Code Explanation

This program consists of two primary components:

- GUI design: Displays information such as latitude, longitude, and the total distance covered by each GPS tracker.
- FRP Logic: Uses FRP to handle real-time updates and processes GPS events as they arrive.

Data Structures:

TimestampedGpsEvent class: This serves as a wrapper for GpsEvent objects by attaching a timestamp to each event, allowing us to track when the event occurred. The timestamp is crucial for filtering out events older than 5 minutes.

trackerDataMap and *trackerDistanceMap*: These are used to store data for each tracker. *trackerDataMap* holds a deque of GPS events for each tracker, allowing us to maintain a sliding window of events. *trackerDistanceMap* stores the total distance travelled by each tracker, calculated from the deque of GPS events.

GUI:

The GUI is divided into two panels.

- Left panel: Displays the raw GPS data, including tracker number, latitude, and longitude for 10 trackers.
- Right panel: Displays filtered data based on user input, including the distance each tracker has travelled within the last 5 minutes.

Updating the tables:

The tables in the GUI are automatically updated when new GPS data is received. I use *DefaultTableModel* to store the data for the tables. Each tracker's latitude and longitude are updated using the *updateLeftTable()* method, while the filtered information and calculated distance are updated in the *updateRightTable()* method.

Filtering Logic:

The right table shows only the GPS data that falls within the latitude and longitude ranges set by the user. These ranges are inputted into text fields, and the values are parsed and processed by Sodium's *CellSink<Double>* objects.

When a new GPS event arrives, the application checks whether it falls within the user-defined latitude and longitude ranges. If it does, the filtered event is added to the deque of the tracker, and the total distance travelled is recalculated using the *updateTotalDistance()* method.

Distance Calculation:

The distance between consecutive GPS points is calculated using the latitude, longitude, and altitude differences. The application uses a simple Pythagorean theorem to compute the straight-line distance between two GPS points, which is then summed up to calculate the total distance traveled by the tracker within the last 5 minutes.

Timer for resetting events:

The *resetEventTimer()* function resets the event label to display "No Events" after 3 seconds, unless a new event is received. This gives the user a clear indication of when the last event occurred.

Sodium FRP Usage:

Sodium FRP is used to handle the real-time updates in the GUI.

- *CellSink<Double>*: These hold the minimum and maximum latitude and longitude values set by the user and are updated whenever the user types into the corresponding text fields.
- *StreamSink<Unit>*: This is triggered when the user clicks the "Set" button, resetting the right table and applying the filter.

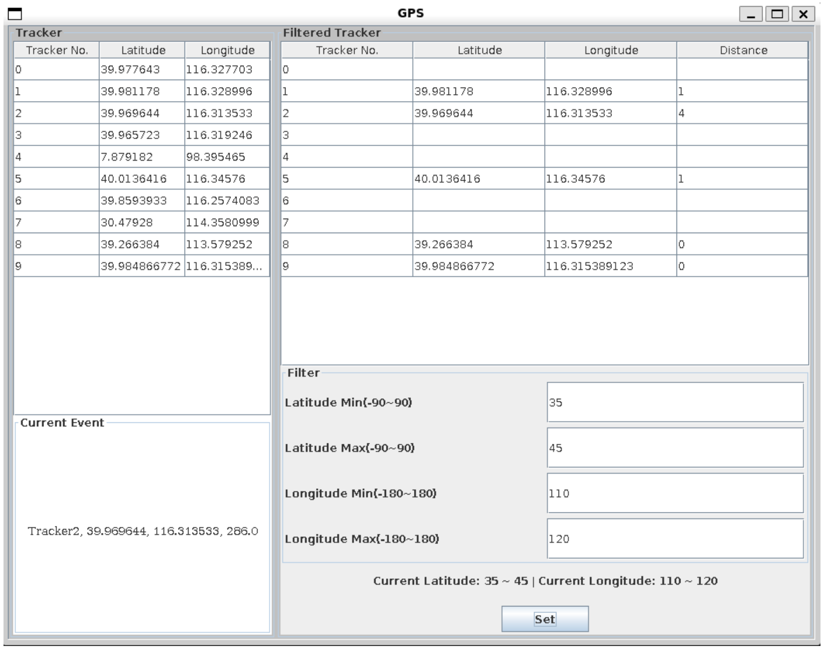
Test Explanation

The test suite consists of three primary test cases:

- *testCalculateDistance*: This test verifies the correct computation of the distance between two GPS events using the *calculateDistance()* method.
- *testUpdateTotalDistance*: This test ensures that the total distance traveled by a tracker is correctly calculated and updated in the GUI's right table using the *updateTotalDistance()* method.
- *testUpdateLeftTable*: This test confirms that the latitude and longitude values for a specific tracker are correctly updated in the GUI's left table using the *updateLeftTable()* method.

1. Overall GUI Design

I. GUI Layout and Components



GPS

Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.981178	116.328996
2	39.969644	116.313533
3	39.965723	116.319246
4	7.879182	98.395465
5	40.0136416	116.34576
6	39.8593933	116.2574083
7	30.47928	114.3580999
8	39.266384	113.579252
9	39.984866772	116.315389...

Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0			
1	39.981178	116.328996	1
2	39.969644	116.313533	4
3			
4			
5	40.0136416	116.34576	1
6			
7			
8	39.266384	113.579252	0
9	39.984866772	116.315389123	0

Current Event

Tracker2, 39.969644, 116.313533, 286.0

Filter

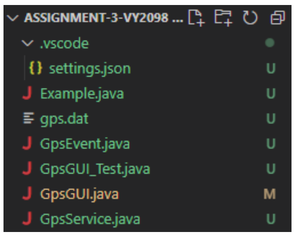
Latitude Min(-90~90)	35
Latitude Max(-90~90)	45
Longitude Min(-180~180)	110
Longitude Max(-180~180)	120

Current Latitude: 35 ~ 45 | Current Longitude: 110 ~ 120

This image shows the GUI layout and components of the program.

2. Testing range and report

I. Instruction to run the program



This image shows the directory structure of the project.

To run this program, use the following command to compile the Java files, ensuring the sodium-with-time.jar is included as a classpath dependency.

```
javac -cp "./sodium-with-time.jar" GpsService.java GpsEvent.java GpsGUI.java
```

Once the files are successfully compiled, run the program by executing the GpsGUI class using the command below:

```
java -cp ".:./sodium-with-time.jar" GpsGUI
```

This command sets the classpath to the current directory (.) and the sodium-with-time.jar file, then launches the GpsGUI class.

