import time

import threading

import datetime

import logging

from enum import Enum

import math

# Set up logging

logging.basicConfig(

level=logging.INFO,

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

handlers=[

logging.FileHandler(r"F:\GPS\task\_2\_waypoints\rover\_failsafe.csv", encoding='utf-8'),

logging.StreamHandler()

]

)

logger = logging.getLogger("rover\_failsafe")

class GPSFailsafeReason(Enum):

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

GPS\_DATA\_LOSS = "GPS data loss"

GPS\_STALE\_DATA = "GPS stale data"

INTERNET\_CONNECTION\_LOST = "Internet connection lost"

INTERNET\_CONNECTION\_SLOW = "Internet connection slow"

OVER\_TEMPERATURE = "Over temperature"

MODULE\_COMMUNICATION\_FAILURE = "Module communication failure"

SIGNAL\_INSTABILITY = "Signal instability"

SIGNIFICANT\_DRIFT = "Significant navigation drift"

PERSISTENT\_DRIFT = "Persistent navigation drift"

CUSTOM = "Custom reason"

GPS\_POSITION\_JUMP = "GPS position jump detected"

GPS\_HEADING\_INCONSISTENCY = "GPS heading inconsistency"

GPS\_ALTITUDE\_ANOMALY = "GPS altitude anomaly"

GPS\_VELOCITY\_CONSTRAINT = "GPS velocity constraint violation"

GPS\_HIGH\_DOP = "GPS high dilution of precision"

GPS\_SIGNAL\_MULTIPATH = "GPS signal multipath detected"

GPS\_CONSTELLATION\_WEAK = "GPS constellation geometry weak"

class DriftSeverity(Enum):

"""Enumeration of drift severity levels"""

MINOR = "Minor drift"

MODERATE = "Moderate drift"

SEVERE = "Severe drift"

CRITICAL = "Critical drift"

class DriftAction(Enum):

"""Enumeration of actions to take for drift"""

MONITOR = "Continue monitoring"

REALIGN = "Perform realignment"

SLOW\_DOWN = "Reduce speed"

PAUSE = "Pause movement"

FAILSAFE = "Enter failsafe mode"

class FailsafeModule:

"""

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

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

safety rather than navigation-specific safety.

"""

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

# Status flags

self.in\_failsafe\_mode = False

self.failsafe\_reason = None

self.recovery\_attempts = 0

self.recovery\_in\_progress = False

# Timestamp tracking

self.last\_gps\_update = None

self.last\_internet\_check = None

self.last\_module\_comm = None

self.last\_recovery\_attempt = None

# Thresholds and settings

self.gps\_stale\_threshold = 30.0 # seconds

self.internet\_timeout = 10.0 # seconds

self.module\_comm\_timeout = 15.0 # INCREASED: from 5.0 to 15.0 seconds

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

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

self.temperature\_threshold = 60.0 # degrees Celsius

# Signal tracking

self.signal\_loss\_events = []

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

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

# Drift monitoring

self.drift\_events = []

self.drift\_window = 300.0 # 5 minute window for drift events

self.drift\_threshold\_minor = 0.2 # meters

self.drift\_threshold\_moderate = 0.5 # meters

self.drift\_threshold\_severe = 1.0 # meters

self.drift\_threshold\_critical = 2.0 # meters

self.drift\_count\_threshold = 3 # Number of severe drift events to trigger failsafe

self.drift\_persistence\_threshold = 10 # Number of consecutive drift events to consider persistent

self.consecutive\_drift\_events = 0

self.last\_realignment = None

self.realignment\_cooldown = 60.0 # seconds between realignments

self.current\_drift\_action = DriftAction.MONITOR

# Callbacks

self.drift\_action\_callback = None

# Initialize monitoring threads

self.monitoring\_active = False

self.monitor\_thread = None

# Callback for failsafe activation

self.failsafe\_callback = None

self.recovery\_callback = None

# Initialize timestamps to prevent immediate failsafe triggers

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

self.last\_module\_comm = time.time() # ADDED: Initialize module communication timestamp

self.last\_internet\_check = time.time() # ADDED: Initialize internet check timestamp

# GPS position tracking

self.previous\_gps\_positions = [] # List of recent positions (lat, lon, alt, timestamp)

self.position\_history\_max = 10 # Maximum number of positions to store

self.max\_position\_jump = 5.0 # Maximum allowed position jump in meters

self.max\_speed = 2.0 # Maximum expected speed in m/s

# GPS heading tracking

self.previous\_headings = [] # List of recent headings (heading, timestamp)

self.heading\_history\_max = 5 # Maximum number of headings to store

self.max\_heading\_change = 45.0 # Maximum allowed heading change in degrees per second

# GPS altitude tracking

self.previous\_altitudes = [] # List of recent altitudes (altitude, timestamp)

self.altitude\_history\_max = 5 # Maximum number of altitudes to store

self.max\_altitude\_change = 1.0 # Maximum allowed altitude change in meters per second

# GPS DOP thresholds

self.hdop\_threshold = 5.0 # HDOP threshold for good precision

self.pdop\_threshold = 7.0 # PDOP threshold for good precision

self.vdop\_threshold = 7.0 # VDOP threshold for good precision

# GPS constellation settings

self.min\_satellites = 6 # Minimum satellites for reliable operation

self.min\_satellite\_snr = 30 # Minimum signal-to-noise ratio (dB-Hz)

# GPS velocity constraints

self.max\_acceleration = 1.0 # Maximum expected acceleration in m/s²

# Multipath detection

self.multipath\_snr\_threshold = 10 # SNR drop threshold for multipath detection

self.previous\_snr\_values = {} # Dictionary to store previous SNR values by satellite ID

# Initialize velocity tracking

self.previous\_velocities = []

self.velocity\_history\_max = 5

# Initialize satellite SNR tracking for multipath detection

if not hasattr(self, 'previous\_snr\_values'):

self.previous\_snr\_values = {}

logger.info("Failsafe module initialized")

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

"""

Set callbacks for failsafe activation and recovery

Args:

failsafe\_callback: Function to call when failsafe is triggered

Should accept FailsafeReason as an argument

recovery\_callback: Function to call when attempting recovery

Should accept FailsafeReason as an argument

drift\_action\_callback: Function to call when drift action changes

Should accept DriftAction and drift\_distance as arguments

"""

self.failsafe\_callback = failsafe\_callback

self.recovery\_callback = recovery\_callback

self.drift\_action\_callback = drift\_action\_callback

logger.info("Failsafe callbacks set")

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

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

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

drift\_thresholds=None, drift\_count\_threshold=None,

drift\_persistence\_threshold=None, realignment\_cooldown=None,

module\_comm\_timeout=None): # ADDED: module\_comm\_timeout parameter

"""Configure failsafe thresholds and settings"""

if gps\_stale\_threshold is not None:

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

if internet\_timeout is not None:

self.internet\_timeout = internet\_timeout

if module\_comm\_timeout is not None: # ADDED: Configure module communication timeout

self.module\_comm\_timeout = module\_comm\_timeout

if recovery\_interval is not None:

self.recovery\_interval = recovery\_interval

if max\_recovery\_attempts is not None:

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

if temperature\_threshold is not None:

self.temperature\_threshold = temperature\_threshold

if signal\_loss\_threshold is not None:

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

if drift\_thresholds is not None:

if 'minor' in drift\_thresholds:

self.drift\_threshold\_minor = drift\_thresholds['minor']

if 'moderate' in drift\_thresholds:

self.drift\_threshold\_moderate = drift\_thresholds['moderate']

if 'severe' in drift\_thresholds:

self.drift\_threshold\_severe = drift\_thresholds['severe']

if 'critical' in drift\_thresholds:

self.drift\_threshold\_critical = drift\_thresholds['critical']

if drift\_count\_threshold is not None:

self.drift\_count\_threshold = drift\_count\_threshold

if drift\_persistence\_threshold is not None:

self.drift\_persistence\_threshold = drift\_persistence\_threshold

if realignment\_cooldown is not None:

self.realignment\_cooldown = realignment\_cooldown

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

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

f"Module comm timeout={self.module\_comm\_timeout}s, " # ADDED: Log module comm timeout

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

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

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

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

def start\_monitoring(self):

"""Start the failsafe monitoring thread"""

if self.monitoring\_active:

logger.warning("Monitoring already active")

return

# Reset timestamps to prevent immediate failsafe trigger

current\_time = time.time()

self.last\_gps\_update = current\_time

self.last\_module\_comm = current\_time # ADDED: Reset module timestamp

self.last\_internet\_check = current\_time # ADDED: Reset internet timestamp

self.monitoring\_active = True

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

self.monitor\_thread.start()

logger.info("Failsafe monitoring started")

def stop\_monitoring(self):

"""Stop the failsafe monitoring thread"""

if not self.monitoring\_active:

logger.warning("Monitoring not active")

return

self.monitoring\_active = False

if self.monitor\_thread:

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

logger.info("Failsafe monitoring stopped")

def \_monitor\_loop(self):

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

while self.monitoring\_active:

try:

# Check various failsafe conditions

self.\_check\_gps\_status()

self.\_check\_internet\_connection()

self.\_check\_module\_communication()

self.\_check\_temperature()

self.\_check\_signal\_stability()

self.\_check\_drift\_status()

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

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

(self.last\_recovery\_attempt is None or

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

self.\_attempt\_recovery()

time.sleep(1.0) # Check conditions every second

except Exception as e:

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

def \_check\_gps\_status(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any GPS data

if self.last\_gps\_update is None:

# Log a warning instead of immediately triggering failsafe

logger.warning("No GPS data available yet")

# Initialize last\_gps\_update to prevent immediate failsafe

self.last\_gps\_update = current\_time

return

# Check if GPS data is stale

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

# Log a warning before triggering failsafe

logger.warning(f"GPS data stale: Last update was {current\_time - self.last\_gps\_update:.1f}s ago")

