INTEL-UNNATI AI&ML JOURNAL
Number 1 (2024), pp. 1–19

Code:IUCP-2024

MACHINE LEARNING

GPS Toll Based System Simulation using Python

Mathews Reji, Muhammed Anees, Navya Prasad, Neeraja S, and Shalin Ann Thomas

Saintgits Group of Institutions, Kottayam, Kerala

15th July 2024

Abstract: This project presents the development of a GPS-based toll collection system simulation using `Python` aimed at modernizing and streamlining toll fee processes. The proposed system leverages GPS technology to automatically detect vehicle movements through virtual toll zones, calculating toll fees based on the distance traveled or specific routes taken. The simulation encompasses key components such as GPS-enabled vehicles, dynamic toll calculation algorithms, and efficient data management. `Python`'s extensive libraries are utilized to handle various aspects of the simulation: `geopy` and `shapely` for geospatial analysis, `pandas` for data manipulation, and `folium` for interactive map visualization. Additionally, visualization tools such as `matplotlib` and `plotly` are employed to present real-time data and system performance metrics. The project aims to demonstrate the feasibility and advantages of a GPS-based toll collection system, highlighting its potential to enhance operational efficiency. The simulation serves as a proof of concept, showcasing the integration of GPS technology into intelligent transportation systems and providing a foundation for further research and development in this field.

Keywords: GPS (Global Positioning System) technology, Toll collection, Automated tolling, Python simulation, Virtual toll zones, Real-time tracking, Distance-based toll calculation, Route-based toll charges, Vehicle movement simulation, Data management, User-friendly interface, Intelligent transportation systems, QGIS, Dynamic toll algorithms, Real-time toll visualization, Python libraries, Seamless tolling, Toll system optimization, Location-based services, Automated fee collection, Simulation modeling, Toll transaction processing, Python programming, Vehicle tracking GIS (Geographic Information System), Smart transportation, ETC (Electronic Toll Collection), Map visualization, Route planning, LightGBM

Introduction

The introduction of GPS technology has greatly boosted operational efficiency and transformed multiple sectors, particularly transportation. GPS tracks vehicle movements by producing sequences of GPS coordinates. These coordinates represent the vehicle's path, often with simulated noise to mimic real-world variability. One notable application is in toll collection systems, where GPS technology enables seamless toll management. GPS, a satellite-based navigation system, provides real-time location data, enabling the automation of toll collection processes. Traditional toll booths, which require vehicles to stop and pay, often cause traffic congestion and delays, but a GPS toll-based system can automatically detect when a vehicle enters and exits predefined toll zones, calculate the corresponding toll fees based on the distance traveled or specific routes taken, and charge the user without requiring them to stop or slow down. This project aims to develop a `Python`-based simulation of a GPS toll-based system. The simulation will encompass the creation of virtual toll zones, GPS-enabled vehicles, and algorithms for dynamic toll calculation. By leveraging `Python`'s extensive libraries for geospatial analysis and data management, we will design a user-friendly interface to visualize vehicle movements and real-time toll charges. The simulation will serve as a proof of concept, demonstrating the system's efficiency and potential benefits over traditional methods. In conclusion, the GPS toll-based system simulation in `Python` provides a versatile platform for studying, developing, and refining toll collection mechanisms using GPS technology. It serves as a crucial tool for transportation planners, policymakers, and technologists seeking to enhance efficiency, fairness, and sustainability in toll management systems.

Python Libraries Used

The project utilizes the following packages for various tasks.

```
    numpy
    pandas
    matplotlib
    geopy
    simpy
    geopandas
    shapely
    datetime
    random
    os
    pickle
    pygame
    math
    networkx
    folium
    joblib
    sklearn
    lightgbm
    flask
    git
    secrets
    request
    json
    CORS
```

Methodology

To simulate a GPS toll-based system using `Python`, we followed a structured methodology as shown below:

Dataset Creation: Creating a dataset using the data obtained from GPS coordinates and measuring the distance between them.

