INTEL-UNNATI AI&ML JOURNAL
Number 2 (2024), pp. 1–16

Team : Terabyte Titans

GPS Toll based System Simulation using `Python`

Sethunath A , Marianna Martin , Rizia Sara Prabin , and Sreya Anna Joseph

Saintgits Group of Institutions, Kottayam, Kerala

Abstract: This project delves into the innovative development and simulation of a GPS-based toll system using Python, poised to revolutionize traditional toll collection methodologies. Harnessing the power of GPS technology, the system adeptly tracks vehicle movements and seamlessly calculates toll charges based on predefined routes and strategically placed toll points. The simulation model brilliantly integrates real-time GPS data processing, dynamic route mapping, and sophisticated toll computation algorithms. Leveraging Python's robust libraries for geospatial data handling and simulation, such as Geopandas,Geopy, Pandas, and Folium, this endeavor showcases the remarkable feasibility and efficiency of automated toll collection. The simulation results illuminate the potential to alleviate traffic congestion at toll booths, minimize human error, and significantly enhance the overall user experience in toll payment systems. This project stands as a foundational framework, paving the way for future advancements and refinements in smart transportation infrastructure.

Keywords: GPS Tracking, Real Time Data Processing, Electronic Toll Collection, Location-Based Services(LBS), GPS Data Parsing, Map integration, Distance calculation

1 Introduction

In recent years, advancements in technology have significantly transformed various sectors, including transportation. One such innovation is the implementation of GPS-based toll systems, which offer a more efficient and streamlined method of toll collection compared to traditional toll booths. This project report delves into the development and simulation of a GPS toll system using Python, a powerful and versatile programming language.

The primary objective of this project is to simulate the functionality of a GPS toll system using Python. This includes designing algorithms for real-time vehicle tracking, route map-

ping, and toll calculation. The simulation incorporates various Python libraries such as Geopy,Geopandas, Folium, and Shapely. These tools collectively enable the creation of a sophisticated and efficient toll collection system, demonstrating the practical application of real-time GPS tracking and geospatial data processing.

By defining geofenced toll zones and calculating charges based on the distance traveled within these zones, the system eliminates the need for physical payment at toll booths, thereby reducing traffic congestion and operational costs. We use leaflet to generate a list of coordinates for the road network .We use polygon to define a toll region. We simulate the vehicle by making it pass through the set of coordinates using a for loop.GeoPandas and Shapely are used for handling and analyzing geospatial data. Geopy facilitates geocoding and geospatial computations, and Folium is utilized for interactive data visualization on maps.

This project aims to showcase the feasibility and advantages of a GPS toll system through a comprehensive simulation. The report details the system architecture, the implementation of toll calculation algorithms, and the integration of various Python libraries to manage and visualize GPS data effectively. By highlighting the system's efficiency and accuracy, this introduction sets the stage for a detailed exploration of the GPS toll system simulation, demonstrating the potential of combining Python with advanced geospatial libraries to innovate toll collection methods

2 Libraries Used

In the project for various tasks, following packages are used.

```
Geopy
Geopandas
Pandas
Shapely
Folium
Email
Tracemalloc
Flask
Datetime
csv
Time
```

3 Literature Review

The advancement of GPS technology and computational simulations has opened new avenues for transportation management, specifically in toll-based systems. This literature review explores the theoretical frameworks, concepts, and prior research in the domain of GPS toll-based system simulations, particularly those employing Python for their development and execution.

Theoretical Framework and Concepts :

1. GPS Technology in Toll Systems: GPS technology has been increasingly utilized for vehicle tracking and toll collection, offering precision and efficiency over traditional toll

booths. GPS-enabled systems can dynamically calculate tolls based on distance traveled and routes taken, which aligns with the principles of Intelligent Transportation Systems (ITS) .

2. Python for Simulation: Python's popularity in scientific computing is due to its simplicity and the robust ecosystem of libraries such as NumPy, SciPy, and SimPy, which are crucial for developing and running simulations. Python also facilitates data analysis and visualization through libraries like Pandas and Matplotlib, making it a powerful tool for simulating toll systems .