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

def \_check\_internet\_connection(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any internet connectivity data

if self.last\_internet\_check is None:

# ADDED: Initialize internet check to prevent immediate failsafe

logger.warning("No internet connection data available yet")

self.last\_internet\_check = current\_time

return

# Check if internet connection is lost or too slow

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

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Internet connection lost: Last check was {current\_time - self.last\_internet\_check:.1f}s ago")

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

def \_check\_module\_communication(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any module communication data

if self.last\_module\_comm is None:

# ADDED: Initialize module communication to prevent immediate failsafe

logger.warning("No module communication data available yet")

self.last\_module\_comm = current\_time

return

# Module communication timeout (increased from 5 to 15 seconds)

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

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Module communication failure: Last communication was {current\_time - self.last\_module\_comm:.1f}s ago")

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

def \_check\_temperature(self):

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

if self.in\_failsafe\_mode:

return

# This is a placeholder for actual temperature checking

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

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

if temperature > self.temperature\_threshold:

# ADDED: Log a warning before triggering failsafe

logger.warning(f"System temperature too high: {temperature:.1f}°C > {self.temperature\_threshold:.1f}°C")

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

def \_check\_signal\_stability(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Remove signal loss events older than the window

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

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

# Check if we have too many signal loss events

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

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Signal instability detected: {len(self.signal\_loss\_events)} losses in the last {self.signal\_loss\_window:.1f}s")

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

def \_check\_drift\_status(self):

"""Check if drift has exceeded thresholds and take appropriate action"""

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Remove drift events older than the window

self.drift\_events = [event for event in self.drift\_events

if current\_time - event['timestamp'] <= self.drift\_window]

# Check for severe drift events

severe\_drift\_events = [event for event in self.drift\_events

if event['severity'] in (DriftSeverity.SEVERE, DriftSeverity.CRITICAL)]

# Check if we have too many severe drift events

if len(severe\_drift\_events) >= self.drift\_count\_threshold:

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Significant drift detected: {len(severe\_drift\_events)} severe events in the last {self.drift\_window:.1f}s")

self.\_trigger\_failsafe(GPSFailsafeReason.SIGNIFICANT\_DRIFT)

return

# Check for persistent drift (many consecutive drift events)

if self.consecutive\_drift\_events >= self.drift\_persistence\_threshold:

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Persistent drift detected: {self.consecutive\_drift\_events} consecutive drift events")

self.\_trigger\_failsafe(GPSFailsafeReason.PERSISTENT\_DRIFT)

return

# If we have any action other than monitoring in place, check if it's time to clear it

if self.current\_drift\_action != DriftAction.MONITOR and not self.drift\_events:

self.\_set\_drift\_action(DriftAction.MONITOR, 0)

def \_trigger\_failsafe(self, reason):

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

if self.in\_failsafe\_mode:

return

print("Sleep mode activated, going into sleep mode")

self.in\_failsafe\_mode = True

self.failsafe\_reason = reason

self.recovery\_attempts = 0

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

# Execute failsafe callback if provided

if self.failsafe\_callback:

try:

self.failsafe\_callback(reason)

except Exception as e:

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

def \_attempt\_recovery(self):

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

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

return

self.recovery\_in\_progress = True

self.recovery\_attempts += 1

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

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

# Execute recovery callback if provided

recovery\_success = False # ADDED: Initialize success flag

if self.recovery\_callback:

try:

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

if recovery\_success:

self.\_clear\_failsafe()

except Exception as e:

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

recovery\_success = False # ADDED: Set success to false on exception

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

if self.recovery\_attempts >= self.max\_recovery\_attempts and not recovery\_success:

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

# Keep in failsafe mode but allow future recovery attempts

self.recovery\_in\_progress = False

def \_clear\_failsafe(self):

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

if not self.in\_failsafe\_mode:

return

old\_reason = self.failsafe\_reason

self.in\_failsafe\_mode = False

self.failsafe\_reason = None

self.recovery\_attempts = 0

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

# ADDED: Reset timestamps to prevent immediate re-triggering of failsafe

current\_time = time.time()

self.last\_gps\_update = current\_time

self.last\_module\_comm = current\_time

self.last\_internet\_check = current\_time

def \_determine\_drift\_severity(self, drift\_distance):

"""

Determine the severity of a drift event based on distance

Args:

drift\_distance: Distance in meters

Returns:

DriftSeverity: Severity level of the drift

"""

if drift\_distance >= self.drift\_threshold\_critical:

return DriftSeverity.CRITICAL

elif drift\_distance >= self.drift\_threshold\_severe:

return DriftSeverity.SEVERE

elif drift\_distance >= self.drift\_threshold\_moderate:

return DriftSeverity.MODERATE

elif drift\_distance >= self.drift\_threshold\_minor:

return DriftSeverity.MINOR

else:

return None # No significant drift

def \_determine\_drift\_action(self, severity, consecutive\_count):

"""

Determine what action to take based on drift severity and consecutive count

Args:

severity: DriftSeverity level

consecutive\_count: Number of consecutive drift events

Returns:

DriftAction: Action to take

"""

if severity == DriftSeverity.CRITICAL:

return DriftAction.FAILSAFE

if severity == DriftSeverity.SEVERE:

if consecutive\_count >= self.drift\_persistence\_threshold:

return DriftAction.FAILSAFE

elif consecutive\_count >= self.drift\_persistence\_threshold // 2:

return DriftAction.PAUSE

else:

return DriftAction.REALIGN

if severity == DriftSeverity.MODERATE:

if consecutive\_count >= self.drift\_persistence\_threshold:

return DriftAction.PAUSE

elif consecutive\_count >= 3:

return DriftAction.REALIGN

else:

return DriftAction.SLOW\_DOWN

if severity == DriftSeverity.MINOR:

if consecutive\_count >= self.drift\_persistence\_threshold:

return DriftAction.SLOW\_DOWN

else:

return DriftAction.MONITOR

return DriftAction.MONITOR

def \_set\_drift\_action(self, action, drift\_distance):

"""

Set and execute a drift action

Args:

action: DriftAction to take

drift\_distance: Current drift distance in meters

"""

# Don't change action if we're in failsafe mode

if self.in\_failsafe\_mode:

return

# Don't downgrade action severity (e.g., from PAUSE to SLOW\_DOWN)

action\_severity = {

DriftAction.MONITOR: 0,

DriftAction.SLOW\_DOWN: 1,

DriftAction.REALIGN: 2,

DriftAction.PAUSE: 3,

DriftAction.FAILSAFE: 4

}

# Only update if the new action is more severe or we're returning to monitoring

if (action == DriftAction.MONITOR or

action\_severity[action] > action\_severity[self.current\_drift\_action]):

# Log the action change

if action != self.current\_drift\_action:

logger.info(f"Drift action changed: {action.value} (drift: {drift\_distance:.2f}m)")

# Update the current action

self.current\_drift\_action = action

# Handle realignment cooldown

current\_time = time.time()

if (action == DriftAction.REALIGN and self.last\_realignment and

current\_time - self.last\_realignment < self.realignment\_cooldown):

logger.info(f"Skipping realignment - cooldown period active")

return

# If it's a realignment, update the timestamp

if action == DriftAction.REALIGN:

self.last\_realignment = current\_time

# Execute action callback if provided

if self.drift\_action\_callback:

try:

self.drift\_action\_callback(action, drift\_distance)

except Exception as e:

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

# If action is failsafe, trigger it

if action == DriftAction.FAILSAFE:

self.\_trigger\_failsafe(GPSFailsafeReason.SIGNIFICANT\_DRIFT)

# API Methods for rover interaction

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

"""

Update GPS status information

Args:

has\_fix: Whether GPS has a valid fix

satellites: Number of satellites in view

hdop: Horizontal dilution of precision

"""

if not has\_fix or satellites < 6 or hdop > 5.0:

# Register as signal loss event

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

else:

# Log when GPS status is updated

logger.debug(f"GPS status updated: fix={has\_fix}, satellites={satellites}, hdop={hdop}")

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

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

"""

Update internet connection status

Args:

connected: Whether internet is connected

latency: Latency in seconds

"""

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

# Consider high latency as a slow connection

if not connected or latency > 2.0:

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

reason = (GPSFailsafeReason.INTERNET\_CONNECTION\_LOST if not connected

else GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW)

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Internet issue detected: {'disconnected' if not connected else f'high latency ({latency:.2f}s)'}")

self.\_trigger\_failsafe(reason)

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

"""

Update module communication timestamp

Args:

timestamp: Optional timestamp to use, defaults to current time

"""

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

# ADDED: Log when module communication is updated

logger.debug(f"Module communication updated: {self.last\_module\_comm}")

def report\_signal\_loss(self):

"""Report a signal loss event"""

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

# ADDED: Log when signal loss is reported

logger.debug(f"Signal loss reported, total events in window: {len(self.signal\_loss\_events)}")

def report\_drift(self, actual\_position, expected\_position):

"""

Report a position drift event with actual and expected positions

Args:

actual\_position: Tuple of (latitude, longitude) or (x, y) for actual position

expected\_position: Tuple of (latitude, longitude) or (x, y) for expected position

Returns:

Tuple of (drift\_distance, severity)

"""

# Calculate drift distance

if len(actual\_position) == 2 and len(expected\_position) == 2:

# For lat/lon coordinates, we need to use haversine formula

if abs(actual\_position[0]) <= 90 and abs(actual\_position[1]) <= 180:

drift\_distance = self.\_haversine\_distance(

actual\_position[0], actual\_position[1],

expected\_position[0], expected\_position[1]

)

else:

# For x/y coordinates, we can use Euclidean distance

drift\_distance = math.sqrt(

(actual\_position[0] - expected\_position[0])\*\*2 +

(actual\_position[1] - expected\_position[1])\*\*2

)

else:

logger.error(f"Invalid position format: {actual\_position}, {expected\_position}")

return 0, None

# Determine severity

severity = self.\_determine\_drift\_severity(drift\_distance)

# If there's significant drift, record it

if severity is not None:

current\_time = time.time()

self.drift\_events.append({

'timestamp': current\_time,

'distance': drift\_distance,

'severity': severity,

'actual': actual\_position,

'expected': expected\_position

})

# Update consecutive drift counter

self.consecutive\_drift\_events += 1

# Log the drift event

logger.info(f"Drift detected: {drift\_distance:.2f}m, Severity: {severity.value}, "

f"Consecutive: {self.consecutive\_drift\_events}")

# Determine and set action

action = self.\_determine\_drift\_action(severity, self.consecutive\_drift\_events)

self.\_set\_drift\_action(action, drift\_distance)

else:

# Reset consecutive drift counter if no significant drift

self.consecutive\_drift\_events = 0

# If we were taking action, go back to monitoring

if self.current\_drift\_action != DriftAction.MONITOR:

self.\_set\_drift\_action(DriftAction.MONITOR, drift\_distance)

return drift\_distance, severity

def \_haversine\_distance(self, lat1, lon1, lat2, lon2):

"""

Calculate the great circle distance between two points

on the earth (specified in decimal degrees)

"""

# Convert decimal degrees to radians

lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

# Haversine formula

dlon = lon2 - lon1

dlat = lat2 - lat1

a = math.sin(dlat/2)\*\*2 + math.cos(lat1) \* math.cos(lat2) \* math.sin(dlon/2)\*\*2

c = 2 \* math.asin(math.sqrt(a))

r = 6371000 # Radius of earth in meters

return c \* r

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

"""

Manually trigger failsafe mode with a custom reason

Args:

reason\_text: Description of the custom reason

"""

custom\_reason = GPSFailsafeReason.CUSTOM

self.\_trigger\_failsafe(custom\_reason)

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

def force\_recovery(self):

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

if self.in\_failsafe\_mode:

logger.info("Forcing recovery attempt")

self.\_attempt\_recovery()

return True # ADDED: Return success indicator

else:

logger.warning("Cannot force recovery - not in failsafe mode")

return False # ADDED: Return failure indicator

def force\_realignment(self):

"""Force a realignment regardless of conditions"""

logger.info("Forcing realignment")

self.\_set\_drift\_action(DriftAction.REALIGN, 0)

self.last\_realignment = time.time()

return True # ADDED: Return success indicator

def reset\_drift\_tracking(self):

"""Reset all drift tracking data"""

self.drift\_events = []

self.consecutive\_drift\_events = 0

self.current\_drift\_action = DriftAction.MONITOR

logger.info("Drift tracking reset")

return True # ADDED: Return success indicator

def get\_status(self):

"""Get current failsafe status information"""

return {

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

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

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

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

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

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

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

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

"drift\_events": len(self.drift\_events),

"consecutive\_drift\_events": self.consecutive\_drift\_events,

"current\_drift\_action": self.current\_drift\_action.value,

"last\_realignment": self.last\_realignment,

"monitoring\_active": self.monitoring\_active

}

def get\_drift\_statistics(self):

"""Get statistics about drift events"""

if not self.drift\_events:

return {

"count": 0,

"avg\_distance": 0,

"max\_distance": 0,

"min\_distance": 0,

"severity\_counts": {s.name: 0 for s in DriftSeverity}

}

# Calculate statistics

distances = [event['distance'] for event in self.drift\_events]

severity\_counts = {s.name: 0 for s in DriftSeverity}

for event in self.drift\_events:

severity\_counts[event['severity'].name] += 1

return {

"count": len(self.drift\_events),

"avg\_distance": sum(distances) / len(distances),

"max\_distance": max(distances),

"min\_distance": min(distances),

"severity\_counts": severity\_counts

}

def \_get\_system\_temperature(self):

"""

Get the current system temperature

This is a placeholder implementation that should be replaced

with actual hardware temperature readings in a real system

"""

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

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

return 45.0 # Simulated temperature in Celsius

def update\_gps\_position(self, latitude, longitude, altitude=None, timestamp=None):

"""

Update current GPS position and check for anomalies

Args:

latitude: Current latitude in decimal degrees

longitude: Current longitude in decimal degrees

altitude: Current altitude in meters (optional)

timestamp: Timestamp of the position fix (optional, defaults to current time)

Returns:

bool: True if position is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# Update last GPS update time

self.last\_gps\_update = timestamp

# Store new position

new\_position = (latitude, longitude, altitude, timestamp)

self.previous\_gps\_positions.append(new\_position)

# Trim history if needed

if len(self.previous\_gps\_positions) > self.position\_history\_max:

self.previous\_gps\_positions.pop(0)

# Check for position jump if we have at least two positions

if len(self.previous\_gps\_positions) >= 2:

return self.\_check\_position\_jump()

return True

def \_check\_position\_jump(self):

"""

Check if there's a sudden jump in GPS position that exceeds physical limits

Returns:

bool: True if position is valid, False if jump detected

"""

# Get the two most recent positions

current = self.previous\_gps\_positions[-1]

previous = self.previous\_gps\_positions[-2]

# Calculate distance and time difference

distance = self.\_haversine\_distance(

current[0], current[1],

previous[0], previous[1]

)

time\_diff = current[3] - previous[3]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate speed

speed = distance / time\_diff # m/s

# Check if speed exceeds maximum expected speed by a significant margin

if speed > self.max\_speed \* 1.5:

logger.warning(f"GPS position jump detected: {distance:.2f}m in {time\_diff:.2f}s ({speed:.2f} m/s)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_POSITION\_JUMP)

return False

return True

def update\_gps\_heading(self, heading, timestamp=None):

"""

Update current GPS heading and check for inconsistencies

Args:

heading: Current heading in degrees (0-360)

timestamp: Timestamp of the heading (optional, defaults to current time)

Returns:

bool: True if heading is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# Store new heading

new\_heading = (heading, timestamp)

self.previous\_headings.append(new\_heading)

# Trim history if needed

if len(self.previous\_headings) > self.heading\_history\_max:

self.previous\_headings.pop(0)

# Check for heading inconsistency if we have at least two headings

if len(self.previous\_headings) >= 2:

return self.\_check\_heading\_consistency()

return True

def \_check\_heading\_consistency(self):

"""

Check if there's a sudden change in GPS heading that exceeds physical limits

Returns:

bool: True if heading is valid, False if inconsistency detected

"""

# Get the two most recent headings

current = self.previous\_headings[-1]

previous = self.previous\_headings[-2]

# Calculate heading difference

heading\_diff = min(

abs(current[0] - previous[0]),

360 - abs(current[0] - previous[0])

)

time\_diff = current[1] - previous[1]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate rate of heading change

heading\_rate = heading\_diff / time\_diff # degrees/s

# Check if heading change exceeds maximum expected change

if heading\_rate > self.max\_heading\_change:

logger.warning(f"GPS heading inconsistency detected: {heading\_diff:.2f}° in {time\_diff:.2f}s ({heading\_rate:.2f}°/s)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HEADING\_INCONSISTENCY)

return False

return True

def update\_gps\_altitude(self, altitude, timestamp=None):

"""

Update current GPS altitude and check for anomalies

Args:

altitude: Current altitude in meters

timestamp: Timestamp of the altitude (optional, defaults to current time)

Returns:

bool: True if altitude is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# Store new altitude

new\_altitude = (altitude, timestamp)

self.previous\_altitudes.append(new\_altitude)

# Trim history if needed

if len(self.previous\_altitudes) > self.altitude\_history\_max:

self.previous\_altitudes.pop(0)

# Check for altitude anomaly if we have at least two altitudes

if len(self.previous\_altitudes) >= 2:

return self.\_check\_altitude\_anomaly()

return True

def \_check\_altitude\_anomaly(self):

"""

Check if there's a sudden change in GPS altitude that exceeds physical limits

Returns:

bool: True if altitude is valid, False if anomaly detected

"""

# Get the two most recent altitudes

current = self.previous\_altitudes[-1]

previous = self.previous\_altitudes[-2]

# Calculate altitude difference

altitude\_diff = abs(current[0] - previous[0])

time\_diff = current[1] - previous[1]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate rate of altitude change

altitude\_rate = altitude\_diff / time\_diff # m/s

# Check if altitude change exceeds maximum expected change

if altitude\_rate > self.max\_altitude\_change:

logger.warning(f"GPS altitude anomaly detected: {altitude\_diff:.2f}m in {time\_diff:.2f}s ({altitude\_rate:.2f} m/s)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_ALTITUDE\_ANOMALY)

return False

return True

def update\_gps\_dop(self, hdop=None, pdop=None, vdop=None):

"""

Update GPS Dilution of Precision values and check against thresholds

Args:

hdop: Horizontal Dilution of Precision

pdop: Position Dilution of Precision

vdop: Vertical Dilution of Precision

Returns:

bool: True if DOP values are acceptable, False otherwise

"""

# Update last GPS update time

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

# Check HDOP if provided

if hdop is not None and hdop > self.hdop\_threshold:

logger.warning(f"GPS high HDOP detected: {hdop:.2f} > {self.hdop\_threshold:.2f}")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HIGH\_DOP)

return False

# Check PDOP if provided

if pdop is not None and pdop > self.pdop\_threshold:

logger.warning(f"GPS high PDOP detected: {pdop:.2f} > {self.pdop\_threshold:.2f}")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HIGH\_DOP)

return False

# Check VDOP if provided

if vdop is not None and vdop > self.vdop\_threshold:

logger.warning(f"GPS high VDOP detected: {vdop:.2f} > {self.vdop\_threshold:.2f}")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HIGH\_DOP)

return False

return True

def update\_gps\_velocity(self, velocity, timestamp=None):

