Notes: Route Boundaries Extraction and Processing

1. Importing and Preprocessing Data

- The function starts by **filtering** the dataset (`df`) to retain only rows where:
  `line_type == 'location'` → Ensures we focus only on location-related entries.
  `geonameId` is **not NaN** → Ensures calculations are done only for valid locations.
- If **no valid locations exist**, the function **returns an empty DataFrame** and logs a warning.

---

2. Extracting First and Last Locations

- The function **groups the dataset by `route_description`** to analyze each route separately.
- Using **vectorized `.agg()` operations**, it extracts:
  First location (latitude & longitude).
  Last location (latitude & longitude).
- This method is **faster than iterating through rows**, making it more efficient for large datasets.

---

3. Computing the Bearing (Direction of Travel)

- The **GeographicLib Geodesic function** computes the **bearing** (azimuth) between the **first and last location** of each route.
- **Safety checks** ensure that if any coordinate is missing, the bearing is **set to NaN** to prevent calculation errors.

---

4. Exporting Processed Data

- The processed data (`bounds_df`) is **saved to a CSV file** for further use

5. Key Optimizations & Improvements

Efficient Data Processing:

- Used **vectorized operations (`.agg()`)** instead of slow loops.

- Pre-filtered NaN values **before processing**, avoiding unnecessary calculations.

Memory & Performance Optimization:

- Removed **redundant `.copy()` calls** to minimize memory usage.
- Used `pd.isna()` instead of `np.isnan()` for better **Pandas compatibility**.

Improved Maintainability & Error Handling:

- **Warning added for empty datasets** (ensuring users are alerted if no valid locations exist).
- **Ensured missing values do not break calculations**, preventing errors in the bearing computation.

Automation script instead of hand checking bounds

```
import pandas as pd
import numpy as np
from geographiclib.geodesic import Geodesic

# Load the existing bounds file
bounds_df =
pd.read_csv('/content/drive/MyDrive/EmDigitPageFiles/GM1684/497635/G
M1684/page/bounds_df.csv')

# Initialize a log to track corrections
correction_log = []

# Geodesic calculator
geod = Geodesic.WGS84

# Function to recompute bearing
def compute_bearing(row):
    """Compute bearing if coordinates are valid."""
    if pd.isna(row['first_lat']) or pd.isna(row['first_lng']) or
pd.isna(row['last_lat']) or pd.isna(row['last_lng']):
        return np.nan
    return geod.Inverse(row['first_lat'], row['first_lng'],
row['last_lat'], row['last_lng'])['azi1']

# Iterate over the dataset for corrections
for index, row in bounds_df.iterrows():
    corrected = False

    # If first_lat or first_lng is missing, try using next valid row
within the same route
    if pd.isna(row['first_lat']) or pd.isna(row['first_lng']):
        next_valid = bounds_df.loc[(bounds_df['route_description']
== row['route_description']) &

bounds_df['first_lat'].notna()].head(1)
        if not next_valid.empty:
            bounds_df.at[index, 'first_lat'] =
next_valid.iloc[0]['first_lat']
            bounds_df.at[index, 'first_lng'] =
next_valid.iloc[0]['first_lng']
            correction_log.append([row['route_description'],
'first_lat/lng corrected using next valid row'])
            corrected = True

    # If last_lat or last_lng is missing, try using previous valid
row within the same route
    if pd.isna(row['last_lat']) or pd.isna(row['last_lng']):
        prev_valid = bounds_df.loc[(bounds_df['route_description']
== row['route_description']) &

bounds_df['last_lat'].notna()].tail(1)
        if not prev_valid.empty:
            bounds_df.at[index, 'last_lat'] =
prev_valid.iloc[0]['last_lat']
            bounds_df.at[index, 'last_lng'] =
prev_valid.iloc[0]['last_lng']
            correction_log.append([row['route_description'],
'last_lat/lng corrected using previous valid row'])
            corrected = True

    # Check if bearing is missing or incorrect, recompute it
    if pd.isna(row['bearing']) or row['bearing'] < 0 or
row['bearing'] > 360:
        new_bearing = compute_bearing(row)
        if not pd.isna(new_bearing):
            bounds_df.at[index, 'bearing'] = new_bearing
            correction_log.append([row['route_description'],
'Bearing recalculated'])
            corrected = True

# Save the corrected file only if changes were made
if correction_log:

bounds_df.to_csv('/content/drive/MyDrive/EmDigitPageFiles/GM1684/497
635/GM1684/page/bounds_df.csv', index=False)

    # Save the correction log
    log_df = pd.DataFrame(correction_log,
columns=['route_description', 'Correction Applied'])

log_df.to_csv('/content/drive/MyDrive/EmDigitPageFiles/GM1684/497635
/GM1684/page/bounds_check_log.csv', index=False)

    print("  Bounds file corrected and saved. Corrections logged in
bounds_check_log.csv.")
else:
    print("  No corrections needed. The bounds file is clean.")
```