Review of Previous Studies :

1. Lee et al. (2007): Lee and colleagues explored the feasibility of GPS-based toll systems by developing a prototype that used GPS data for toll calculation. Their study demonstrated that GPS technology could effectively manage toll collection and provided insights into system design challenges.

3. Bulanon et al. (2016): Bulanon and colleagues investigated the optimization of toll collection points using GPS data. They developed an algorithm to minimize the overall cost for drivers while maximizing revenue for toll authorities. Their study demonstrated how optimization techniques can improve the efficiency of GPS-based toll systems.

Gaps in the Literature : Despite the extensive research on GPS-based toll systems and simulation methodologies, there are notable gaps:

1. Integration of Real-Time Data: Many existing simulations lack the integration of real-time traffic and toll data, which is crucial for dynamic toll pricing and adaptive traffic management. Future research should focus on incorporating live data streams to enhance the accuracy and responsiveness of the simulations.

2. Scalability and Efficiency: While Python offers numerous advantages, the scalability and computational efficiency of large-scale simulations remain challenging. Studies should investigate optimization techniques and parallel computing frameworks to address these issues, ensuring that simulations can handle extensive transportation networks.

4 System Study

Using advanced computer vision technology, modern toll systems employ cameras to accurately detect and count vehicles passing through designated polygons, ensuring efficient and precise toll collection.

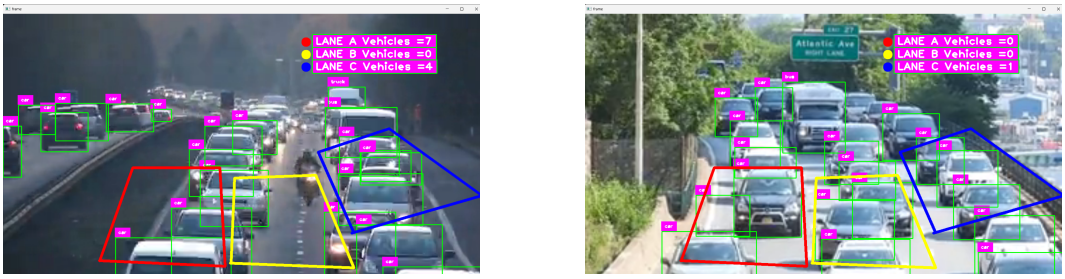


Figure 1: Vehicle Detection

5 Methodology

Various stages in the implementation process are:

Data Loading: Load toll data, including rates, zones, and geographic coordinates.Load journey data, which includes sequences of GPS coordinates.

Data Pre-processing & Cleaning: Remove invalid or missing entries.Convert coordinates and rates to appropriate numerical formats.Normalize vehicle types and other categorical data.

Feature extraction: Extract distance between GPS coordinates using the Haversine formula.Extract vehicle type, time of travel, discount type, and payment method. Determine the geographic zone based on coordinates.

Dataset Preparation: Create lists or arrays to store journey segments and their respective features. Ensure all relevant information for each segment is available for toll calculation.

Toll Calculation Algorithm : Define a class or function for toll calculation.Implement distance calculation using the Haversine formula.Apply vehicle type multipliers, time of travel multipliers, geographic zone multipliers, discounts, and additional fees.

Simulation Execution: Loop through each journey and calculate the total toll.Store and log the results for analysis.

Model Evaluation: Compare calculated tolls with expected values (if available).Evaluate accuracy, consistency, and performance under different scenarios.

Model selection and reporting Analyze the results from different vehicle types, times of travel, and geographic zones. Choose the configuration that provides the most accurate and consistent results. Report performance metrics, including total tolls, average tolls, and any deviations.