"""

Update current GPS velocity and check against physical constraints

Args:

velocity: Current velocity in m/s

timestamp: Timestamp of the velocity measurement (optional)

Returns:

bool: True if velocity is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# We'll need to implement velocity history tracking first

if not hasattr(self, 'previous\_velocities'):

self.previous\_velocities = []

self.velocity\_history\_max = 5

# Store new velocity

new\_velocity = (velocity, timestamp)

self.previous\_velocities.append(new\_velocity)

# Trim history if needed

if len(self.previous\_velocities) > self.velocity\_history\_max:

self.previous\_velocities.pop(0)

# Check for velocity constraint violation if we have at least two velocities

if len(self.previous\_velocities) >= 2:

return self.\_check\_velocity\_constraint()

return True

def \_check\_velocity\_constraint(self):

"""

Check if there's a sudden change in GPS velocity that exceeds physical limits

Returns:

bool: True if velocity is valid, False if anomaly detected

"""

# Get the two most recent velocities

current = self.previous\_velocities[-1]

previous = self.previous\_velocities[-2]

# Calculate velocity difference

velocity\_diff = abs(current[0] - previous[0])

time\_diff = current[1] - previous[1]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate acceleration

acceleration = velocity\_diff / time\_diff # m/s²

# Check if acceleration exceeds maximum expected value

if acceleration > self.max\_acceleration:

logger.warning(f"GPS velocity constraint violation: {velocity\_diff:.2f} m/s in {time\_diff:.2f}s ({acceleration:.2f} m/s²)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_VELOCITY\_CONSTRAINT)

return False

return True

def update\_gps\_satellites(self, satellites\_info):

"""

Update information about GPS satellites and check for constellation issues

Args:

satellites\_info: List of dictionaries with satellite information, each containing:

- 'id': Satellite ID

- 'elevation': Satellite elevation in degrees

- 'azimuth': Satellite azimuth in degrees

- 'snr': Signal-to-noise ratio in dB-Hz

- 'used': Boolean indicating if the satellite is used in solution

Returns:

bool: True if constellation is acceptable, False otherwise

"""

# Update last GPS update time

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

# Count satellites used in solution

satellites\_used = sum(1 for sat in satellites\_info if sat.get('used', False))

# Check if we have enough satellites

if satellites\_used < self.min\_satellites:

logger.warning(f"GPS weak constellation: {satellites\_used} satellites used < {self.min\_satellites} required")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_CONSTELLATION\_WEAK)

return False

# Check for multipath

if self.\_check\_multipath(satellites\_info):

logger.warning("GPS multipath detected based on satellite SNR changes")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_SIGNAL\_MULTIPATH)

return False

return True

def \_check\_multipath(self, satellites\_info):

"""

Check for multipath by monitoring sudden changes in SNR

Args:

satellites\_info: List of dictionaries with satellite information

Returns:

bool: True if multipath is detected, False otherwise

"""

multipath\_detected = False

for sat in satellites\_info:

sat\_id = sat.get('id')

current\_snr = sat.get('snr', 0)

# Skip if no SNR provided or satellite not used

if not sat.get('used', False) or current\_snr <= 0:

continue

# Check if we have previous SNR for this satellite

if sat\_id in self.previous\_snr\_values:

prev\_snr = self.previous\_snr\_values[sat\_id]

# Check for significant SNR drop

if prev\_snr - current\_snr > self.multipath\_snr\_threshold:

logger.warning(f"GPS multipath detected on satellite {sat\_id}: SNR dropped from {prev\_snr} to {current\_snr}")

multipath\_detected = True

# Update previous SNR

self.previous\_snr\_values[sat\_id] = current\_snr

return multipath\_detected

def update\_comprehensive\_gps\_data(self, data):

"""

Update all GPS data at once and perform comprehensive checks

Args:

data: Dictionary containing GPS data:

- 'position': (latitude, longitude, altitude)

- 'heading': heading in degrees

- 'velocity': velocity in m/s

- 'dop': dictionary with hdop, pdop, vdop values

- 'satellites': list of satellite dictionaries

- 'timestamp': time of the GPS fix

Returns:

bool: True if all checks pass, False if any failsafe is triggered

"""

timestamp = data.get('timestamp', time.time())

# Update position if available

position\_valid = True

if 'position' in data and len(data['position']) >= 2:

position = data['position']

if len(position) >= 3:

position\_valid = self.update\_gps\_position(position[0], position[1], position[2], timestamp)

else:

position\_valid = self.update\_gps\_position(position[0], position[1], None, timestamp)

# Update altitude separately if available with 3D position

if len(position) >= 3:

altitude\_valid = self.update\_gps\_altitude(position[2], timestamp)

position\_valid = position\_valid and altitude\_valid

# Update heading if available

heading\_valid = True

if 'heading' in data:

heading\_valid = self.update\_gps\_heading(data['heading'], timestamp)

# Update velocity if available

velocity\_valid = True

if 'velocity' in data:

velocity\_valid = self.update\_gps\_velocity(data['velocity'], timestamp)

# Update DOP values if available

dop\_valid = True

if 'dop' in data:

dop = data['dop']

dop\_valid = self.update\_gps\_dop(

dop.get('hdop'),

dop.get('pdop'),

dop.get('vdop')

)

# Update satellites info if available

satellites\_valid = True

if 'satellites' in data:

satellites\_valid = self.update\_gps\_satellites(data['satellites'])

# Mark last GPS update timestamp

self.last\_gps\_update = timestamp

# Return overall validity

return position\_valid and heading\_valid and velocity\_valid and dop\_valid and satellites\_valid

# Add a method to get GPS statistics

def get\_gps\_statistics(self):

"""Get statistics about GPS performance"""

return {

"position\_history\_count": len(self.previous\_gps\_positions),

"heading\_history\_count": len(self.previous\_headings),

"altitude\_history\_count": len(self.previous\_altitudes),

"velocity\_history\_count": len(getattr(self, 'previous\_velocities', [])),

"satellite\_count": len(self.previous\_snr\_values),

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

"thresholds": {

"max\_position\_jump": self.max\_position\_jump,

"max\_speed": self.max\_speed,

"max\_heading\_change": self.max\_heading\_change,

"max\_altitude\_change": self.max\_altitude\_change,

"hdop\_threshold": self.hdop\_threshold,

"pdop\_threshold": self.pdop\_threshold,

"vdop\_threshold": self.vdop\_threshold,

"min\_satellites": self.min\_satellites,

"min\_satellite\_snr": self.min\_satellite\_snr,

"max\_acceleration": self.max\_acceleration,

"multipath\_snr\_threshold": self.multipath\_snr\_threshold

}

}

# Example usage:

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

# Create an instance of the FailsafeModule

failsafe = FailsafeModule()

# Configure and start monitoring

failsafe.configure(

gps\_stale\_threshold=30.0,

internet\_timeout=10.0

)

# Set up callbacks

def on\_failsafe(reason):

print(f"FAILSAFE TRIGGERED: {reason.value}")

def on\_recovery(reason):

print(f"Recovery attempt for: {reason.value}")

return True

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

# Start monitoring

failsafe.start\_monitoring()

# Simulate GPS data updates

import random

import time

lat, lon = 37.7749, -122.4194 # Starting position

heading = 90.0 # Starting heading

altitude = 10.0 # Starting altitude

velocity = 0.5 # Starting velocity

try:

for i in range(100):

# Normal update

if i < 80:

# Simulate normal movement

lat += random.uniform(-0.0001, 0.0001)

lon += random.uniform(-0.0001, 0.0001)

heading = (heading + random.uniform(-5, 5)) % 360

altitude += random.uniform(-0.1, 0.1)

velocity = max(0, velocity + random.uniform(-0.1, 0.1))

# Update comprehensive GPS data

failsafe.update\_comprehensive\_gps\_data({

'position': (lat, lon, altitude),

'heading': heading,

'velocity': velocity,

'dop': {'hdop': random.uniform(1.0, 3.0)},

'satellites': [

{'id': i, 'snr': random.uniform(35, 45), 'used': True}

for i in range(10)

],

'timestamp': time.time()

})

else:

# Simulate GPS position jump to trigger failsafe

if i == 80:

print("Simulating GPS position jump...")

lat += 0.01 # Large jump

# Simulate GPS heading inconsistency

if i == 85:

print("Simulating GPS heading inconsistency...")

heading = (heading + 180) % 360 # 180-degree turn

# Simulate GPS altitude anomaly

if i == 90:

print("Simulating GPS altitude anomaly...")

altitude += 10.0 # Large altitude change

# Update with anomalous data

failsafe.update\_comprehensive\_gps\_data({

'position': (lat, lon, altitude),

'heading': heading,

'velocity': velocity,

'dop': {'hdop': random.uniform(1.0, 8.0)},

'satellites': [

{'id': i, 'snr': random.uniform(20, 40), 'used': True}

for i in range(random.randint(4, 10))

],

'timestamp': time.time()

})

# Print status

if i % 10 == 0:

status = failsafe.get\_status()

print(f"Status: {'FAILSAFE' if status['in\_failsafe\_mode'] else 'NORMAL'}")

if status['in\_failsafe\_mode']:

print(f"Reason: {status['failsafe\_reason']}")

time.sleep(0.5)

except KeyboardInterrupt:

print("Stopping...")

finally:

failsafe.stop\_monitoring()

print("Monitoring stopped")

SLEEP MODE:

import time

import threading

import datetime

import logging

from enum import Enum

import math

# Set up logging

logging.basicConfig(

level=logging.INFO,

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

handlers=[

logging.FileHandler(r"F:\GPS\task\_2\_waypoints\rover\_failsafe.csv", encoding='utf-8'),

logging.StreamHandler()