Simulation: Develop a simulation based on the dataset of the GPS toll-based system, including entities, events, and their interactions.

Data Handling: Use libraries like `simpy`, `shapely` for managing events, processes, and resource allocation. Also using `pandas` for analyzing data.

Simulation Refinement: Based on initial simulation results and feedback, refine the model to improve accuracy and realism.

Classification: Implementing various Machine Learning Classification models on the datasets to check which version of the GPS Toll Based System Simulation provided the most efficiency and accuracy.

Model Training: Choose a classification algorithm, such as LightGBM, Linear Regression, Random Forest Regressor, Gradient Boosting Regressor, Support Vector Regressor, K-nearest Neighbors Regressor, XGBoost Regressor, Ridge Regression, Lasso Regression and train the model using the training set & the extracted features.

Model Evaluation: Test the trained model on the testing set to evaluate its performance. Use the metrics such as training time, prediction, Mean Squared Error and R∧2 Score to assess the model's effectiveness.

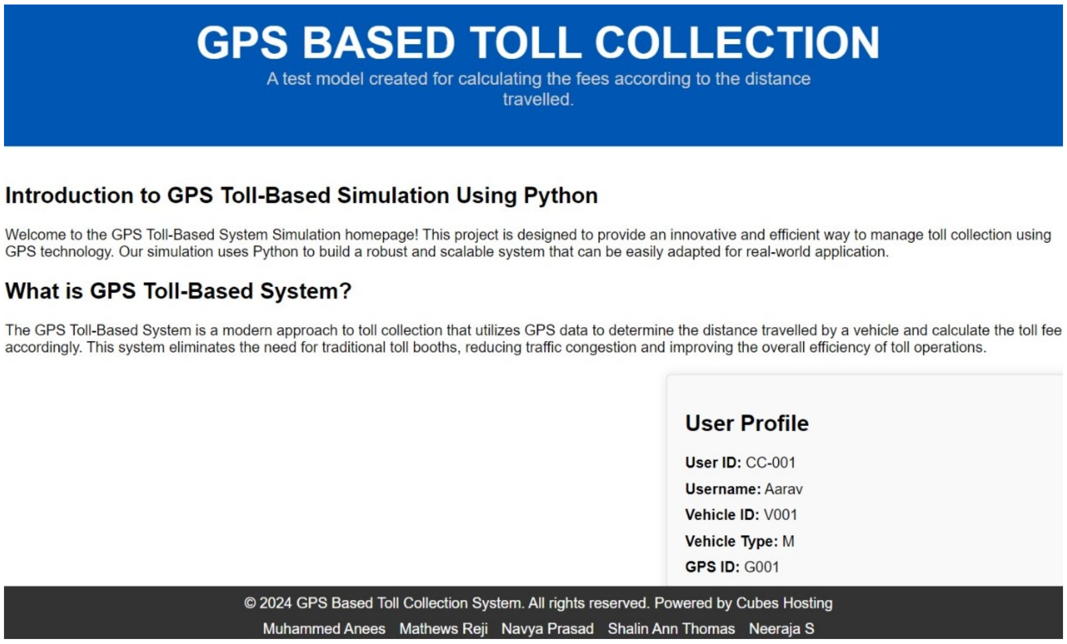
Model selection and reporting: In this stage, based on various performance criteria, the better model of GPS Toll Based System is selected.

Implementation

As the first step in the task to implement a GPS toll based system simulation using `Python`, we first needed to create a dataset containing roads of an area. This was achieved by using a software geofabrik. The dataset thus created was of the type shapefile. Now, in-order to separate the roads and highways specifically, we use a software QGIS. We procured a 36 square kilometer area dedicated to vehicle simulation using data-driven methods. The dataset for Version-1 is `01-06-simorg-combined.csv`. In the initial version of our simulation (Version-1), we focused on 30 vehicles of the same type, without factoring in their speed. Our primary objective was to accurately predict fees for various travel distances. To achieve this, we employed a rigorous approach to train machine learning models, experimenting with different methodologies. After thorough evaluation, we determined that LightGBM delivered the most effective results. LightGBM produced the best R∧2 score with very low prediction time. Subsequently, we proceeded to deploy this optimized model for practical application. Version-1 of GPS Toll Based system worked perfectly, but it was extremely basic and did not meet all of the required criteria. We needed to include variety of constraints in the toll calculation such as vehicle type, rush hour toll, etc.