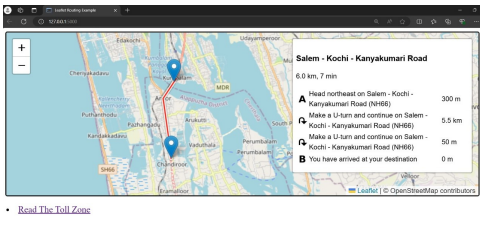
6 Implementation

To begin, you'll need to set up your Python environment with essential libraries.Once your environment is configured, the next step involves loading geographic data that defines the toll zones and possibly road networks.With the geographic data loaded, you can visualize the toll zones and road networks to ensure they are correctly defined. Visualization can be done using plotting libraries like Folium for interactive maps. Toll Zone Detection approaches we adopted are:

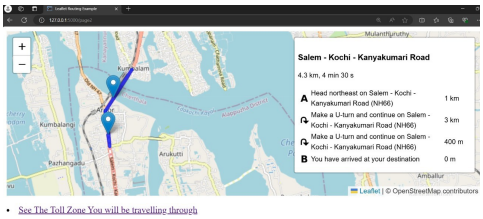
Approach 1 : We simulated vehicle movement and toll zone entry using Python libraries like geopandas. Each vehicle has a starting point, destination, and balance. We calculated distances and apply taxes based on vehicle type and distance traveled within toll zones. Flags and functions manage zone entry checks and balance deductions during simulation. The limitation of this approach in our GPS Toll system simulation project using Python is that since the vehicle moves irregularly, not following a continuous path along the desired points from the starting point to the ending point, toll zones cannot be reliably detected along the route.The irregular movement of vehicles in your GPS Toll system simulation project is due to using a mathematical formula that places the next points randomly after the starting point. This approach disrupts continuous movement, affects distance calculation, and results in inaccurate movements, ultimately leading to unreliable detection of toll zones along the route. The mathematical formula that we used here is :

```
def move_vehicle(env, vehicle, step_size=0.1):

    start = vehicle['start']
    end = vehicle['end']
    current_position = start
    speed_mps=1.0
    while current_position.distance(end) >0.00001:

        yield env.timeout(1)  # Assuming each step takes 1 time unit
        # Convert bearing to radians
        bearing_rad = math.radians(45.0)

        # Calculate movement per second
        movement_per_second_lat =  step_size* math.cos(bearing_rad) / 111300  #
                                            Approx. meters per degree
                                            latitude
        movement_per_second_lon = step_size * math.sin(bearing_rad) / (111300 *
                                            math.cos(math.radians(start.x)))
                                             # Approx. meters per degree
                                            longitude
        current_position = Point(
            current_position.x + movement_per_second_lat  ,
            current_position.y + movement_per_second_lon
        )
        vehicle['current_position'] = current_position
        #movement_log.append((env.now, vehicle['vehicle_id'], current_position))
        print(f"Time {env.now}: Vehicle {vehicle['vehicle_id']} at {
                                            current_position}")

        check_toll_zones(vehicle, toll_zones)
```

Approach 2 :Our second approach for the GPS toll system simulation using Python involves creating an interactive map with Leaflet and integrating it with Python code using Flask. We begin by passing coordinates entered by the user to the Python script and then to an HTML file, which then displays these coordinates correctly on the map. As the coordinates are processed, we calculate the toll rate based on the distance travelled and time taken, providing detailed summaries. We check if the road coordinates intersect with toll zone coordinates. If a vehicle enters a toll zone, we record the coordinates as long as they remain within the zone.The limitation in this approach is that, even if a toll zone is located between the starting and ending coordinates, the coordinates are displayed as non-intersecting lines due to the marking format in Leaflet. Leaflet represents them as two separate lines, even when the starting and ending coordinates are the same, resulting in no coinciding points. Our concept relies on identifying intersecting points, and since there are none, it impairs our ability to accurately calculate distances along the route.



(a) Starting and ending point of Toll Zone



(b) Travelling Road Network

Final Approach :Our project involves simulating a GPS toll system using Python. We began by obtaining the vehicle's start and end coordinates. Using Leaflet, we integrated these coordinates into a map to generate intermediate points along the route. These points were then compiled into a Python dictionary and processed by a Python script. By comparing these coordinates to the predefined toll zone polygons, we could accurately calculate the actual distance traveled by the vehicle.
Now, simulate vehicle movement and toll transactions. You can generate synthetic or real-world-like routes using libraries for graph-based road networks or simply by defining waypoints and paths manually. Each vehicle movement should include details such as vehicle ID, timestamp of entry and exit from toll zones, and potentially GPS coordinates (simulated or real-time).