To run the test program, use the following command to compile the Java files, ensure that the sodium-with-time.jar, junit-4.13.2.jar, mockito-core-3.6.0.jar, hamcrest-core-1.3.jar, and other necessary dependencies are included in the classpath.

```
javac -cp ".:*" GpsGUI_Test.java
```

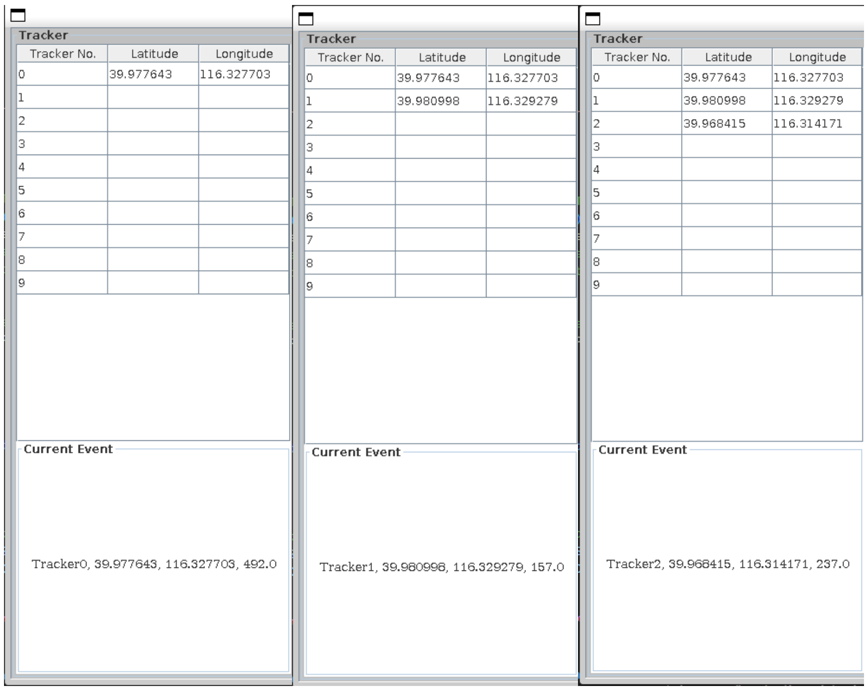
This command compiles the Java files (GpsService.java, GpsEvent.java, GpsGUI.java, and GpsGUI_Test.java) with all the necessary dependencies.

Once the files are successfully compiled, run the test class using the following command:

```
java -cp ".:*" org.junit.runner.JUnitCore GpsGUI_Test
```

This command sets the classpath to the current directory (.) and includes all the JAR dependencies required to run the JUnit test, then it launches the *GpsGUI_Test* class using the JUnit test runner.

II. Ten Tracker Simplified Display



Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1		
2		
3		
4		
5		
6		
7		
8		
9		

Current Event

Tracker0, 39.977643, 116.327703, 492.0

Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.980998	116.329279
2		
3		
4		
5		
6		
7		
8		
9		

Current Event

Tracker1, 39.980998, 116.329279, 157.0

Tracker

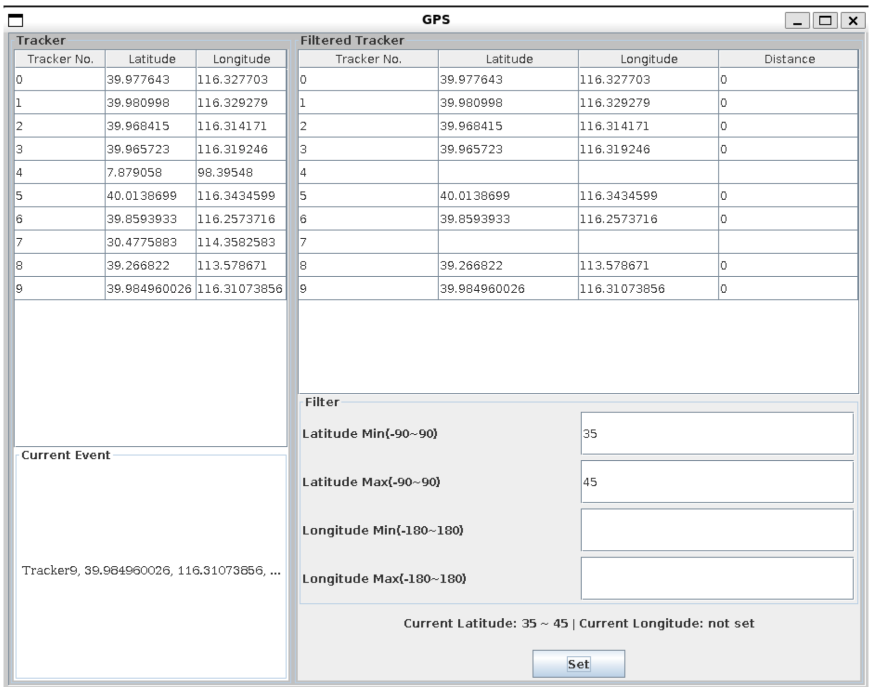
Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.980998	116.329279
2	39.968415	116.314171
3		
4		
5		
6		
7		
8		
9		

Current Event

Tracker2, 39.968415, 116.314171, 237.0

The images show the simplified display for 10 trackers, with correct latitude and longitude values. Altitude is removed. Trackers 0, 1, and 2 are progressively displayed in the GUI with delay by modifying GpsService settings.