In Version-2 of the GPS toll-based collection system, several enhancements and new features are introduced to make the system more efficient and comprehensive compared to Version-1. A new dataset `simulated-output.csv` is created. The system now supports multiple types of vehicles including heavy vehicles, medium vehicles, two-wheeled vehicles, and special vehicles. Specific categories like police cars, ambulances, and vehicles owned by people living on the highway are classified as special vehicles and are exempted from toll charges. The system only calculates toll charges if the vehicle moves a minimum of 30 meters and more. This ensures that accidental entries into the toll zones do not incur charges. Toll is only calculated when a vehicle enters predefined toll zones. The system calculates the average speed of each vehicle over its journey. This information can be used to analyze traffic patterns and enforce speed regulations. To manage traffic congestion, the system implements a rush hour toll. This means higher toll charges are applied during peak traffic hours to encourage drivers to travel during off-peak hours, thereby distributing traffic more evenly throughout the day. Each vehicle is assigned a unique Vehicle ID. It ensures that each vehicle's toll is calculated based on its specific journey and type.

The front end of the website was developed using `HTML` (HyperText Markup Language), `CSS` (Cascading Style Sheets) and `JavaScript` while the back end was mainly developed using `Python`'s flask app.The interaction of the website was made possible using `requests` library. The website was hosted by Cubes Hosting https://cubes.host/. The website displays a visual plot of the user's most recent journey on a map. To login in to the website, the user has an ID with a secure password which must contain uppercase letter, lowercase letter, numbers and symbols. The complex password prevents the leakage of user data. The website contains a user profile; which includes the User ID, User Name, Vehicle ID, Vehicle Type and GPS ID, it also displays the toll obtained by the user, displays the last 5 journeys as well as plots the map of the latest journey. The website also has a payment gateway that allows the user to pay the tolls directly. To manage this website, an administrator web-page was developed which allowed the administrators to check the vehicle data as well as user data.



GPS BASED TOLL COLLECTION

A test model created for calculating the fees according to the distance travelled.

Introduction to GPS Toll-Based Simulation Using Python

Welcome to the GPS Toll-Based System Simulation homepage! This project is designed to provide an innovative and efficient way to manage toll collection using GPS technology. Our simulation uses Python to build a robust and scalable system that can be easily adapted for real-world application.

What is GPS Toll-Based System?

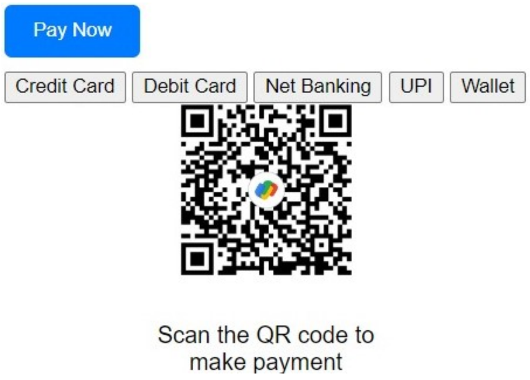
The GPS Toll-Based System is a modern approach to toll collection that utilizes GPS data to determine the distance travelled by a vehicle and calculate the toll fee accordingly. This system eliminates the need for traditional toll booths, reducing traffic congestion and improving the overall efficiency of toll operations.