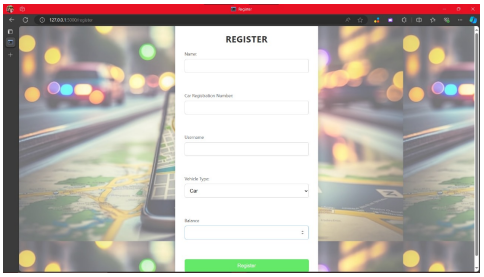
Vehicle Detection has been implemented using computer vision.Implement data structures to store this information efficiently. Use dictionaries, lists, or pandas DataFrames depending on the complexity and volume of data you're handling. For example, maintain a list of vehicles with their routes and timestamps, and another structure to track toll transactions including vehicle ID, toll zone entered, exit time, and calculated toll fee.Calculate toll fees based on predetermined rules such as vehicle type (e.g., car, truck), distance traveled through toll zones, or fixed rates per entry/exit. This calculation involves checking when a vehicle enters and exits a toll zone, computing the distance traveled if applicable, and applying the appropriate toll fee calculation logic.

Ensure the accuracy of toll collection by validating GPS coordinates against the predefined toll zone boundaries. This step ensures that toll fees are accurately applied only when a vehicle enters a designated toll zone.Finally, visualize the simulated data and toll revenues. Use plotting libraries to create graphs showing traffic patterns, toll transactions over time, and revenues generated. Alternatively, use interactive mapping libraries like Folium to create dynamic maps that illustrate vehicle movements and toll transactions spatially.By simulating the GPS toll collection system in this manner, you can thoroughly test its functionality, efficiency, and accuracy before deploying it in real-world scenarios. This simulation approach allows for iterative improvements and optimizations to ensure smooth operation and effective toll management.
Assumption : In our GPS toll system simulation using Python, we operate under the assumption that real-time GPS data provides accurate location updates, where the endpoint signifies the vehicle's stationary position upon completion of the journey. Users specify both the starting and ending locations, ensuring consistency in the vehicle's route. Our system relies on the reliability of user inputs to facilitate seamless navigation and toll processing.

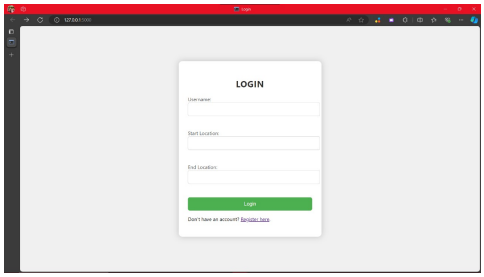
7 Results & Discussion

Our GPS Toll System Simulation App is a comprehensive solution developed to simulate toll collection using GPS technology. The application leverages the power of Python for backend processing, ensuring efficient and accurate toll calculations based on real-time GPS data.
Front End Development: The front end of our GPS Toll System Simulation App is designed

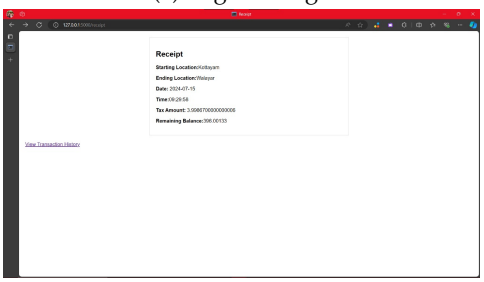
with a focus on simplicity and user experience. HTML and CSS were utilized to create a responsive and visually appealing interface.