III. Filter Display by Latitude and Longitude



GPS

Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.980998	116.329279
2	39.968415	116.314171
3	39.965723	116.319246
4	7.879058	98.39548
5	40.0138699	116.3434599
6	39.8593933	116.2573716
7	30.4775883	114.3582583
8	39.266822	113.578671
9	39.984960026	116.31073856

Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0	39.977643	116.327703	0
1	39.980998	116.329279	0
2	39.968415	116.314171	0
3	39.965723	116.319246	0
4			
5	40.0138699	116.3434599	0
6	39.8593933	116.2573716	0
7			
8	39.266822	113.578671	0
9	39.984960026	116.31073856	0

Current Event

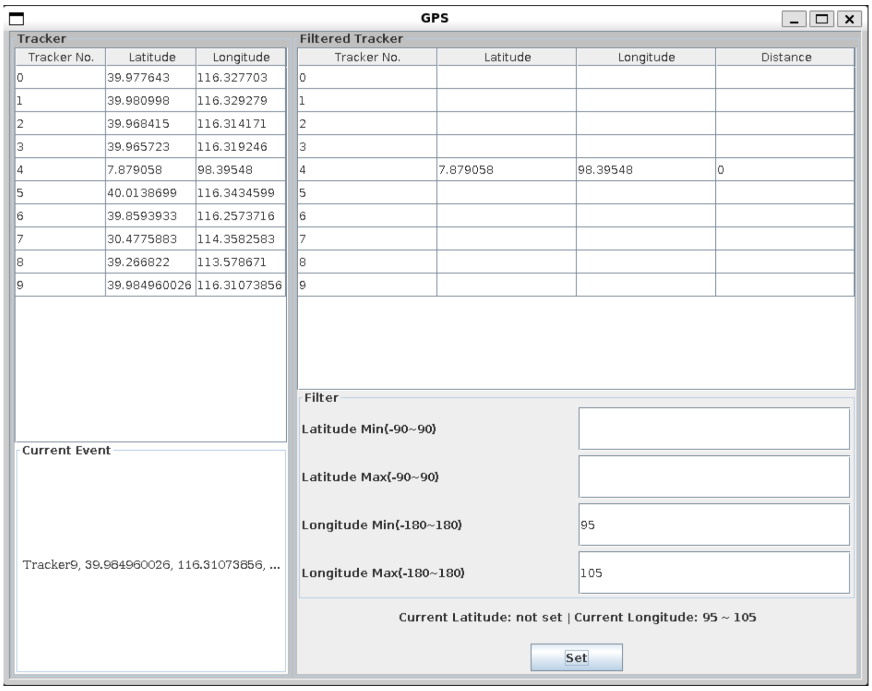
Tracker9, 39.984960026, 116.31073856, ...

Filter

Latitude Min(-90~90)	35
Latitude Max(-90~90)	45
Longitude Min(-180~180)	
Longitude Max(-180~180)	

Current Latitude: 35 ~ 45 | Current Longitude: not set

The latitude filter is set to 35~45, showing only trackers within that latitude range.



GPS

Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.980998	116.329279
2	39.968415	116.314171
3	39.965723	116.319246
4	7.879058	98.39548
5	40.0138699	116.3434599
6	39.8593933	116.2573716
7	30.4775883	114.3582583
8	39.266822	113.578671
9	39.984960026	116.31073856

Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0			
1			
2			
3			
4	7.879058	98.39548	0
5			
6			
7			
8			
9			

Current Event

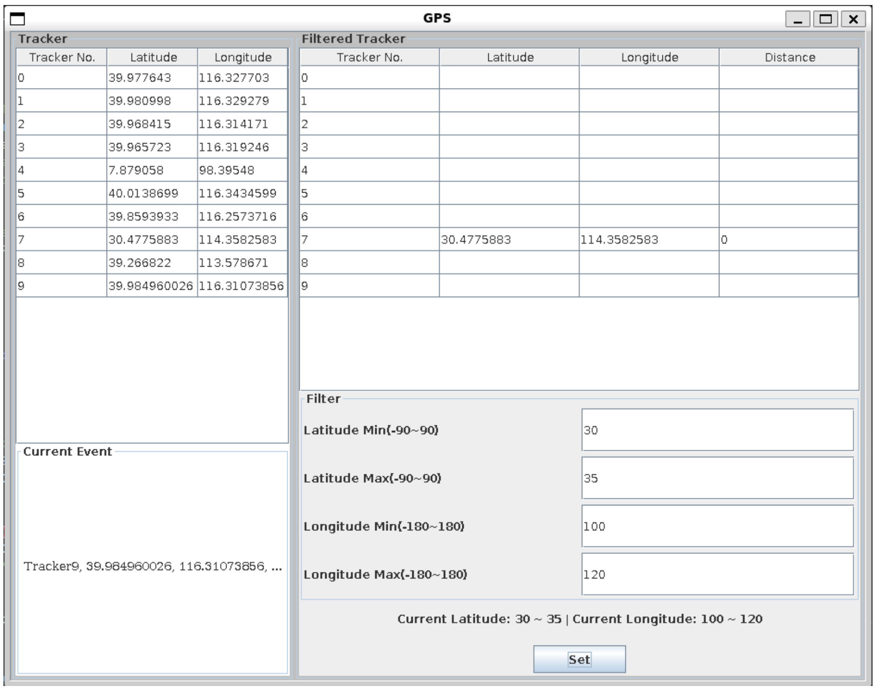
Tracker9, 39.984960026, 116.31073856, ...

Filter

Latitude Min(-90~90)	
Latitude Max(-90~90)	
Longitude Min(-180~180)	95
Longitude Max(-180~180)	105

Current Latitude: not set | Current Longitude: 95 ~ 105

The longitude filter is set to 95~105, showing only trackers within that longitude range.


GPS

Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.980998	116.329279
2	39.968415	116.314171
3	39.965723	116.319246
4	7.879058	98.39548
5	40.0138699	116.3434599
6	39.8593933	116.2573716
7	30.4775883	114.3582583
8	39.266822	113.578671
9	39.984960026	116.31073856

Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0			
1			
2			
3			
4			
5			
6			
7	30.4775883	114.3582583	0
8			
9			

Current Event

Tracker9, 39.984960026, 116.31073856, ...

Filter

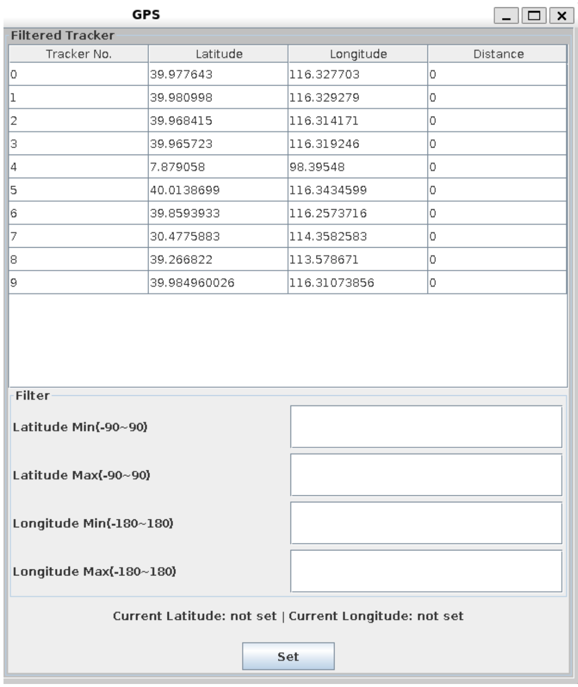
Latitude Min(-90~90)	30
Latitude Max(-90~90)	35
Longitude Min(-180~180)	100
Longitude Max(-180~180)	120

Current Latitude: 30 ~ 35 | Current Longitude: 100 ~ 120

Both latitude and longitude filters are applied (latitude: 30~35, longitude: 100~120), displaying only trackers within both ranges.