]

)

logger = logging.getLogger("rover\_failsafe")

class GPSFailsafeReason(Enum):

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

GPS\_DATA\_LOSS = "GPS data loss"

GPS\_STALE\_DATA = "GPS stale data"

INTERNET\_CONNECTION\_LOST = "Internet connection lost"

INTERNET\_CONNECTION\_SLOW = "Internet connection slow"

OVER\_TEMPERATURE = "Over temperature"

MODULE\_COMMUNICATION\_FAILURE = "Module communication failure"

SIGNAL\_INSTABILITY = "Signal instability"

SIGNIFICANT\_DRIFT = "Significant navigation drift"

PERSISTENT\_DRIFT = "Persistent navigation drift"

CUSTOM = "Custom reason"

GPS\_POSITION\_JUMP = "GPS position jump detected"

GPS\_HEADING\_INCONSISTENCY = "GPS heading inconsistency"

GPS\_ALTITUDE\_ANOMALY = "GPS altitude anomaly"

GPS\_VELOCITY\_CONSTRAINT = "GPS velocity constraint violation"

GPS\_HIGH\_DOP = "GPS high dilution of precision"

GPS\_SIGNAL\_MULTIPATH = "GPS signal multipath detected"

GPS\_CONSTELLATION\_WEAK = "GPS constellation geometry weak"

class DriftSeverity(Enum):

"""Enumeration of drift severity levels"""

MINOR = "Minor drift"

MODERATE = "Moderate drift"

SEVERE = "Severe drift"

CRITICAL = "Critical drift"

class DriftAction(Enum):

"""Enumeration of actions to take for drift"""

MONITOR = "Continue monitoring"

REALIGN = "Perform realignment"

SLOW\_DOWN = "Reduce speed"

PAUSE = "Pause movement"

FAILSAFE = "Enter failsafe mode"

class FailsafeModule:

"""

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

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

safety rather than navigation-specific safety.

"""

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

# Status flags

self.in\_failsafe\_mode = False

self.failsafe\_reason = None

self.recovery\_attempts = 0

self.recovery\_in\_progress = False

# Timestamp tracking

self.last\_gps\_update = None

self.last\_internet\_check = None

self.last\_module\_comm = None

self.last\_recovery\_attempt = None

# Thresholds and settings

self.gps\_stale\_threshold = 30.0 # seconds

self.internet\_timeout = 10.0 # seconds

self.module\_comm\_timeout = 15.0 # INCREASED: from 5.0 to 15.0 seconds

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

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

self.temperature\_threshold = 60.0 # degrees Celsius

# Signal tracking

self.signal\_loss\_events = []

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

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

# Drift monitoring

self.drift\_events = []

self.drift\_window = 300.0 # 5 minute window for drift events

self.drift\_threshold\_minor = 0.2 # meters

self.drift\_threshold\_moderate = 0.5 # meters

self.drift\_threshold\_severe = 1.0 # meters

self.drift\_threshold\_critical = 2.0 # meters

self.drift\_count\_threshold = 3 # Number of severe drift events to trigger failsafe

self.drift\_persistence\_threshold = 10 # Number of consecutive drift events to consider persistent

self.consecutive\_drift\_events = 0

self.last\_realignment = None

self.realignment\_cooldown = 60.0 # seconds between realignments

self.current\_drift\_action = DriftAction.MONITOR

# Callbacks

self.drift\_action\_callback = None

# Initialize monitoring threads

self.monitoring\_active = False

self.monitor\_thread = None

# Callback for failsafe activation

self.failsafe\_callback = None

self.recovery\_callback = None

# Initialize timestamps to prevent immediate failsafe triggers

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

self.last\_module\_comm = time.time() # ADDED: Initialize module communication timestamp

self.last\_internet\_check = time.time() # ADDED: Initialize internet check timestamp

# GPS position tracking

self.previous\_gps\_positions = [] # List of recent positions (lat, lon, alt, timestamp)

self.position\_history\_max = 10 # Maximum number of positions to store

self.max\_position\_jump = 5.0 # Maximum allowed position jump in meters

self.max\_speed = 2.0 # Maximum expected speed in m/s

# GPS heading tracking

self.previous\_headings = [] # List of recent headings (heading, timestamp)

self.heading\_history\_max = 5 # Maximum number of headings to store

self.max\_heading\_change = 45.0 # Maximum allowed heading change in degrees per second

# GPS altitude tracking

self.previous\_altitudes = [] # List of recent altitudes (altitude, timestamp)

self.altitude\_history\_max = 5 # Maximum number of altitudes to store

self.max\_altitude\_change = 1.0 # Maximum allowed altitude change in meters per second

# GPS DOP thresholds

self.hdop\_threshold = 5.0 # HDOP threshold for good precision

self.pdop\_threshold = 7.0 # PDOP threshold for good precision

self.vdop\_threshold = 7.0 # VDOP threshold for good precision

# GPS constellation settings

self.min\_satellites = 6 # Minimum satellites for reliable operation

self.min\_satellite\_snr = 30 # Minimum signal-to-noise ratio (dB-Hz)

# GPS velocity constraints

self.max\_acceleration = 1.0 # Maximum expected acceleration in m/s²

# Multipath detection

self.multipath\_snr\_threshold = 10 # SNR drop threshold for multipath detection

self.previous\_snr\_values = {} # Dictionary to store previous SNR values by satellite ID

# Initialize velocity tracking

self.previous\_velocities = []

self.velocity\_history\_max = 5

# Initialize satellite SNR tracking for multipath detection

if not hasattr(self, 'previous\_snr\_values'):

self.previous\_snr\_values = {}

logger.info("Failsafe module initialized")

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

"""

Set callbacks for failsafe activation and recovery

Args:

failsafe\_callback: Function to call when failsafe is triggered

Should accept FailsafeReason as an argument

recovery\_callback: Function to call when attempting recovery

Should accept FailsafeReason as an argument

drift\_action\_callback: Function to call when drift action changes

Should accept DriftAction and drift\_distance as arguments

"""

self.failsafe\_callback = failsafe\_callback

self.recovery\_callback = recovery\_callback

self.drift\_action\_callback = drift\_action\_callback

logger.info("Failsafe callbacks set")

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

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

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

drift\_thresholds=None, drift\_count\_threshold=None,

drift\_persistence\_threshold=None, realignment\_cooldown=None,

module\_comm\_timeout=None): # ADDED: module\_comm\_timeout parameter

"""Configure failsafe thresholds and settings"""

if gps\_stale\_threshold is not None:

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

if internet\_timeout is not None:

self.internet\_timeout = internet\_timeout

if module\_comm\_timeout is not None: # ADDED: Configure module communication timeout

self.module\_comm\_timeout = module\_comm\_timeout

if recovery\_interval is not None:

self.recovery\_interval = recovery\_interval

if max\_recovery\_attempts is not None:

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

if temperature\_threshold is not None:

self.temperature\_threshold = temperature\_threshold

if signal\_loss\_threshold is not None:

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

if drift\_thresholds is not None:

if 'minor' in drift\_thresholds:

self.drift\_threshold\_minor = drift\_thresholds['minor']

if 'moderate' in drift\_thresholds:

self.drift\_threshold\_moderate = drift\_thresholds['moderate']

if 'severe' in drift\_thresholds:

self.drift\_threshold\_severe = drift\_thresholds['severe']

if 'critical' in drift\_thresholds:

self.drift\_threshold\_critical = drift\_thresholds['critical']

if drift\_count\_threshold is not None:

self.drift\_count\_threshold = drift\_count\_threshold

if drift\_persistence\_threshold is not None:

self.drift\_persistence\_threshold = drift\_persistence\_threshold

if realignment\_cooldown is not None:

self.realignment\_cooldown = realignment\_cooldown

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

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

f"Module comm timeout={self.module\_comm\_timeout}s, " # ADDED: Log module comm timeout

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

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

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

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

def start\_monitoring(self):

"""Start the failsafe monitoring thread"""

if self.monitoring\_active:

logger.warning("Monitoring already active")

return

# Reset timestamps to prevent immediate failsafe trigger

current\_time = time.time()

self.last\_gps\_update = current\_time

self.last\_module\_comm = current\_time # ADDED: Reset module timestamp

self.last\_internet\_check = current\_time # ADDED: Reset internet timestamp

self.monitoring\_active = True

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

self.monitor\_thread.start()

logger.info("Failsafe monitoring started")

def stop\_monitoring(self):

"""Stop the failsafe monitoring thread"""

if not self.monitoring\_active:

logger.warning("Monitoring not active")

return

self.monitoring\_active = False

if self.monitor\_thread:

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

logger.info("Failsafe monitoring stopped")

def \_monitor\_loop(self):

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

while self.monitoring\_active:

try:

# Check various failsafe conditions

self.\_check\_gps\_status()

self.\_check\_internet\_connection()

self.\_check\_module\_communication()

self.\_check\_temperature()

self.\_check\_signal\_stability()

self.\_check\_drift\_status()

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

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

(self.last\_recovery\_attempt is None or

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

self.\_attempt\_recovery()

time.sleep(1.0) # Check conditions every second

except Exception as e:

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

def \_check\_gps\_status(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any GPS data

if self.last\_gps\_update is None:

# Log a warning instead of immediately triggering failsafe

logger.warning("No GPS data available yet")

# Initialize last\_gps\_update to prevent immediate failsafe

self.last\_gps\_update = current\_time

return

# Check if GPS data is stale

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

# Log a warning before triggering failsafe

logger.warning(f"GPS data stale: Last update was {current\_time - self.last\_gps\_update:.1f}s ago")

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

def \_check\_internet\_connection(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any internet connectivity data

if self.last\_internet\_check is None:

# ADDED: Initialize internet check to prevent immediate failsafe

logger.warning("No internet connection data available yet")

self.last\_internet\_check = current\_time

return

# Check if internet connection is lost or too slow

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

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Internet connection lost: Last check was {current\_time - self.last\_internet\_check:.1f}s ago")

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

def \_check\_module\_communication(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Check if we have any module communication data

if self.last\_module\_comm is None:

# ADDED: Initialize module communication to prevent immediate failsafe

logger.warning("No module communication data available yet")

self.last\_module\_comm = current\_time

return

# Module communication timeout (increased from 5 to 15 seconds)

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

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Module communication failure: Last communication was {current\_time - self.last\_module\_comm:.1f}s ago")

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

def \_check\_temperature(self):

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

if self.in\_failsafe\_mode:

return

# This is a placeholder for actual temperature checking

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

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

if temperature > self.temperature\_threshold:

# ADDED: Log a warning before triggering failsafe

logger.warning(f"System temperature too high: {temperature:.1f}°C > {self.temperature\_threshold:.1f}°C")

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

def \_check\_signal\_stability(self):

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

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Remove signal loss events older than the window

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

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

# Check if we have too many signal loss events

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

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Signal instability detected: {len(self.signal\_loss\_events)} losses in the last {self.signal\_loss\_window:.1f}s")

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

def \_check\_drift\_status(self):

"""Check if drift has exceeded thresholds and take appropriate action"""

if self.in\_failsafe\_mode:

return

current\_time = time.time()

# Remove drift events older than the window

self.drift\_events = [event for event in self.drift\_events

if current\_time - event['timestamp'] <= self.drift\_window]

# Check for severe drift events

severe\_drift\_events = [event for event in self.drift\_events

if event['severity'] in (DriftSeverity.SEVERE, DriftSeverity.CRITICAL)]

# Check if we have too many severe drift events

if len(severe\_drift\_events) >= self.drift\_count\_threshold:

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Significant drift detected: {len(severe\_drift\_events)} severe events in the last {self.drift\_window:.1f}s")

self.\_trigger\_failsafe(GPSFailsafeReason.SIGNIFICANT\_DRIFT)

return

# Check for persistent drift (many consecutive drift events)

if self.consecutive\_drift\_events >= self.drift\_persistence\_threshold:

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Persistent drift detected: {self.consecutive\_drift\_events} consecutive drift events")

self.\_trigger\_failsafe(GPSFailsafeReason.PERSISTENT\_DRIFT)

return

# If we have any action other than monitoring in place, check if it's time to clear it

if self.current\_drift\_action != DriftAction.MONITOR and not self.drift\_events:

self.\_set\_drift\_action(DriftAction.MONITOR, 0)

def \_trigger\_failsafe(self, reason):

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

if self.in\_failsafe\_mode:

return

print("Sleep mode activated, going into sleep mode")

self.in\_failsafe\_mode = True

self.failsafe\_reason = reason

self.recovery\_attempts = 0

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

# Execute failsafe callback if provided

if self.failsafe\_callback:

try:

self.failsafe\_callback(reason)

except Exception as e:

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

def \_attempt\_recovery(self):

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

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

return

self.recovery\_in\_progress = True

self.recovery\_attempts += 1

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

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

# Execute recovery callback if provided

recovery\_success = False # ADDED: Initialize success flag

if self.recovery\_callback:

try:

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

if recovery\_success:

self.\_clear\_failsafe()

except Exception as e:

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

recovery\_success = False # ADDED: Set success to false on exception

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

if self.recovery\_attempts >= self.max\_recovery\_attempts and not recovery\_success:

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

# Keep in failsafe mode but allow future recovery attempts

self.recovery\_in\_progress = False

def \_clear\_failsafe(self):

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

if not self.in\_failsafe\_mode:

return

old\_reason = self.failsafe\_reason

self.in\_failsafe\_mode = False

self.failsafe\_reason = None

self.recovery\_attempts = 0

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

# ADDED: Reset timestamps to prevent immediate re-triggering of failsafe

current\_time = time.time()

self.last\_gps\_update = current\_time

self.last\_module\_comm = current\_time

self.last\_internet\_check = current\_time

def \_determine\_drift\_severity(self, drift\_distance):

"""

Determine the severity of a drift event based on distance

Args:

drift\_distance: Distance in meters

Returns:

DriftSeverity: Severity level of the drift

"""

if drift\_distance >= self.drift\_threshold\_critical:

return DriftSeverity.CRITICAL

elif drift\_distance >= self.drift\_threshold\_severe:

return DriftSeverity.SEVERE

elif drift\_distance >= self.drift\_threshold\_moderate:

return DriftSeverity.MODERATE

elif drift\_distance >= self.drift\_threshold\_minor:

return DriftSeverity.MINOR

else:

return None # No significant drift

def \_determine\_drift\_action(self, severity, consecutive\_count):

"""

Determine what action to take based on drift severity and consecutive count

Args:

severity: DriftSeverity level

consecutive\_count: Number of consecutive drift events

Returns:

DriftAction: Action to take

"""

if severity == DriftSeverity.CRITICAL:

return DriftAction.FAILSAFE

if severity == DriftSeverity.SEVERE:

if consecutive\_count >= self.drift\_persistence\_threshold:

return DriftAction.FAILSAFE

elif consecutive\_count >= self.drift\_persistence\_threshold // 2:

return DriftAction.PAUSE

else:

return DriftAction.REALIGN

if severity == DriftSeverity.MODERATE:

if consecutive\_count >= self.drift\_persistence\_threshold:

return DriftAction.PAUSE

elif consecutive\_count >= 3:

return DriftAction.REALIGN

else:

return DriftAction.SLOW\_DOWN

if severity == DriftSeverity.MINOR:

if consecutive\_count >= self.drift\_persistence\_threshold:

return DriftAction.SLOW\_DOWN

else:

return DriftAction.MONITOR

return DriftAction.MONITOR

def \_set\_drift\_action(self, action, drift\_distance):

"""

Set and execute a drift action

Args:

action: DriftAction to take

drift\_distance: Current drift distance in meters

"""

# Don't change action if we're in failsafe mode

if self.in\_failsafe\_mode:

return

# Don't downgrade action severity (e.g., from PAUSE to SLOW\_DOWN)

action\_severity = {

DriftAction.MONITOR: 0,

DriftAction.SLOW\_DOWN: 1,

DriftAction.REALIGN: 2,

DriftAction.PAUSE: 3,

DriftAction.FAILSAFE: 4

}

# Only update if the new action is more severe or we're returning to monitoring

if (action == DriftAction.MONITOR or

action\_severity[action] > action\_severity[self.current\_drift\_action]):

# Log the action change

if action != self.current\_drift\_action:

logger.info(f"Drift action changed: {action.value} (drift: {drift\_distance:.2f}m)")

# Update the current action

self.current\_drift\_action = action

# Handle realignment cooldown

current\_time = time.time()

if (action == DriftAction.REALIGN and self.last\_realignment and

current\_time - self.last\_realignment < self.realignment\_cooldown):

logger.info(f"Skipping realignment - cooldown period active")

return

# If it's a realignment, update the timestamp

if action == DriftAction.REALIGN:

self.last\_realignment = current\_time

# Execute action callback if provided

if self.drift\_action\_callback:

try:

self.drift\_action\_callback(action, drift\_distance)

except Exception as e:

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

# If action is failsafe, trigger it

if action == DriftAction.FAILSAFE:

self.\_trigger\_failsafe(GPSFailsafeReason.SIGNIFICANT\_DRIFT)

# API Methods for rover interaction

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

"""

Update GPS status information

Args:

has\_fix: Whether GPS has a valid fix

satellites: Number of satellites in view

hdop: Horizontal dilution of precision

"""

if not has\_fix or satellites < 6 or hdop > 5.0:

# Register as signal loss event

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

else:

# Log when GPS status is updated

logger.debug(f"GPS status updated: fix={has\_fix}, satellites={satellites}, hdop={hdop}")

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

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

"""

Update internet connection status

Args:

connected: Whether internet is connected

latency: Latency in seconds

"""

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

# Consider high latency as a slow connection

if not connected or latency > 2.0:

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

reason = (GPSFailsafeReason.INTERNET\_CONNECTION\_LOST if not connected

else GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW)

# ADDED: Log a warning before triggering failsafe

logger.warning(f"Internet issue detected: {'disconnected' if not connected else f'high latency ({latency:.2f}s)'}")

self.\_trigger\_failsafe(reason)

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

"""

Update module communication timestamp

Args:

timestamp: Optional timestamp to use, defaults to current time

"""

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

# ADDED: Log when module communication is updated

logger.debug(f"Module communication updated: {self.last\_module\_comm}")

def report\_signal\_loss(self):

"""Report a signal loss event"""

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

# ADDED: Log when signal loss is reported

logger.debug(f"Signal loss reported, total events in window: {len(self.signal\_loss\_events)}")

def report\_drift(self, actual\_position, expected\_position):

"""

Report a position drift event with actual and expected positions

Args:

actual\_position: Tuple of (latitude, longitude) or (x, y) for actual position

expected\_position: Tuple of (latitude, longitude) or (x, y) for expected position

Returns:

Tuple of (drift\_distance, severity)

"""

# Calculate drift distance

if len(actual\_position) == 2 and len(expected\_position) == 2:

# For lat/lon coordinates, we need to use haversine formula

if abs(actual\_position[0]) <= 90 and abs(actual\_position[1]) <= 180:

drift\_distance = self.\_haversine\_distance(

actual\_position[0], actual\_position[1],

expected\_position[0], expected\_position[1]

)

else:

# For x/y coordinates, we can use Euclidean distance

drift\_distance = math.sqrt(

(actual\_position[0] - expected\_position[0])\*\*2 +

(actual\_position[1] - expected\_position[1])\*\*2

)

else:

logger.error(f"Invalid position format: {actual\_position}, {expected\_position}")

return 0, None

# Determine severity

severity = self.\_determine\_drift\_severity(drift\_distance)

# If there's significant drift, record it

if severity is not None:

current\_time = time.time()

self.drift\_events.append({

'timestamp': current\_time,

'distance': drift\_distance,

'severity': severity,

'actual': actual\_position,

'expected': expected\_position

})

# Update consecutive drift counter

self.consecutive\_drift\_events += 1

# Log the drift event

logger.info(f"Drift detected: {drift\_distance:.2f}m, Severity: {severity.value}, "

f"Consecutive: {self.consecutive\_drift\_events}")

# Determine and set action

action = self.\_determine\_drift\_action(severity, self.consecutive\_drift\_events)

self.\_set\_drift\_action(action, drift\_distance)

else:

# Reset consecutive drift counter if no significant drift

self.consecutive\_drift\_events = 0

# If we were taking action, go back to monitoring

if self.current\_drift\_action != DriftAction.MONITOR:

self.\_set\_drift\_action(DriftAction.MONITOR, drift\_distance)

return drift\_distance, severity

def \_haversine\_distance(self, lat1, lon1, lat2, lon2):

"""

Calculate the great circle distance between two points

on the earth (specified in decimal degrees)

"""

# Convert decimal degrees to radians

lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

# Haversine formula

dlon = lon2 - lon1

dlat = lat2 - lat1

a = math.sin(dlat/2)\*\*2 + math.cos(lat1) \* math.cos(lat2) \* math.sin(dlon/2)\*\*2

c = 2 \* math.asin(math.sqrt(a))

r = 6371000 # Radius of earth in meters

return c \* r

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

"""

Manually trigger failsafe mode with a custom reason

Args:

reason\_text: Description of the custom reason

"""

custom\_reason = GPSFailsafeReason.CUSTOM

self.\_trigger\_failsafe(custom\_reason)

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

def force\_recovery(self):

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

if self.in\_failsafe\_mode:

logger.info("Forcing recovery attempt")

self.\_attempt\_recovery()

return True # ADDED: Return success indicator

else:

logger.warning("Cannot force recovery - not in failsafe mode")

return False # ADDED: Return failure indicator

def force\_realignment(self):

"""Force a realignment regardless of conditions"""

logger.info("Forcing realignment")

self.\_set\_drift\_action(DriftAction.REALIGN, 0)

self.last\_realignment = time.time()

return True # ADDED: Return success indicator

def reset\_drift\_tracking(self):

"""Reset all drift tracking data"""

self.drift\_events = []

self.consecutive\_drift\_events = 0

self.current\_drift\_action = DriftAction.MONITOR

logger.info("Drift tracking reset")

return True # ADDED: Return success indicator

def get\_status(self):

"""Get current failsafe status information"""

return {

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

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

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

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

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

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

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

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

"drift\_events": len(self.drift\_events),

"consecutive\_drift\_events": self.consecutive\_drift\_events,

"current\_drift\_action": self.current\_drift\_action.value,

"last\_realignment": self.last\_realignment,

"monitoring\_active": self.monitoring\_active

}

def get\_drift\_statistics(self):

"""Get statistics about drift events"""

if not self.drift\_events:

return {

"count": 0,

"avg\_distance": 0,

"max\_distance": 0,

"min\_distance": 0,

"severity\_counts": {s.name: 0 for s in DriftSeverity}

}

# Calculate statistics

distances = [event['distance'] for event in self.drift\_events]

severity\_counts = {s.name: 0 for s in DriftSeverity}

for event in self.drift\_events:

severity\_counts[event['severity'].name] += 1

return {

"count": len(self.drift\_events),

"avg\_distance": sum(distances) / len(distances),

"max\_distance": max(distances),

"min\_distance": min(distances),

"severity\_counts": severity\_counts

}

def \_get\_system\_temperature(self):

"""

Get the current system temperature

This is a placeholder implementation that should be replaced

with actual hardware temperature readings in a real system

"""

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

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

return 45.0 # Simulated temperature in Celsius

def update\_gps\_position(self, latitude, longitude, altitude=None, timestamp=None):

"""

Update current GPS position and check for anomalies

Args:

latitude: Current latitude in decimal degrees

longitude: Current longitude in decimal degrees

altitude: Current altitude in meters (optional)

timestamp: Timestamp of the position fix (optional, defaults to current time)

Returns:

bool: True if position is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# Update last GPS update time

self.last\_gps\_update = timestamp

# Store new position

new\_position = (latitude, longitude, altitude, timestamp)

self.previous\_gps\_positions.append(new\_position)

# Trim history if needed

if len(self.previous\_gps\_positions) > self.position\_history\_max:

self.previous\_gps\_positions.pop(0)

# Check for position jump if we have at least two positions

if len(self.previous\_gps\_positions) >= 2:

return self.\_check\_position\_jump()

return True

def \_check\_position\_jump(self):

"""

Check if there's a sudden jump in GPS position that exceeds physical limits

Returns:

bool: True if position is valid, False if jump detected

"""

# Get the two most recent positions

current = self.previous\_gps\_positions[-1]

previous = self.previous\_gps\_positions[-2]

# Calculate distance and time difference

distance = self.\_haversine\_distance(

current[0], current[1],

previous[0], previous[1]

)

time\_diff = current[3] - previous[3]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate speed

speed = distance / time\_diff # m/s

# Check if speed exceeds maximum expected speed by a significant margin

if speed > self.max\_speed \* 1.5:

logger.warning(f"GPS position jump detected: {distance:.2f}m in {time\_diff:.2f}s ({speed:.2f} m/s)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_POSITION\_JUMP)

return False

return True

def update\_gps\_heading(self, heading, timestamp=None):

"""

Update current GPS heading and check for inconsistencies

Args:

heading: Current heading in degrees (0-360)

timestamp: Timestamp of the heading (optional, defaults to current time)

Returns:

bool: True if heading is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# Store new heading

new\_heading = (heading, timestamp)

self.previous\_headings.append(new\_heading)

# Trim history if needed

if len(self.previous\_headings) > self.heading\_history\_max:

self.previous\_headings.pop(0)

# Check for heading inconsistency if we have at least two headings

if len(self.previous\_headings) >= 2:

return self.\_check\_heading\_consistency()

return True

def \_check\_heading\_consistency(self):

"""

Check if there's a sudden change in GPS heading that exceeds physical limits

Returns:

bool: True if heading is valid, False if inconsistency detected

"""

# Get the two most recent headings

current = self.previous\_headings[-1]

previous = self.previous\_headings[-2]

# Calculate heading difference

heading\_diff = min(

abs(current[0] - previous[0]),

360 - abs(current[0] - previous[0])

)

time\_diff = current[1] - previous[1]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate rate of heading change

heading\_rate = heading\_diff / time\_diff # degrees/s

# Check if heading change exceeds maximum expected change

if heading\_rate > self.max\_heading\_change:

logger.warning(f"GPS heading inconsistency detected: {heading\_diff:.2f}° in {time\_diff:.2f}s ({heading\_rate:.2f}°/s)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HEADING\_INCONSISTENCY)

return False

return True

def update\_gps\_altitude(self, altitude, timestamp=None):

"""

Update current GPS altitude and check for anomalies

Args:

altitude: Current altitude in meters

timestamp: Timestamp of the altitude (optional, defaults to current time)

Returns:

bool: True if altitude is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# Store new altitude

new\_altitude = (altitude, timestamp)

self.previous\_altitudes.append(new\_altitude)

# Trim history if needed

if len(self.previous\_altitudes) > self.altitude\_history\_max:

self.previous\_altitudes.pop(0)

# Check for altitude anomaly if we have at least two altitudes

if len(self.previous\_altitudes) >= 2:

return self.\_check\_altitude\_anomaly()

return True

def \_check\_altitude\_anomaly(self):

"""

Check if there's a sudden change in GPS altitude that exceeds physical limits

Returns:

bool: True if altitude is valid, False if anomaly detected

"""

# Get the two most recent altitudes

current = self.previous\_altitudes[-1]

previous = self.previous\_altitudes[-2]

# Calculate altitude difference

altitude\_diff = abs(current[0] - previous[0])

time\_diff = current[1] - previous[1]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate rate of altitude change

altitude\_rate = altitude\_diff / time\_diff # m/s

# Check if altitude change exceeds maximum expected change

if altitude\_rate > self.max\_altitude\_change:

logger.warning(f"GPS altitude anomaly detected: {altitude\_diff:.2f}m in {time\_diff:.2f}s ({altitude\_rate:.2f} m/s)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_ALTITUDE\_ANOMALY)

return False

return True

def update\_gps\_dop(self, hdop=None, pdop=None, vdop=None):

"""

Update GPS Dilution of Precision values and check against thresholds

Args:

hdop: Horizontal Dilution of Precision

pdop: Position Dilution of Precision

vdop: Vertical Dilution of Precision

Returns:

bool: True if DOP values are acceptable, False otherwise

"""

# Update last GPS update time

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

# Check HDOP if provided

if hdop is not None and hdop > self.hdop\_threshold:

logger.warning(f"GPS high HDOP detected: {hdop:.2f} > {self.hdop\_threshold:.2f}")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HIGH\_DOP)

return False

# Check PDOP if provided

if pdop is not None and pdop > self.pdop\_threshold:

logger.warning(f"GPS high PDOP detected: {pdop:.2f} > {self.pdop\_threshold:.2f}")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HIGH\_DOP)

return False

# Check VDOP if provided

if vdop is not None and vdop > self.vdop\_threshold:

logger.warning(f"GPS high VDOP detected: {vdop:.2f} > {self.vdop\_threshold:.2f}")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_HIGH\_DOP)

return False

return True

def update\_gps\_velocity(self, velocity, timestamp=None):