User Profile

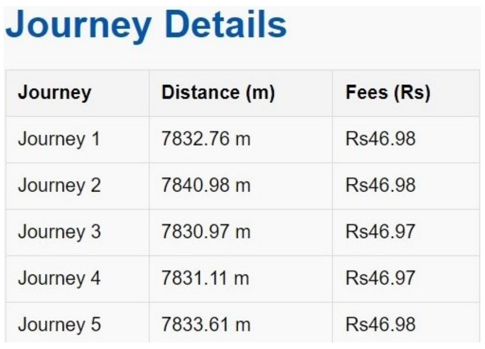
User ID: CC-001
Username: Aarav
Vehicle ID: V001
Vehicle Type: M
GPS ID: G001

© 2024 GPS Based Toll Collection System. All rights reserved. Powered by Cubes Hosting
Muhammed Anees Mathews Reji Navya Prasad Shalin Ann Thomas Neeraja S

Figure 1: User Home Page of GPS Toll-Based System Simulation using Python



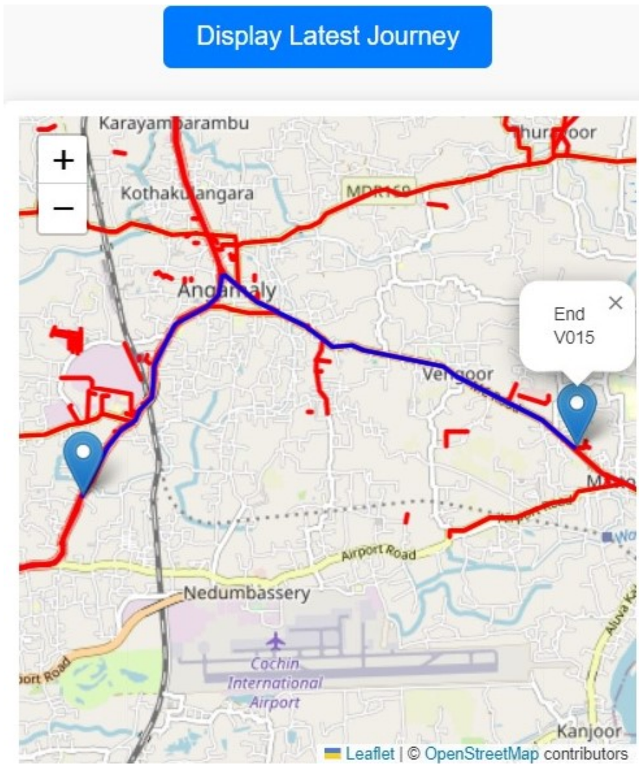
(a) Payment using Google Pay



Journey Details

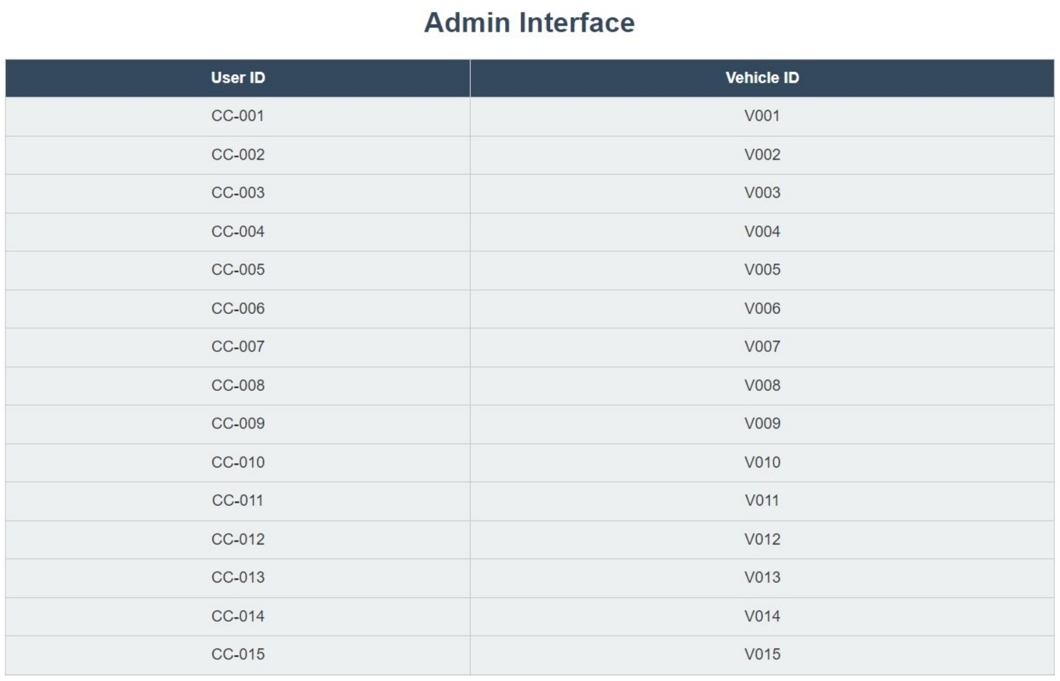
Journey	Distance (m)	Fees (Rs)
Journey 1	7832.76 m	Rs46.98
Journey 2	7840.98 m	Rs46.98
Journey 3	7830.97 m	Rs46.97
Journey 4	7831.11 m	Rs46.97
Journey 5	7833.61 m	Rs46.98