The filtered trackers are shown in the "Filtered Tracker" section, while the "Tracker" section continues to show all trackers. The application correctly hides or displays trackers based on the user-specified filter values.

IV. Distance Calculation for Each Tracker


GPS

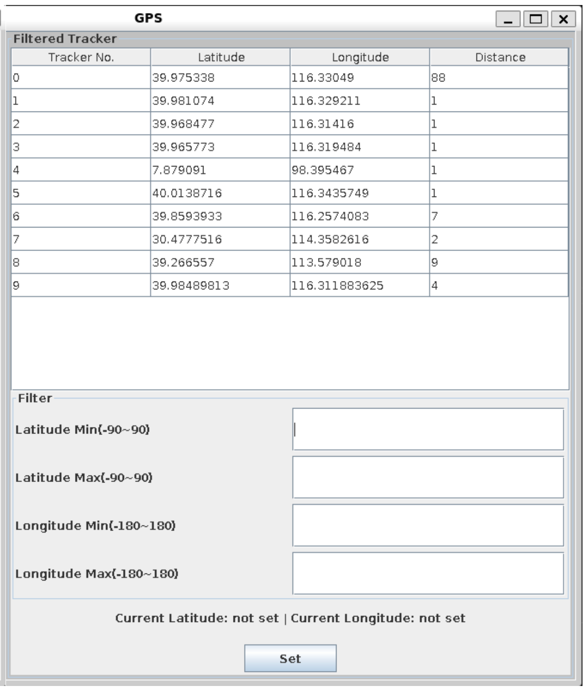
Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0	39.977643	116.327703	0
1	39.980998	116.329279	0
2	39.968415	116.314171	0
3	39.965723	116.319246	0
4	7.879058	98.39548	0
5	40.0138699	116.3434599	0
6	39.8593933	116.2573716	0
7	30.4775883	114.3582583	0
8	39.266822	113.578671	0
9	39.984960026	116.31073856	0

Filter

Latitude Min(-90~90)	
Latitude Max(-90~90)	
Longitude Min(-180~180)	
Longitude Max(-180~180)	

Current Latitude: not set | Current Longitude: not set


GPS

Filtered Tracker

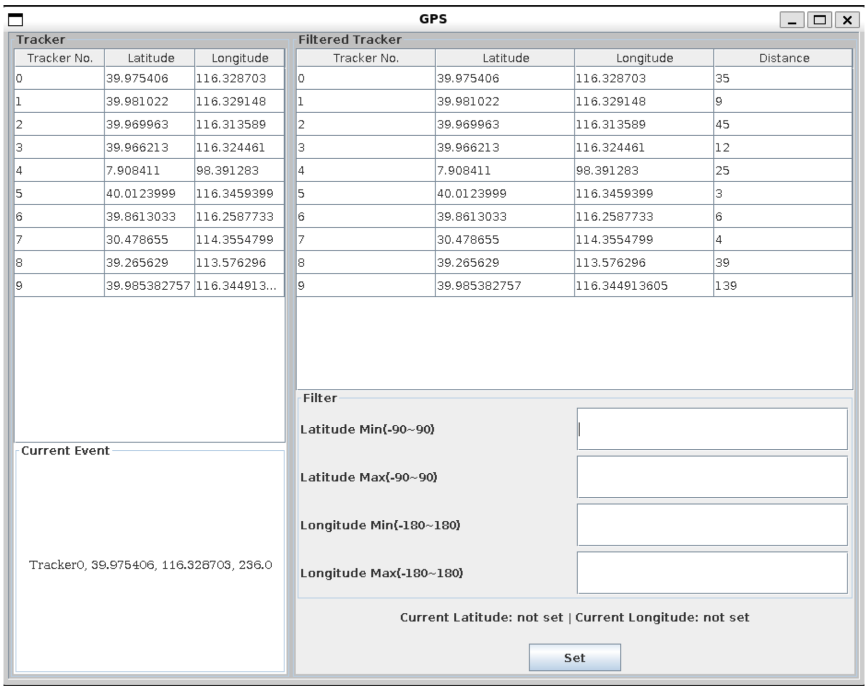
Tracker No.	Latitude	Longitude	Distance
0	39.975338	116.33049	88
1	39.981074	116.329211	1
2	39.968477	116.31416	1
3	39.965773	116.319484	1
4	7.879091	98.395467	1
5	40.0138716	116.3435749	1
6	39.8593933	116.2574083	7
7	30.4777516	114.3582616	2
8	39.266557	113.579018	9
9	39.98489813	116.311883625	4

Filter

Latitude Min(-90~90)	
Latitude Max(-90~90)	
Longitude Min(-180~180)	
Longitude Max(-180~180)	

Current Latitude: not set | Current Longitude: not set

Initially, the distance for all trackers is set to 0. Then Tracker 1 has moved a long distance, causing its distance to increase significantly compared to the others.


GPS

Tracker

Tracker No.	Latitude	Longitude
0	39.975406	116.328703
1	39.981022	116.329148
2	39.969963	116.313589
3	39.966213	116.324461
4	7.908411	98.391283
5	40.0123999	116.3459399
6	39.8613033	116.2587733
7	30.478655	114.3554799
8	39.265629	113.576296
9	39.985382757	116.344913...