"""

Update current GPS velocity and check against physical constraints

Args:

velocity: Current velocity in m/s

timestamp: Timestamp of the velocity measurement (optional)

Returns:

bool: True if velocity is valid, False if anomaly detected

"""

if timestamp is None:

timestamp = time.time()

# We'll need to implement velocity history tracking first

if not hasattr(self, 'previous\_velocities'):

self.previous\_velocities = []

self.velocity\_history\_max = 5

# Store new velocity

new\_velocity = (velocity, timestamp)

self.previous\_velocities.append(new\_velocity)

# Trim history if needed

if len(self.previous\_velocities) > self.velocity\_history\_max:

self.previous\_velocities.pop(0)

# Check for velocity constraint violation if we have at least two velocities

if len(self.previous\_velocities) >= 2:

return self.\_check\_velocity\_constraint()

return True

def \_check\_velocity\_constraint(self):

"""

Check if there's a sudden change in GPS velocity that exceeds physical limits

Returns:

bool: True if velocity is valid, False if anomaly detected

"""

# Get the two most recent velocities

current = self.previous\_velocities[-1]

previous = self.previous\_velocities[-2]

# Calculate velocity difference

velocity\_diff = abs(current[0] - previous[0])

time\_diff = current[1] - previous[1]

# Avoid division by zero

if time\_diff <= 0:

return True

# Calculate acceleration

acceleration = velocity\_diff / time\_diff # m/s²

# Check if acceleration exceeds maximum expected value

if acceleration > self.max\_acceleration:

logger.warning(f"GPS velocity constraint violation: {velocity\_diff:.2f} m/s in {time\_diff:.2f}s ({acceleration:.2f} m/s²)")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_VELOCITY\_CONSTRAINT)

return False

return True

def update\_gps\_satellites(self, satellites\_info):

"""

Update information about GPS satellites and check for constellation issues

Args:

satellites\_info: List of dictionaries with satellite information, each containing:

- 'id': Satellite ID

- 'elevation': Satellite elevation in degrees

- 'azimuth': Satellite azimuth in degrees

- 'snr': Signal-to-noise ratio in dB-Hz

- 'used': Boolean indicating if the satellite is used in solution

Returns:

bool: True if constellation is acceptable, False otherwise

"""

# Update last GPS update time

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

# Count satellites used in solution

satellites\_used = sum(1 for sat in satellites\_info if sat.get('used', False))

# Check if we have enough satellites

if satellites\_used < self.min\_satellites:

logger.warning(f"GPS weak constellation: {satellites\_used} satellites used < {self.min\_satellites} required")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_CONSTELLATION\_WEAK)

return False

# Check for multipath

if self.\_check\_multipath(satellites\_info):

logger.warning("GPS multipath detected based on satellite SNR changes")

self.\_trigger\_failsafe(GPSFailsafeReason.GPS\_SIGNAL\_MULTIPATH)

return False

return True

def \_check\_multipath(self, satellites\_info):

"""

Check for multipath by monitoring sudden changes in SNR

Args:

satellites\_info: List of dictionaries with satellite information

Returns:

bool: True if multipath is detected, False otherwise

"""

multipath\_detected = False

for sat in satellites\_info:

sat\_id = sat.get('id')

current\_snr = sat.get('snr', 0)

# Skip if no SNR provided or satellite not used

if not sat.get('used', False) or current\_snr <= 0:

continue

# Check if we have previous SNR for this satellite

if sat\_id in self.previous\_snr\_values:

prev\_snr = self.previous\_snr\_values[sat\_id]

# Check for significant SNR drop

if prev\_snr - current\_snr > self.multipath\_snr\_threshold:

logger.warning(f"GPS multipath detected on satellite {sat\_id}: SNR dropped from {prev\_snr} to {current\_snr}")

multipath\_detected = True

# Update previous SNR

self.previous\_snr\_values[sat\_id] = current\_snr

return multipath\_detected

def update\_comprehensive\_gps\_data(self, data):

"""

Update all GPS data at once and perform comprehensive checks

Args:

data: Dictionary containing GPS data:

- 'position': (latitude, longitude, altitude)

- 'heading': heading in degrees

- 'velocity': velocity in m/s

- 'dop': dictionary with hdop, pdop, vdop values

- 'satellites': list of satellite dictionaries

- 'timestamp': time of the GPS fix

Returns:

bool: True if all checks pass, False if any failsafe is triggered

"""

timestamp = data.get('timestamp', time.time())

# Update position if available

position\_valid = True

if 'position' in data and len(data['position']) >= 2:

position = data['position']

if len(position) >= 3:

position\_valid = self.update\_gps\_position(position[0], position[1], position[2], timestamp)

else:

position\_valid = self.update\_gps\_position(position[0], position[1], None, timestamp)

# Update altitude separately if available with 3D position

if len(position) >= 3:

altitude\_valid = self.update\_gps\_altitude(position[2], timestamp)

position\_valid = position\_valid and altitude\_valid

# Update heading if available

heading\_valid = True

if 'heading' in data:

heading\_valid = self.update\_gps\_heading(data['heading'], timestamp)

# Update velocity if available

velocity\_valid = True

if 'velocity' in data:

velocity\_valid = self.update\_gps\_velocity(data['velocity'], timestamp)

# Update DOP values if available

dop\_valid = True

if 'dop' in data:

dop = data['dop']

dop\_valid = self.update\_gps\_dop(

dop.get('hdop'),

dop.get('pdop'),

dop.get('vdop')

)

# Update satellites info if available

satellites\_valid = True

if 'satellites' in data:

satellites\_valid = self.update\_gps\_satellites(data['satellites'])

# Mark last GPS update timestamp

self.last\_gps\_update = timestamp

# Return overall validity

return position\_valid and heading\_valid and velocity\_valid and dop\_valid and satellites\_valid

# Add a method to get GPS statistics

def get\_gps\_statistics(self):

"""Get statistics about GPS performance"""

return {

"position\_history\_count": len(self.previous\_gps\_positions),

"heading\_history\_count": len(self.previous\_headings),

"altitude\_history\_count": len(self.previous\_altitudes),

"velocity\_history\_count": len(getattr(self, 'previous\_velocities', [])),

"satellite\_count": len(self.previous\_snr\_values),

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

"thresholds": {

"max\_position\_jump": self.max\_position\_jump,

"max\_speed": self.max\_speed,

"max\_heading\_change": self.max\_heading\_change,

"max\_altitude\_change": self.max\_altitude\_change,

"hdop\_threshold": self.hdop\_threshold,

"pdop\_threshold": self.pdop\_threshold,

"vdop\_threshold": self.vdop\_threshold,

"min\_satellites": self.min\_satellites,

"min\_satellite\_snr": self.min\_satellite\_snr,

"max\_acceleration": self.max\_acceleration,

"multipath\_snr\_threshold": self.multipath\_snr\_threshold

}

}

# Example usage:

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

# Create an instance of the FailsafeModule

failsafe = FailsafeModule()

# Configure and start monitoring

failsafe.configure(

gps\_stale\_threshold=30.0,

internet\_timeout=10.0

)

# Set up callbacks

def on\_failsafe(reason):

print(f"FAILSAFE TRIGGERED: {reason.value}")

def on\_recovery(reason):

print(f"Recovery attempt for: {reason.value}")

return True

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

# Start monitoring

failsafe.start\_monitoring()

# Simulate GPS data updates

import random

import time

lat, lon = 37.7749, -122.4194 # Starting position

heading = 90.0 # Starting heading

altitude = 10.0 # Starting altitude

velocity = 0.5 # Starting velocity

try:

for i in range(100):

# Normal update

if i < 80:

# Simulate normal movement

lat += random.uniform(-0.0001, 0.0001)

lon += random.uniform(-0.0001, 0.0001)

heading = (heading + random.uniform(-5, 5)) % 360

altitude += random.uniform(-0.1, 0.1)

velocity = max(0, velocity + random.uniform(-0.1, 0.1))

# Update comprehensive GPS data

failsafe.update\_comprehensive\_gps\_data({

'position': (lat, lon, altitude),

'heading': heading,

'velocity': velocity,

'dop': {'hdop': random.uniform(1.0, 3.0)},

'satellites': [

{'id': i, 'snr': random.uniform(35, 45), 'used': True}

for i in range(10)

],

'timestamp': time.time()

})

else:

# Simulate GPS position jump to trigger failsafe

if i == 80:

print("Simulating GPS position jump...")

lat += 0.01 # Large jump

# Simulate GPS heading inconsistency

if i == 85:

print("Simulating GPS heading inconsistency...")

heading = (heading + 180) % 360 # 180-degree turn

# Simulate GPS altitude anomaly

if i == 90:

print("Simulating GPS altitude anomaly...")

altitude += 10.0 # Large altitude change

# Update with anomalous data

failsafe.update\_comprehensive\_gps\_data({

'position': (lat, lon, altitude),

'heading': heading,

'velocity': velocity,

'dop': {'hdop': random.uniform(1.0, 8.0)},

'satellites': [

{'id': i, 'snr': random.uniform(20, 40), 'used': True}

for i in range(random.randint(4, 10))

],

'timestamp': time.time()

})

# Print status

if i % 10 == 0:

status = failsafe.get\_status()

print(f"Status: {'FAILSAFE' if status['in\_failsafe\_mode'] else 'NORMAL'}")

if status['in\_failsafe\_mode']:

print(f"Reason: {status['failsafe\_reason']}")

time.sleep(0.5)

except KeyboardInterrupt:

print("Stopping...")

finally:

failsafe.stop\_monitoring()

print("Monitoring stopped")