

plt.show(block=True)

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

failsafe.stop\_monitoring()

cleanup\_resources()

def cleanup\_resources():

"""Enhanced cleanup with better error handling"""

global rover, ntrip\_client, failsafe, gps\_thread, global\_serial\_connection

print("\nCleaning up resources...")

try:

if global\_serial\_connection and global\_serial\_connection.is\_open:

try:

global\_serial\_connection.close()

print("✅ Closed global serial connection")

except Exception as e:

print(f"⚠️ Error closing global serial connection: {e}")

global\_serial\_connection = None

# FIRST: Stop all GPS-related threads and connections

if rover and hasattr(rover, 'gps\_reader') and rover.gps\_reader: # ADDED CHECK

print("Stopping GPS reader...")

try:

rover.gps\_reader.stop\_reading() # Stop reading first

time.sleep(1) # Give it time

rover.gps\_reader.disconnect() # Then disconnect

time.sleep(1) # Give OS time to release port

print("✅ GPS disconnected successfully")

except Exception as e:

print(f"⚠️ GPS cleanup error: {e}")

# Cleanup all GPS connections (using our new function)

try:

from emlid\_gps\_integration import cleanup\_all\_gps\_connections

cleanup\_all\_gps\_connections()

except Exception as e:

print(f"⚠️ Global GPS cleanup error: {e}")

if ntrip\_client:

print("Cleaning up NTRIP...")

try:

ntrip\_client.cleanup()

except Exception as e:

print(f"⚠️ NTRIP cleanup error: {e}")

ntrip\_client = None

if rover:

print("Stopping GPS logger...")

try:

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

except Exception as e:

print(f"⚠️ Logger cleanup error: {e}")

if failsafe:

print("Stopping failsafe monitoring...")

try:

failsafe.stop\_monitoring()

except Exception as e:

print(f"⚠️ Failsafe cleanup error: {e}")

try:

plt.close('all')

except Exception as e:

print(f"⚠️ Plot cleanup error: {e}")

print("✅ Cleanup complete")

except Exception as e:

print(f"❌ Cleanup error: {e}")

def simulate\_emlid\_gps\_reading():

"""

Simulate an Emlid GPS reading for testing purposes.

Returns a dictionary with lat/lon coordinates and RTK status.

"""

# These are example coordinates - in a real implementation,

# you would get these from the Emlid GPS receiver

# Randomly choose a solution status for demonstration

solution\_statuses = ["fixed", "float", "single", "dgps"]

solution\_status = random.choice(solution\_statuses)

# Determine appropriate HDOP based on solution status

if solution\_status == "fixed":

hdop = random.uniform(0.01, 0.2)

elif solution\_status == "float":

hdop = random.uniform(0.2, 0.5)

elif solution\_status == "dgps":

hdop = random.uniform(0.5, 1.0)

else: # single

hdop = random.uniform(1.0, 2.0)

return {

'latitude': 28.6139, # Example latitude

'longitude': 77.2090, # Example longitude

'solution\_status': solution\_status, # RTK solution status from Emlid

'satellites': random.randint(8, 15), # Number of satellites

'hdop': hdop, # Horizontal dilution of precision

'fix\_quality': solution\_status.upper(), # Add fix\_quality

'altitude': random.uniform(200, 300), # Add altitude

'speed': random.uniform(0, 5), # Add speed

'heading': random.uniform(0, 360) # Add heading

}

def gps\_status\_monitor():

"""Monitor GPS status and quality"""

global rover

while True:

try:

if rover and hasattr(rover, 'gps\_reader'):

if rover.gps\_reader.last\_position:

fix\_type = rover.gps\_reader.last\_position.get('solution\_status', 'Unknown')

satellites = rover.gps\_reader.last\_position.get('satellites', 0)

hdop = rover.gps\_reader.last\_position.get('hdop', 0.0)

if fix\_type == 'fixed':

print("🟢 RTK Fixed")

elif fix\_type == 'float':

print("🟡 RTK Float")

else:

print("🔴 No RTK")

print(f"Satellites: {satellites}, HDOP: {hdop:.2f}")

time.sleep(5) # Update every 5 seconds

except Exception as e:

print(f"GPS status monitor error: {e}")

time.sleep(1)

def test\_emlid\_integration():

"""

Test function to verify Emlid GPS integration with the rover system.

"""

print("🧪 Testing Emlid GPS integration...")