Current Event

Tracker0, 39.975406, 116.328703, 236.0

Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0	39.975406	116.328703	35
1	39.981022	116.329148	9
2	39.969963	116.313589	45
3	39.966213	116.324461	12
4	7.908411	98.391283	25
5	40.0123999	116.3459399	3
6	39.8613033	116.2587733	6
7	30.478655	114.3554799	4
8	39.265629	113.576296	39
9	39.985382757	116.344913605	139

Filter

Latitude Min(-90~90):

Latitude Max(-90~90):

Longitude Min(-180~180):

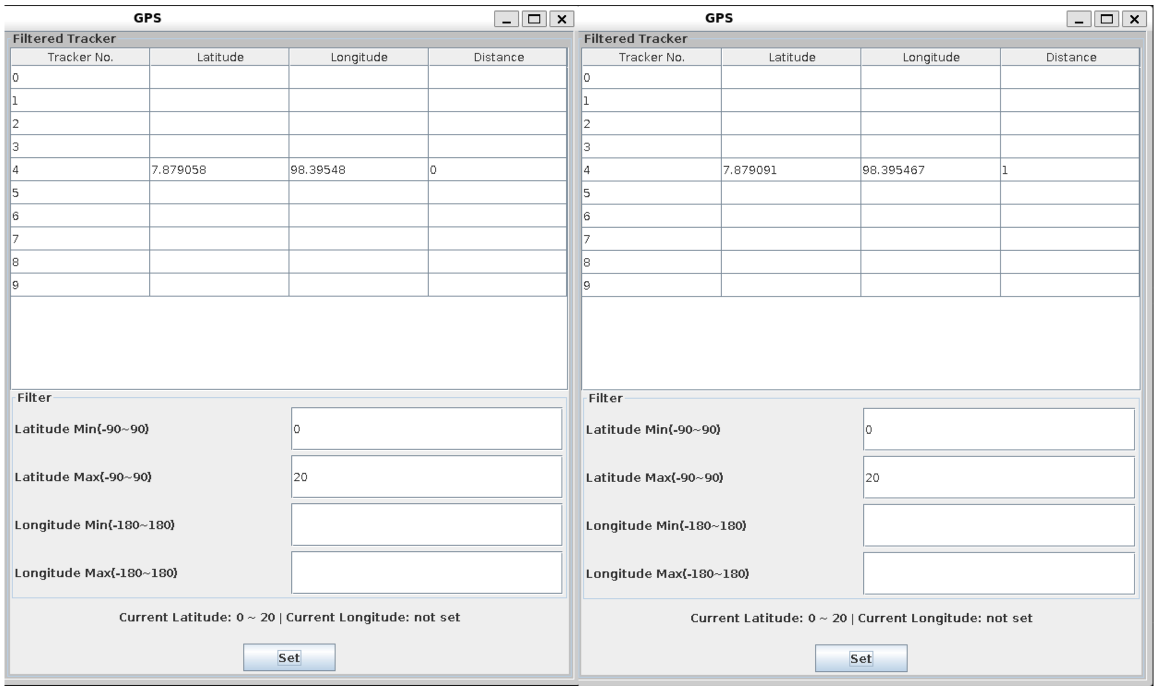
Longitude Max(-180~180):

Current Latitude: not set | Current Longitude: not set

Set

After 5 minutes, the window slides, and the distance for Tracker 1 decreases as earlier movement falls outside the 5-minute window, showing the dynamic update of distance calculations over time.

V. Distance Calculation with Filter


GPS

Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0			
1			
2			
3			
4	7.879058	98.39548	0
5			
6			
7			
8			
9			

Filter

Latitude Min(-90~90): 0

Latitude Max(-90~90): 20

Longitude Min(-180~180):

Longitude Max(-180~180):

Current Latitude: 0 ~ 20 | Current Longitude: not set

Set

GPS

Filtered Tracker

Tracker No.	Latitude	Longitude	Distance
0			
1			
2			
3			
4	7.879091	98.395467	1
5			
6			
7			
8			
9			

Filter

Latitude Min(-90~90): 0

Latitude Max(-90~90): 20

Longitude Min(-180~180):

Longitude Max(-180~180):

Current Latitude: 0 ~ 20 | Current Longitude: not set

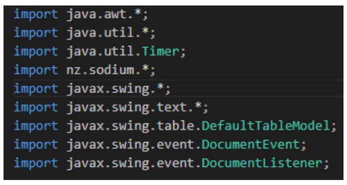
Set

Initially, Tracker 4 is displayed based on the filter, with a latitude of 7.879058 and longitude of 98.39548. The distance is set to 0. After an event occurs, Tracker 4's position is updated, and the distance is now calculated as 1.

The program ensures that only trackers falling within the latitude filter range are used when calculating distances.

3. FRP Transformations

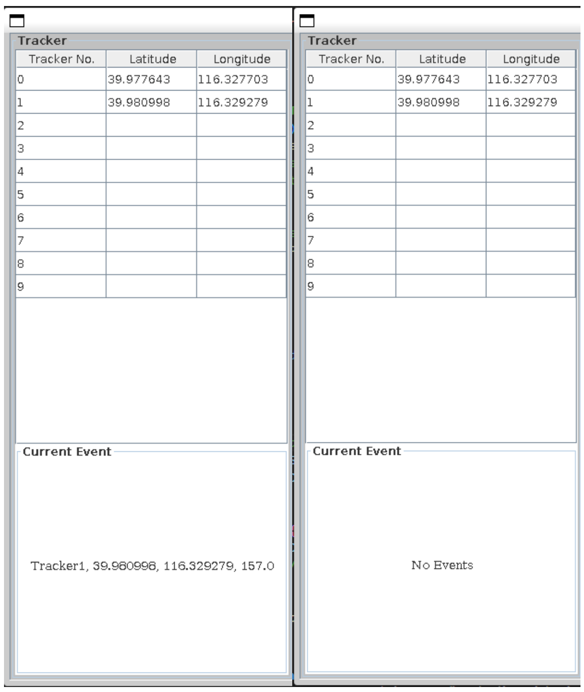
I. Usage of Sodium and Swidget



```
import java.awt.*;
import java.util.*;
import java.util.Timer;
import nz.sodium.*;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.table.DefaultTableModel;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
```

In this program, Sodium has been used to handle reactive event streams for frequent GPS updates. Although Swidget was not used, the GUI was developed using Java Swing, which integrates with Sodium to maintain responsiveness. Swing components are updated in real-time as new GPS events are processed, demonstrating FRP principles while ensuring the user interface remains interactive and responsive.