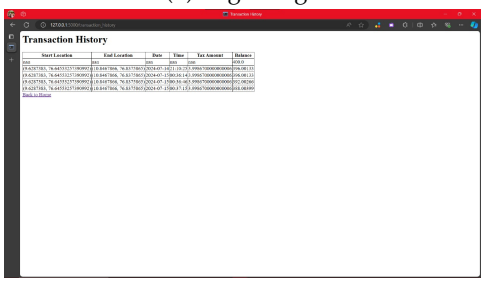
(a) Register Page



(b) Login Page



(c) Receipt



(d) Transaction History

Figure 3: App Screenshots

Back End Development: The backend is powered by Python, handling the core functionalities such as GPS data processing, toll calculations, and data management.Leaflet provides continuous coordinates and generates maps. Python robust libraries and frameworks ensure the app performs efficiently and accurately under various conditions.

8 Conclusions

The GPS Toll Collection Simulation project successfully demonstrates the integration of GPS technology with toll collection systems using Python. By simulating the movement of vehicles and their interactions with virtual toll checkpoints, we have shown how modern toll collection can be automated and streamlined.
Key Achievements:

- Effective Simulation of Vehicle Movement: The project accurately models the real-time movement of vehicles using GPS coordinates, allowing for realistic tracking and toll calculation.
- Automated Toll Collection: We implemented an efficient system for automatically calculating tolls based on distance traveled and toll points crossed, reducing the need for manual intervention.

- Scalability and Flexibility: The simulation can handle various scenarios, including different toll rates, vehicle types, and traffic patterns, demonstrating the systemâĂŹs adaptability.
- Cost Efficiency and Reduced Congestion: By leveraging GPS technology, the system can potentially reduce operational costs and traffic congestion, providing a seamless experience for drivers.

Acknowledgments

We would like to express our heartfelt gratitude and appreciation to Intel© Corporation for providing an opportunity to this project.First and foremost, we would like to extend our sincere thanks to our team mentor Er. Gayathri J L for her invaluable guidance and constant support throughout the project.We are deeply indebted to Saintgits College of Engineering and Technology for providing us with the necessary resources,and sessions on machine learning. We extend our gratitude to all the researchers, scholars, and experts in the field of machine learning and natural language processing and artificial intelligence, whose seminal work has paved the way for our project. We acknowledge the mentors, institutional heads, and industrial mentors for their invaluable guidance and support in completing this industrial training under Intel© -Unnati Programme whose expertise and encouragement have been instrumental in shaping our work. [1–8]

References

[1] AHMED, M. U., AND BEGUM, S. Convolutional neural network for driving maneuver identification based on inertial measurement unit (imu) and global positioning system (gps). *Frontiers in Sustainable Cities 2* (2020), 34.

[2] HABIBULLAH, B., TENG, R., AND SATO, K. Highway toll collection method for connected automated vehicle platooning using spatio-temporal grid reservation. *Communications and Network 14*, 4 (2022), 171–199.

[3] MANY, G. Simulating gps signals. *GPS World* (2012).

[4] NARAIN, S., RANGANATHAN, A., AND NOUBIR, G. Security of gps/ins based on-road location tracking systems. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 587–601.

[5] NETWORKS, C. N. Understanding convolutional neural networks, 2024. Accessed: 2024-07-11.

[6] TECH-WATT. Vehicle counting in lanes, 2023. Accessed: 2024-07-07.

[7] WANG, L., JIANG, P., ZHONG, J., LI, L., ZHANG, J., WEI, X., AND LI, E. Intelligent traffic guidance system based on dynamic toll collection policy. In *2014 Fourth International Conference on Communication Systems and Network Technologies* (2014), IEEE, pp. 1172–1176.

[8] WI, Y. N. Graph theory: Application of graphs in determining the most efficient highway routes.