How This Script Works

1. **Loads `bounds_df.csv`** for checking.
2. **Fixes Missing Values**:
   - If `first_lat` or `first_lng` is missing, replaces with **next valid row** within the same route.
   - If `last_lat` or `last_lng` is missing, replaces with **previous valid row** within the same route.
3. **Fixes Bearings**:
   - If `bearing` is missing or out of range, recalculates it.
4. **Logs All Corrections**:
   - Saves corrections in `bounds_check_log.csv`.
5. **Saves Corrected File Only If Needed**:

  - **If corrections were made**, overwrites `bounds_df.csv`.
  - **If no issues were found**, prints `"  No corrections needed."`

Why This is Foolproof

Avoids Hardcoding Fixes: Uses dynamic logic to fill in missing data based on real values in the dataset.
Prevents Wrong Replacements: Only applies fixes **when valid replacement values exist**.
Ensures Bearings are Always Correct: If a bearing is invalid or missing, it gets recalculated properly.
Keeps a Log of All Changes: Saves a `bounds_check_log.csv` so you can see exactly what was modified.
Does Not Modify Clean Data: If no corrections are needed, the script **does nothing** and exits safely.

Final Output

- **`bounds_df.csv`** (Corrected, if needed)
- **`bounds_check_log.csv`** (List of all applied corrections)

This **automates the manual checking process** and ensures **bounds data is always accurate** with minimal manual intervention.

Key Optimizations Applied - to establishing distances

Vectorized Operations:

- Used `.apply()` and `.map()` instead of row-wise loops for better performance.
  Memory Optimization:
- Removed redundant `.copy()` calls and used in-place modifications where applicable.
  Structured into Functions & Cells:
- Each **functional task** (unit extraction, missing coordinate filling, distance calculation) is handled in **separate cells**.
  Error Handling:
- Prevents **NaN issues** by ensuring calculations only run when valid data exists.

Key Enhancements

Fast & Scalable:

- Uses **vectorized `.apply()`**, ensuring **efficient calculations on large datasets**.

Handles Missing or Invalid Data Gracefully:

- **Avoids errors from missing distances or ratios** while logging issues for debugging.

Ensures State-Level Priority:

- **Checks state first**, falling back to `country_code` if unavailable.

Handles Multiple Distance Values:

- **Extracts the first valid numeric value** if `|`-separated entries exist.

Ensures Correct Output Format:

- Returns **rounded values (2 decimal places)** for consistency.

Logs Warnings for Debugging (Optional):

- Prints **alerts for invalid distance values** or missing conversion ratios.

Key Optimizations Applied - Bearing calculation

Vectorized Processing:

- Eliminates the **row-wise iteration (`iterrows()`)** and uses **efficient Pandas operations**.

Ensures Accuracy in Bearing Calculation:

- Uses **Geodesic calculation (GeographicLib)** instead of the Haversine approximation.

Handles Edge Cases Gracefully:

- Skips rows without valid preceding or following locations.

Better Readability & Maintainability:

- Structured function for clarity, with clear **step-by-step logic**.

Key Enhancements

Eliminated `iterrows()` for Speed:

- Uses **Pandas vectorized `.shift()`** to efficiently fetch preceding and following locations.

Replaces Haversine Approximation with Accurate GeographicLib Calculation:

- Uses **Geodesic WGS84 model** for precise azimuth (bearing).

Handles Missing Data Properly:

- Ensures **only valid locations are considered**, skipping missing values safely.

Significantly Faster Processing:

- Instead of iterating through **each row**, the function **computes bearing in one step** for the entire DataFrame.

More Readable & Maintainable:

- Uses **clear function definitions** and **drops unnecessary helper columns after calculations**.

Key Optimizations Applied - for approx coordinates

Key Enhancements

Eliminates Inefficient Loops:

- Uses **vectorized `.apply()`** instead of iterating through each row.

Uses Accurate Geodesic Distance Calculations:

- **Replaces Euclidean approximations** with the **Geodesic WGS84 model** for **precision**.

Handles Missing Values Properly:

- Skips calculations for rows **without bearing, distance, or previous location**.

Interpolates Prose Coordinates Efficiently:

- Ensures **prose entries without distances** are set **midway between the nearest valid locations**.

Fast, Scalable, and Robust:

- Can handle **large datasets without significant performance loss**.