II. Most Recent Event Display



Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.980998	116.329279
2		
3		
4		
5		
6		
7		
8		
9		

Current Event

Tracker1, 39.980998, 116.329279, 157.0

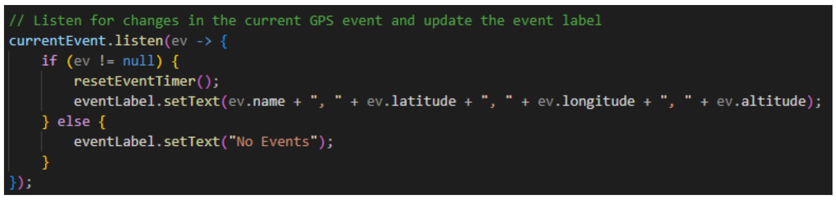
Tracker

Tracker No.	Latitude	Longitude
0	39.977643	116.327703
1	39.980998	116.329279
2		
3		
4		
5		
6		
7		
8		
9		

Current Event

No Events

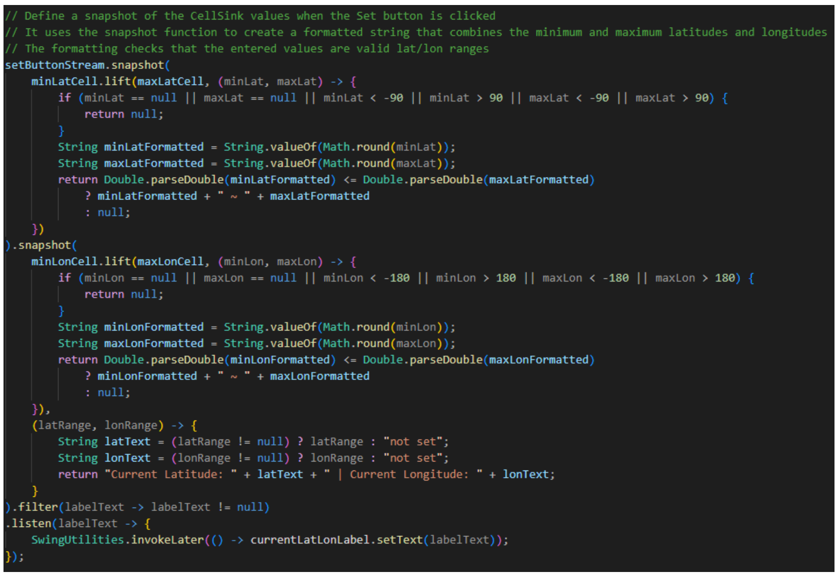
When a new GPS event is received, the event details are displayed. After 3 seconds without new event received, the program changes back to “No Events” in the "Current Event" section.



```
// Listen for changes in the current GPS event and update the event label
currentEvent.listen(ev -> {
    if (ev != null) {
        resetEventTimer();
        eventLabel.setText(ev.name + ", " + ev.latitude + ", " + ev.longitude + ", " + ev.altitude);
    } else {
        eventLabel.setText("No Events");
    }
});
```

This code snippet demonstrates the functionality for displaying the most recent GPS event and clearing it after 3 seconds of inactivity

III. Filter Display by Latitude and Longitude

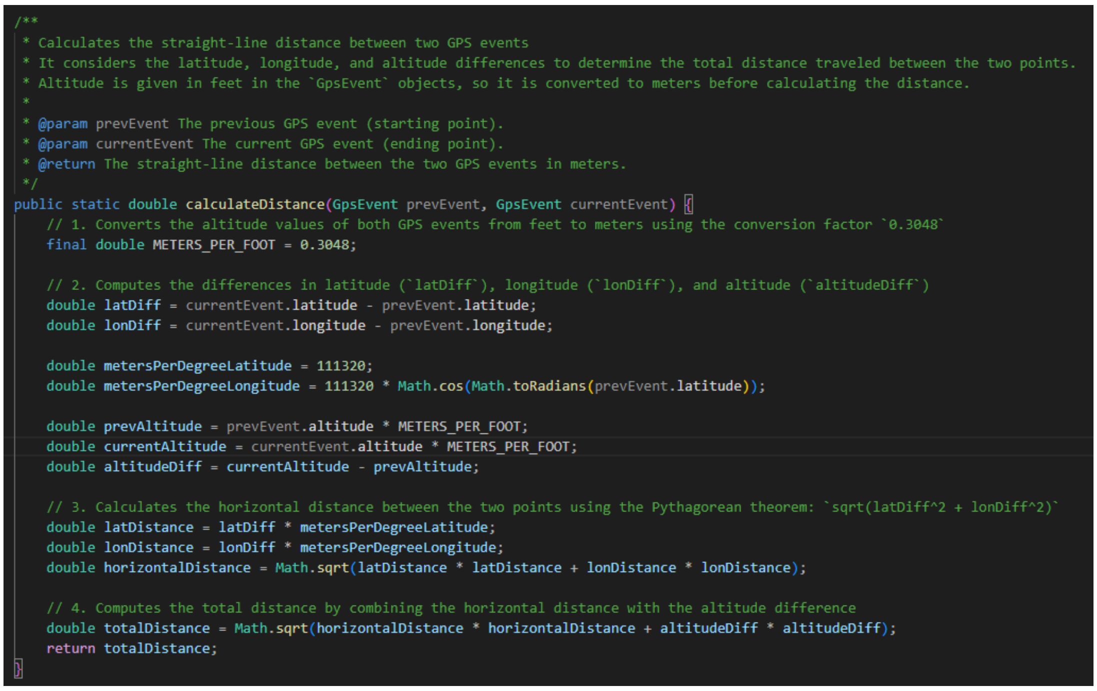


```java
// Define a snapshot of the CellSink values when the Set button is clicked
// It uses the snapshot function to create a formatted string that combines the minimum and maximum latitudes and longitudes
// The formatting checks that the entered values are valid lat/lon ranges
setButtonStream.snapshot(
    minLatCell.lift(maxLatCell, (minLat, maxLat) -> {
        if (minLat == null || maxLat == null || minLat < -90 || minLat > 90 || maxLat < -90 || maxLat > 90) {
            return null;
        }
        String minLatFormatted = String.valueOf(Math.round(minLat));
        String maxLatFormatted = String.valueOf(Math.round(maxLat));
        return Double.parseDouble(minLatFormatted) <= Double.parseDouble(maxLatFormatted)
            ? minLatFormatted + " ~ " + maxLatFormatted
            : null;
    })
).snapshot(
    minLonCell.lift(maxLonCell, (minLon, maxLon) -> {
        if (minLon == null || maxLon == null || minLon < -180 || minLon > 180 || maxLon < -180 || maxLon > 180) {
            return null;
        }
        String minLonFormatted = String.valueOf(Math.round(minLon));
        String maxLonFormatted = String.valueOf(Math.round(maxLon));
        return Double.parseDouble(minLonFormatted) <= Double.parseDouble(maxLonFormatted)
            ? minLonFormatted + " ~ " + maxLonFormatted
            : null;
    }),
    (latRange, lonRange) -> {
        String latText = (latRange != null) ? latRange : "not set";
        String lonText = (lonRange != null) ? lonRange : "not set";
        return "Current Latitude: " + latText + " | Current Longitude: " + lonText;
    }
).filter(labelText -> labelText != null)
.listen(labelText -> {
    SwingUtilities.invokeLater(() -> currentLatLonLabel.setText(labelText));
});
```

The image shows the code responsible for applying latitude and longitude filters using a "Set" button in the GUI. When the user clicks the "Set" button, a snapshot of the entered latitude and longitude ranges is taken. The code ensures the input values are valid. If valid, the current latitude and longitude ranges are formatted and displayed on the GUI, and the tracker data is filtered accordingly.