A Main code sections for the solution

A.1 Importing Libraries

```
    #IMPORTING LIBRARIES
from email import message
from tracemalloc import start
from flask import Flask, flash, render_template, request, jsonify,redirect
from numpy import zeros
from shapely.geometry import Point,Polygon

import pandas as pd
import folium
from datetime import date, datetime,time
from geopy.geocoders import Nominatim
from time import sleep, strftime
import csv
```

A.2 Geocoding Function

```
#####################################
'''
READING COORDINATES FROM NAME
'''
# Initialize the Nominatim geocoder
geolocator = Nominatim(user_agent="my_geocoder_app") # Function to get coordinates
                                                     for a given place
def get_coordinates(place):
   location = geolocator.geocode(place)
   if location:
        return (location.latitude, location.longitude)
   else:
        raise Exception(f"Could not get coordinates for {place}")
```

A.3 Flask App initialization

```
# Places to geocode

app = Flask(_name_)
app.secret_key = 'your_secret_key'
i=0
C=[]
list_of_C=[]
map_instance = folium.Map(location=(10.800874457768161, 76.78083325901277),
                                              zoom_start=12)
```

A.4 Reading Toll Zone Coordinates

```
# Sample vehicle details
#READING TOLL ZONE**********************
# Define the corners of the toll area
```

```
toll_area_coords = [(10.800874457768161, 76.78083325901277
), (10.799205197444179, 76.78144017364988
), (10.825256523519297, 76.81391010673558
), (10.8361,76.867)]

vehicle={}
toll_polygon = Polygon(toll_area_coords)
toll_start=(10.80013677469304, 76.78150921255425)
toll_end =(10.799346364106235, 76.74865751636764)
coordinates_r =[]
```

A.5 Storing data to a CSV File

```
#***********************************
'''
STORING
DATA
TO A
CSV FILE
'''
# CSV file path
data_file = 'login_data.csv'
csv_file='vehicle_data.csv'
try:
    # Try to load existing data
    df = pd.read_csv(csv_file)
except FileNotFoundError:
    # If file does not exist, create an empty DataFrame
    df = pd.DataFrame(columns=['Starting Location', 'Ending Location', 'Time', '
                                                    Date','Tax Amount','Balance'])
def get_real_time_data(sn,e,t,b):
    data = {
        'Starting Location': sn,
        'Ending Location': e,
        'Time': datetime.now().strftime("%H:%M:%S"),
        'Date': datetime.now().strftime("%Y-%m-%d"),
        'Tax Amount':t,
        'Balance':b
    }
    return data

def collect_and_store_data(sn,e,t,b):
    try:
        # Try to load existing data
        df = pd.read_csv(csv_file)
    except FileNotFoundError:
        # If file does not exist, create an empty DataFrame
        df = pd.DataFrame(columns=['Starting Location', 'Ending Location', 'Time',
                                                    'Date','Tax Amount','Balance'])

    # Get real-time data
    data_to_append = get_real_time_data(sn,e,t,b)
    # Convert the dictionary to a DataFrame
    new_data = pd.DataFrame([data_to_append])
    # Append the new data to the existing DataFrame
    df = pd.concat([df, new_data], ignore_index=True)
    # Save the updated DataFrame to the CSV file
    df.to_csv(csv_file, index=False)
```

A.6 Determine Peak time Function

```
#*************************************
flag=0
# map_instance = folium.Map(location=road_start, zoom_start=12)
'''
FUNCTION TO DETERMINE PEAK TIME
'''
def simulate(coordinates_road):
    global coordinates_r
    coordinates_r=coordinates_road#value globally assigned
def get_time_type(current_time):
    # Define the time ranges and their labels
    time_ranges = [
        ((7, 0), (9, 0), "Peak Time"),
        ((16, 0), (18, 0), "Peak Time"),
        ((9, 0), (16, 0), "Non-Peak Time"),
        ((18, 0), (23, 59), "Non-Peak Time"),
        ((0, 0), (7, 0), "Off-Peak Time"),
    ]

    # Convert current_time to time object for comparison
    current_time_obj = current_time.time()

    # Check current_time against defined time ranges
    for start, end, label in time_ranges:
        start_time = time(*start)
        end_time = time(*end)

        if start_time <= current_time_obj <= end_time:
            return label

    return "Unknown Time"
```