# Create rover instance

rover = Rover()

# Initialize coordinate converter

converter = CoordinateConverter()

rover.coordinate\_converter = converter

# Create row navigator (needed for UTM offsets)

navigator = RowNavigator(rover)

rover.navigator = navigator

# Set default UTM offsets for testing

navigator.utm\_offset\_x = 380000.0

navigator.utm\_offset\_y = 2044880.0

# Initialize GPS logger

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

# Simulate Emlid GPS reading

emlid\_data = simulate\_emlid\_gps\_reading()

print(f"📡 Simulated Emlid GPS reading: Lat={emlid\_data['latitude']}, Lon={emlid\_data['longitude']}")

# Use the built-in function from the logging\_100mm module

success = logging\_100mm.update\_rover\_position\_from\_emlid(rover, emlid\_data)

# Check the result

if success:

print("✅ Successfully processed Emlid GPS data")

print(f"🚜 Rover position (UTM): X={rover.x:.3f}, Y={rover.y:.3f}")

# Calculate the actual global UTM coordinates

actual\_easting = rover.x + rover.navigator.utm\_offset\_x

actual\_northing = rover.y + rover.navigator.utm\_offset\_y

# Convert back to lat/lon for verification

lat, lon = converter.utm\_to\_latlon\_coord(

actual\_easting, actual\_northing,

zone\_number=43, zone\_letter='N' # Make sure to use the correct zone

)

print(f"🌐 Rover position (Lat/Lon): {lat:.6f}, {lon:.6f}")

# Calculate difference from original coordinates

original\_lat = emlid\_data['latitude']

original\_lon = emlid\_data['longitude']

lat\_diff = abs(lat - original\_lat)

lon\_diff = abs(lon - original\_lon)

print(f"📊 Conversion difference: Lat={lat\_diff:.8f}, Lon={lon\_diff:.8f}")

if lat\_diff < 0.0001 and lon\_diff < 0.0001:

print("✅ Conversion accuracy check passed")

else:

print("❌ Conversion accuracy check failed - differences too large")

else:

print("❌ Failed to process Emlid GPS data")

# Cleanup

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

print("🧪 Test completed")

def force\_release\_com\_port(port='COM12'):

"""Force release a COM port using mode command"""

print(f"\n🔧 Forcing release of {port}...")

try:

# Try closing any existing connections

try:

temp\_ser = serial.Serial(port)

temp\_ser.close()

except:

pass

# Use mode command to reset port

os.system(f"mode {port} BAUD=115200 PARITY=N DATA=8 STOP=1")

time.sleep(2) # Give OS time to release

# Kill any Python processes that might be using serial ports

import psutil

current\_pid = os.getpid()

for proc in psutil.process\_iter(['pid', 'name']):

try:

if proc.pid != current\_pid and proc.name().lower() in ['python.exe', 'pythonw.exe']:

proc.kill()

except:

pass

time.sleep(2) # Wait for processes to terminate

print("✅ Port cleanup complete")

return True

except Exception as e:

print(f"⚠️ Port cleanup warning: {e}")

return False

def test\_com12\_availability():

"""Test if COM12 is available"""

print("🧪 Testing COM12 availability...")

try:

ser = serial.Serial(

port='COM12',

baudrate=115200,

timeout=1,

bytesize=serial.EIGHTBITS,

parity=serial.PARITY\_NONE,

stopbits=serial.STOPBITS\_ONE

)

print("✅ COM12 is available and opened successfully")

ser.close()

except serial.SerialException as e:

print(f"❌ Failed to open COM12: {e}")

if "Access is denied" in str(e):

print(" 🔍 COM12 is likely in use by another application or process.")

print(" Please close other programs (e.g., ReachView, PuTTY) and try again.")

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

try:

print("🚜 Starting Farm Simulation...")

# Force initial cleanup

try:

from emlid\_gps\_integration import cleanup\_all\_gps\_connections

cleanup\_all\_gps\_connections()

comprehensive\_port\_release('COM12')

time.sleep(2) # Give OS time to release ports

print("✅ Initial cleanup complete")

except Exception as e:

print(f"⚠️ Cleanup warning: {e}")

# First, try direct serial connection like in test.py

real\_gps = direct\_serial\_test()

# Run simulation in simulation mode by default

print("\n" + "=" \* 50)

if not real\_gps:

# Set simulation flag in GPS module

import emlid\_gps\_integration

emlid\_gps\_integration.simulate\_gps = True

print("🧪 Running in simulation mode")

run\_simulation()

except KeyboardInterrupt:

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

except Exception as e:

print(f"\n❌ Simulation error: {e}")

import traceback

traceback.print\_exc()

finally:

cleanup\_resources()