This process ensures accurate filtering, as shown in the GUI in the earlier part, where only trackers within the set ranges are shown.

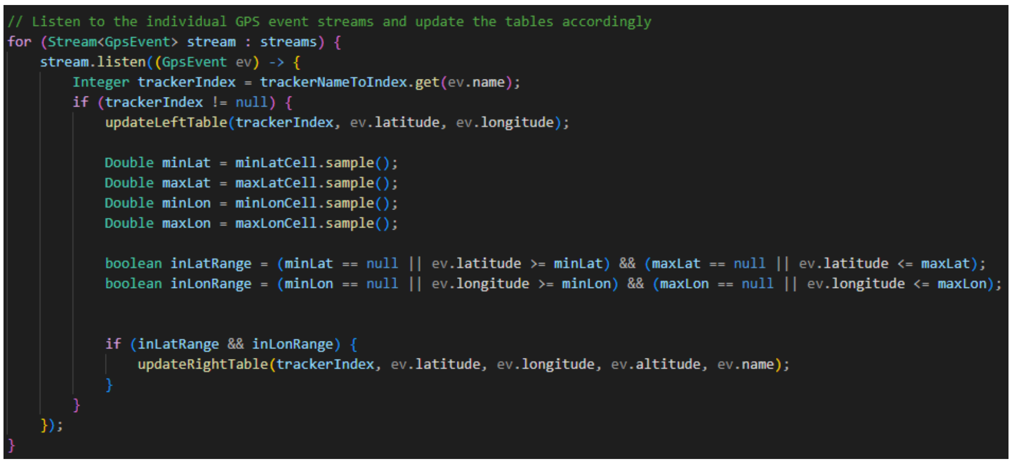
IV. Distance Calculation Tests:



```java
/**
 * Calculates the straight-line distance between two GPS events
 * It considers the latitude, longitude, and altitude differences to determine the total distance traveled between the two points.
 * Altitude is given in feet in the `GpsEvent` objects, so it is converted to meters before calculating the distance.
 *
 * @param prevEvent The previous GPS event (starting point).
 * @param currentEvent The current GPS event (ending point).
 * @return The straight-line distance between the two GPS events in meters.
 */
public static double calculateDistance(GpsEvent prevEvent, GpsEvent currentEvent) {
    // 1. Converts the altitude values of both GPS events from feet to meters using the conversion factor `0.3048`
    final double METERS_PER_FOOT = 0.3048;

    // 2. Computes the differences in latitude (`latDiff`), longitude (`lonDiff`), and altitude (`altitudeDiff`)
    double latDiff = currentEvent.latitude - prevEvent.latitude;
    double lonDiff = currentEvent.longitude - prevEvent.longitude;

    double metersPerDegreeLatitude = 111320;
    double metersPerDegreeLongitude = 111320 * Math.cos(Math.toRadians(prevEvent.latitude));

    double prevAltitude = prevEvent.altitude * METERS_PER_FOOT;
    double currentAltitude = currentEvent.altitude * METERS_PER_FOOT;
    double altitudeDiff = currentAltitude - prevAltitude;

    // 3. Calculates the horizontal distance between the two points using the Pythagorean theorem: `sqrt(latDiff^2 + lonDiff^2)`
    double latDistance = latDiff * metersPerDegreeLatitude;
    double lonDistance = lonDiff * metersPerDegreeLongitude;
    double horizontalDistance = Math.sqrt(latDistance * latDistance + lonDistance * lonDistance);

    // 4. Computes the total distance by combining the horizontal distance with the altitude difference
    double totalDistance = Math.sqrt(horizontalDistance * horizontalDistance + altitudeDiff * altitudeDiff);
    return totalDistance;
}
```

The code provided calculates the straight-line distance between two GPS events, considering latitude, longitude, and altitude differences. Altitudes are converted from feet to meters using a conversion factor (0.3048 meters per foot). The latitude and longitude differences are converted to distances using a fixed approximation for degrees of latitude and longitude. The horizontal distance between the two points is calculated using the Pythagorean theorem, and the total distance is determined by combining this horizontal distance with the altitude difference, also using the Pythagorean theorem.

V. Altitude Removal



```
// Listen to the individual GPS event streams and update the tables accordingly
for (Stream<GpsEvent> stream : streams) {
    stream.listen((GpsEvent ev) -> {
        Integer trackerIndex = trackerNameToIndex.get(ev.name);
        if (trackerIndex != null) {
            updateLeftTable(trackerIndex, ev.latitude, ev.longitude);

            Double minLat = minLatCell.sample();
            Double maxLat = maxLatCell.sample();
            Double minLon = minLonCell.sample();
            Double maxLon = maxLonCell.sample();

            boolean inLatRange = (minLat == null || ev.latitude >= minLat) && (maxLat == null || ev.latitude <= maxLat);
            boolean inLonRange = (minLon == null || ev.longitude >= minLon) && (maxLon == null || ev.longitude <= maxLon);


            if (inLatRange && inLonRange) {
                updateRightTable(trackerIndex, ev.latitude, ev.longitude, ev.altitude, ev.name);
            }
        }
    });
}
```

The code provided demonstrates how FRP is used to update the tracker tables in real-time as GPS events are received. The altitude data is selectively removed from the simplified tracker display while being retained for distance calculations.