A.7 Fetching Vehicle Data

```
def fetch_vehicle_data(username):
         try:
            df=pd.read_csv(csv_file)
            user_data=df[df['Username'] == username].iloc[0]
            vehicle_type=user_data['Vehicle Type']
            balance=float(user_data['Balance'])
            return vehicle_type,balance
         except Exception as e:
            print(f"Error Fetching Vehicle Data: {e}")
            return None,None
```

A.8 Flask Routes and views

```

###################################
@app.route('/')
def index1():
    return render_template('login.html')

@app.route('/register')
```

```
def register():
    return render_template('register.html')

@app.route('/register',methods=['POST'])
def register_submit():
    # Fetch form data
    global Balance
    name = request.form['name']
    reg_number = request.form['reg_number']
    Username = request.form['Username']
    vehicle_type = request.form['vehicle_type']
    Balance = request.form['Balance']

    # Prepare data to append to CSV
    data = {
        'Name': name,
        'Registration Number': reg_number,
        'Username': Username,
        'Vehicle Type': vehicle_type,
        'Balance': Balance
    }

    # Append data to CSV
    with open(data_file, 'a', newline='') as file:
        writer = csv.DictWriter(file, fieldnames=data.keys())
        if file.tell() == 0:
            writer.writeheader()  # Write header only if file is empty
        writer.writerow(data)

    # Redirect back to login page after successful registration
    return render_template('login.html')
```

A.9 Login Route

```
# Function to check if username exists in CSV file
def check_username(username):
    try:
        with open(data_file, 'r') as file:
            reader = csv.DictReader(file)
            for row in reader:
                if row['Username'] == username:
                    return True
        return False
    except FileNotFoundError:
        return False

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        global start_location,end_location
        username = request.form['username']
        start_location = request.form['start_location']
        end_location = request.form['end_location']
        global z,y,vehicle
        z = get_coordinates(start_location)
        print(f"Coordinates of 0{'start_point'}: {z}")
        sleep(1)  # Sleep for 1 second to avoid hitting rate limits
        y = get_coordinates(end_location)
```

```
        print(f"Coordinates of {'end_point'}: {y}")
        # Get coordinates with a delay to avoid hitting usage limits
        vehicle_type,balance=fetch_vehicle_data(username)
        if vehicle_type is not None and balance is not None:
            vehicle ={
                'type':vehicle_type,
                'balance':balance
            }

    if check_username(username):
            return render_template('index.html',map=map_instance.repr_html(),
                                                    start_point=z,end_point=y)
    else:
            # Handle invalid username (e.g., show error message)
            flash('Your Username is Invalid..Try Again')
            return render_template('register.html', message="Error")
```

A.10 Index and Page Routes

```
# @app.route('/process', methods=['POST'])
# def process():
#     global start_location,end_location
#     start_location = request.form['start_location']
#     end_location = request.form['end_location']
#     balance = request.form['balance']
#     vehicle_type = request.form['vehicle-select']
#     recharge_amount = request.form.get('recharge', 0)  # Default to 0 if not
                                        provided

#     # Process the data
#     processed_data = process_data( balance, vehicle_type, recharge_amount)
#     print(processed_data)

    # Render the result page and pass processed data
    # return render_template('result.html', data=processed_data)
@app.route('/index')
def index():
    # Render the index.html template with the map instance
    return render_template('index.html', map=map_instance.repr_html(),start_point=
                                            (9.6287383, 76.64553257390992),
                                            end_point=(-1.87534,87.876545))

@app.route("/page2")
def page2():

    # Render another HTML file with the same map instance
    return render_template('page2.html',map=map_instance.repr_html(),start_point=
                                            toll_start,end_point=toll_end)
```

A.11 Saving coordinate Routes

```
@app.route('/save_coordinates_route1', methods=['POST'])
def save_coordinates_route1():

    data_road = request.json
    coordinates_road = data_road.get('coordinates')
```

```
    simulate(coordinates_road)
    # Process the received coordinates for Route 1
    #print('Received coordinates for ROAD:', coordinates_road)
    # Example response

    return jsonify(status='success', message='Received coordinates for Route 1')


@app.route('/save_coordinates_route2', methods=['POST'])
def save_coordinates_route2():

    data_toll = request.json
    #print(type(data_toll)) dict
    return jsonify(status='success', message='Received coordinates for Route 2')
```