Notes to Log for Debugging

Issue Type	Cause	Action Taken
⚠️ `No valid data found`	No rows have bearing + distance + route_description	Function exits safely
⚠️ `Missing values skipped`	Entries lack latitude, bearing, or distance	Function handles gracefully

⚠️ `Incorrect Coordinate Calculation`	Outliers or extreme distances detected	Uses `round()` for precision
⚠️ `Prose Midpoint Calculation`		

🔧 Key Optimizations & Fixes - for matching to alternative locs

Uses Vectorized Operations for Speed → Eliminates **`iterrows()`** by using Pandas **apply functions**.

Ensures Accurate Distance Calculations → Uses `geopy.distance.geodesic` correctly for **precise geospatial matching**.

Handles Missing or Invalid Coordinates Properly → **Skips NaN values** and **ensures numeric parsing** before processing.

Efficiently Finds the Closest Gazetteer Match → Uses **NumPy broadcasting** for **faster spatial distance lookups**.

Merges Approximate & Gazetteer Data Efficiently → Ensures **duplicate-free merging** and **preserves relevant attributes**.

Notes for Logs - Distance Tests

Purpose of Distance Tests

The script runs two distance validation checks to flag potential inconsistencies in location data:

1. `dist_test1`: Flags locations where the revised distance is more than 30 km from the previous known location.
2. `dist_test2`: Flags locations where the actual recorded coordinates are more than 30 km from the approximated coordinates.

Logging Notes for Debugging

Issue Type	Possible Cause	Action Taken
Missing `revised_distance`	Data entry issue or missing calculations	Skipped row safely
Invalid `approx_coordinates` Format	Corrupted or incorrectly formatted coordinates	Skipped row safely
`dist_test1` **Flagged**	The calculated distance from the previous location exceeds 30 km	Row is flagged for review
`dist_test2` **Flagged**	The actual coordinates are more than 30 km from the approximated coordinates	Row is flagged for review

Geodesic Calculation Error	Invalid latitude/longitude values in the dataset	Ensured numeric parsing before calculations

---

Expected Output in `full_test_df.csv`

- **Columns `dist_test1` & `dist_test2`**: Contain `True` for flagged locations, otherwise `False`.
- **Ensures Correct Processing**: **No NaN values in `dist_test` columns**, all missing values default to `False`.
- **Prepares Data for Review**: Flags discrepancies in **coordinate accuracy & distance consistency**.

Log Notes - Coordinate & State Tests

Purpose of Tests

The script runs a series of validation checks to detect anomalies in coordinate trends and state consistency:

1. **`dist_test1`** → Flags locations where the revised distance is more than **30 km** from the previous known location.
2. **`dist_test2`** → Flags locations where actual recorded coordinates are more than **30 km** from the approximated coordinates.
3. **`coords_test`** → Flags locations where **both latitude and longitude change trends** (increase/decrease flip).
4. **`state_test`** → Flags locations where the **state differs** from both the previous and next locations.
5. **`state_test2`** → Flags locations where the state does not **match any state occurring in the same route description**.

Logging Notes for Debugging

Issue Type	Possible Cause	Action Taken
Missing `revised_distance`	Data entry issue or missing calculations	Skipped row safely
Invalid `approx_coordinates` Format	Corrupted or incorrectly formatted coordinates	Skipped row safely
`dist_test1` Flagged	Distance from previous location exceeds 30 km	Row flagged for review
`dist_test2` Flagged	Actual coordinates differ more than 30 km from approximated coordinates	Row flagged for review
Geodesic Calculation Error	Invalid latitude/longitude values in the dataset	Ensured numeric parsing before calculations
`coords_test` Flagged	Latitude & longitude trend reverses (increase → decrease)	Row flagged for review
`state_test` Flagged	State differs from both previous and next locations	Row flagged for review
`state_test2` Flagged	State does not match any other in the same route	Row flagged for review

---

Final Output in `full_test_df.csv`

- **Columns `dist_test1`, `dist_test2`, `coords_test`, `state_test`, and `state_test2`** → Contain `True` for flagged locations, otherwise `False`.
- **Ensures Correct Processing** → **No NaN values in test columns**, all missing values default to `False`.
- **Prepares Data for Review** → Flags discrepancies in **coordinate trends, distance consistency, and state validity**.
- **Column `Automated_Flag`** → Set to `True` if **any of the test conditions are met**.

Final Log Summary

All flagged locations should be reviewed for potential data inconsistencies
Invalid coordinates or missing data were skipped without affecting calculations
Final dataset is exported to `full_test_df.csv` for further validation