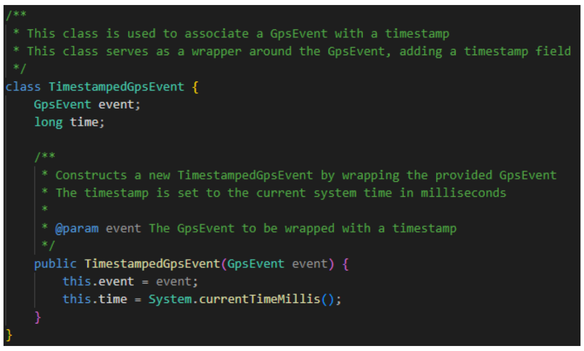
The *updateLeftTable()* method is used to update the simplified display, showing only the latitude and longitude values. This method does not include altitude in its parameters, ensuring that the altitude data is excluded from the simplified table.

The *updateRightTable()* method includes altitude as a parameter because it is necessary for calculating distances between GPS events. However, altitude is not displayed either.

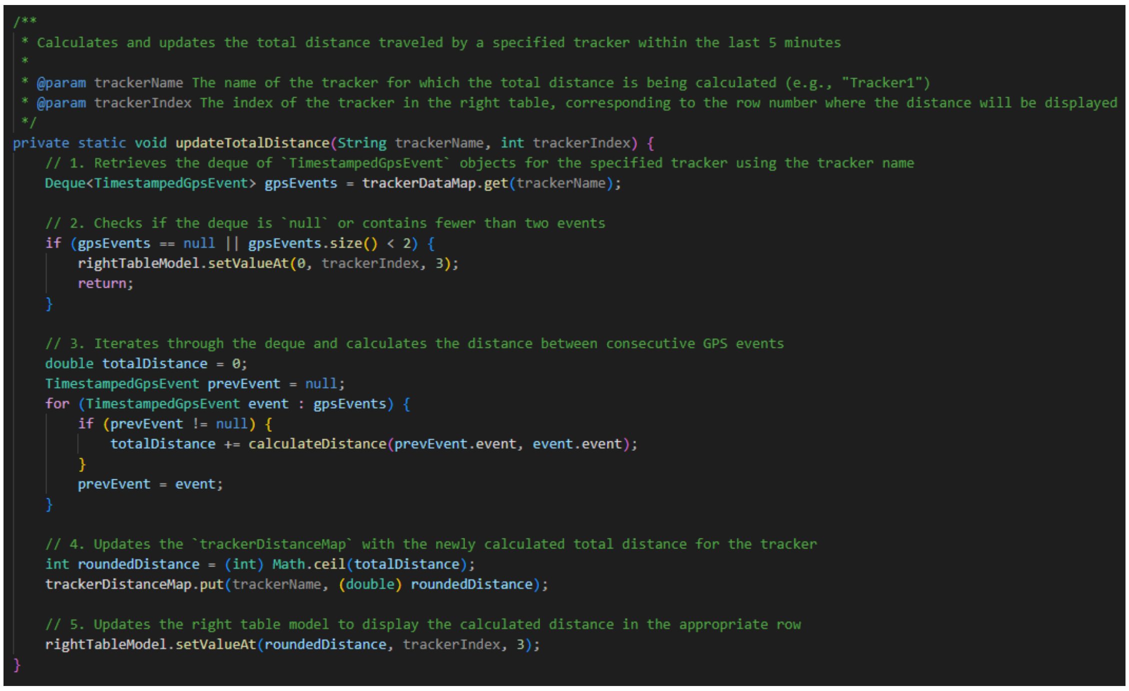
VI. Altitude Conversion

As explained in **IV. Distance Calculation Tests**, the altitude is provided in feet in the GPS event data and is converted to meters for accurate distance calculations. This is achieved by multiplying the altitude by the conversion factor 0.3048 meters per foot.

VII. Distance Calculation



```
/**
 * This class is used to associate a GpsEvent with a timestamp
 * This class serves as a wrapper around the GpsEvent, adding a timestamp field
 */
class TimestampedGpsEvent {
    GpsEvent event;
    long time;

    /**
     * Constructs a new TimestampedGpsEvent by wrapping the provided GpsEvent
     * The timestamp is set to the current system time in milliseconds
     *
     * @param event The GpsEvent to be wrapped with a timestamp
     */
    public TimestampedGpsEvent(GpsEvent event) {
        this.event = event;
        this.time = System.currentTimeMillis();
    }
}
```



```java
/**
 * Calculates and updates the total distance traveled by a specified tracker within the last 5 minutes
 *
 * @param trackerName The name of the tracker for which the total distance is being calculated (e.g., "Tracker1")
 * @param trackerIndex The index of the tracker in the right table, corresponding to the row number where the distance will be displayed
 */
private static void updateTotalDistance(String trackerName, int trackerIndex) {
    // 1. Retrieves the deque of `TimestampedGpsEvent` objects for the specified tracker using the tracker name
    Deque<TimestampedGpsEvent> gpsEvents = trackerDataMap.get(trackerName);

    // 2. Checks if the deque is `null` or contains fewer than two events
    if (gpsEvents == null || gpsEvents.size() < 2) {
        rightTableModel.setValueAt(0, trackerIndex, 3);
        return;
    }

    // 3. Iterates through the deque and calculates the distance between consecutive GPS events
    double totalDistance = 0;
    TimestampedGpsEvent prevEvent = null;
    for (TimestampedGpsEvent event : gpsEvents) {
        if (prevEvent != null) {
            totalDistance += calculateDistance(prevEvent.event, event.event);
        }
        prevEvent = event;
    }

    // 4. Updates the `trackerDistanceMap` with the newly calculated total distance for the tracker
    int roundedDistance = (int) Math.ceil(totalDistance);
    trackerDistanceMap.put(trackerName, (double) roundedDistance);

    // 5. Updates the right table model to display the calculated distance in the appropriate row
    rightTableModel.setValueAt(roundedDistance, trackerIndex, 3);
}
```

The images illustrate the distance calculation process using FRP primitives for each tracker over the last 5 minutes:

TimestampedGpsEvent Class: A new class created to associate a GPS event with a timestamp. It wraps around a *GpsEvent* object and adds a time field (in milliseconds), allowing tracking of when each GPS event occurs.

updateTotalDistance Method: This method calculates the total distance travelled by a tracker within the last 5 minutes:

A deque is used to store *TimestampedGpsEvent* objects for each tracker.

The method checks if there are at least two events to calculate the distance. It iterates through the deque, calculating the straight-line distance between consecutive GPS events using the *calculateDistance* method (explained previously). The total distance is updated and displayed in the right table, corresponding to the tracker's row.