A.12 Processing Route Coordinates

```
@app.route("/page3")
def page3():
    #define the list of tuples here !!!!!!!!!
    #coordinates_r=simulate()#check for toll zones and convert list of dict into
                                                list of tuples and pass to next html
                                                file
    global flag

    #print("CCCCC=",coordinates_r)
    vehicle_df = pd.DataFrame(coordinates_r)
    vehicle_geometry = [Point(xy) for xy in zip(vehicle_df['lat'], vehicle_df['lng
                                                '])]

    inside_toll_zone=[]
    for i in vehicle_geometry:
        if toll_polygon.contains(i):
            inside_toll_zone.append(i)
            flag=1
            # Optionally, print or log a message
            # print(f"The point {i} is inside the toll zone")
        else:
            # Optionally, print or log a message
            # print(f"The point {i} is outside the toll zone.")
            pass
    coordinates_tuples = [(point.x, point.y) for point in inside_toll_zone]
# Convert to list of tuples

    # Render another HTML file with the same map instance
    return render_template('page3.html',map=map_instance.repr_html(),start_point=
                                                coordinates_tuples[0],end_point=
                                                coordinates_tuples[-1])

@app.route('/save_coordinates_route3', methods=['POST'])
def save_coordinates_route3():
    global tax,B
    tax=0
    B=vehicle['balance']
    data_travelled = request.json
    # print(type(data_toll)) dict
    summary=data_travelled.get('summary')
    print(summary)
```

```
    if flag==1:
        # Get the current time
        current_time =datetime.now()
        global time_type
# Determine the type of time
        time_type = get_time_type(current_time)
        #Determining base rate on the basis of vehicle
        T='Car'
        D=summary['totalDistance']/1000
        print("Distance in kms=",D)
        base_rate = {
         'Car': 0.65,
         'Jeep': 0.65,
         'Van': 0.65,
         'Light motor vehicle': 0.65,
         'Light commercial vehicle': 1.05,
         'Light goods vehicle': 1.05,
         'Mini bus': 1.05,
         'Bus': 2.2,
         'Truck': 2.2,
         'Others': 3.39
         }

        #tax calculation

        B=vehicle['balance']

        if T in base_rate:
           tax=base_rate[T] * D
           if time_type=="Peak Time":
              B=B-5
        if tax<B:
            B=B-tax
            print("Total amount=",tax)
            print("Balance=",B)
            vehicle['balance']=B
        else:
            flash("Not Enough Balance")
            print(tax-B,"more needed")

    collect_and_store_data(z,y,tax,B)

    # Example response
    return jsonify(status='success', message='Received coordinates for Route 2')
```

A.13 Receipt route

```
    @app.route('/receipt')
def receipt():

    try:
        df=pd.read_csv(csv_file)
        if df.empty:
            print("No transactions found in the DataFrame.")
    # Process the latest transaction
        else:
          latest_transaction = df.tail(1).iloc[0]
```

```
        return render_template('receipt.html',start_location=start_location,
                                              end_location=end_location,date=
                                              datetime.now().strftime("%Y-%m-
                                              %d"),time=datetime.now().
                                              strftime("%H:%M:%S"),t=tax,
                                              balance=B)

    except FileNotFoundError:
        return render_template(receipt.html)
```

A.14 Transaction History Route

```
# Route to display transaction history
@app.route('/transaction_history')
def transaction_history():
    try:
        df = pd.read_csv(csv_file)
        transactions = df.to_dict('records')  # Convert DataFrame to list of
                                                dictionaries
        return render_template('transaction_history.html', transactions=
                                              transactions)
    except FileNotFoundError:
        flash("No transactions found.")
        return render_template('transaction_history.html')
```

A.15 Run the App

```
if _name_ == '_main_':
    app.run(debug=True)
```