(b) Displaying the last 5 journeys of the User



(c) Plotting the User's latest journey

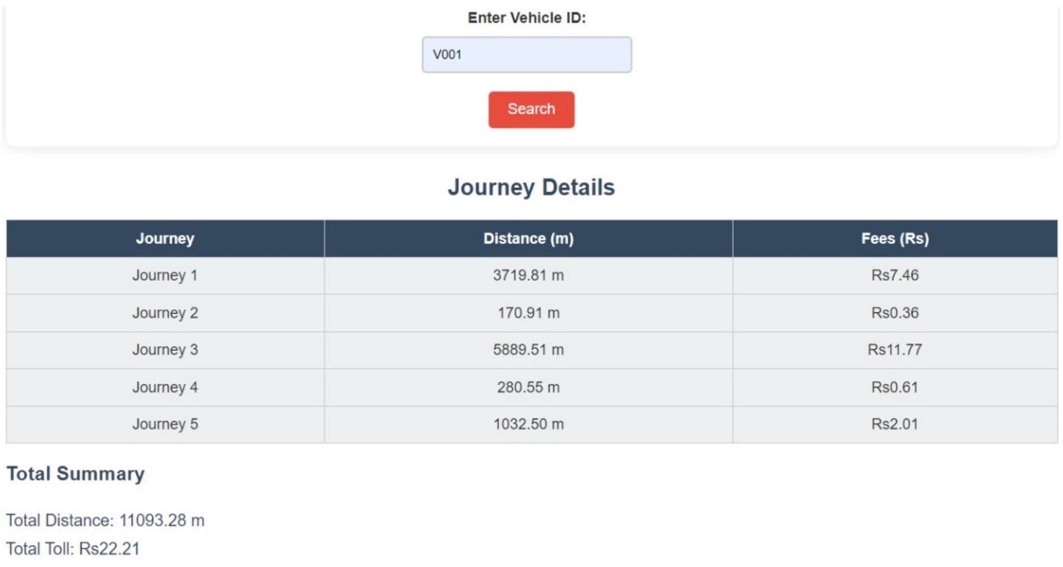
Figure 2: User Interface Features



Admin Interface

User ID	Vehicle ID
CC-001	V001
CC-002	V002
CC-003	V003
CC-004	V004
CC-005	V005
CC-006	V006
CC-007	V007
CC-008	V008
CC-009	V009
CC-010	V010
CC-011	V011
CC-012	V012
CC-013	V013
CC-014	V014
CC-015	V015

Figure 3: Admins handling the details of the Users



Journey Details

Journey	Distance (m)	Fees (Rs)
Journey 1	3719.81 m	Rs7.46
Journey 2	170.91 m	Rs0.36
Journey 3	5889.51 m	Rs11.77
Journey 4	280.55 m	Rs0.61
Journey 5	1032.50 m	Rs2.01

Total Summary

Total Distance: 11093.28 m
Total Toll: Rs22.21

Figure 4: Admins can obtain the journeys travelled by the vehicle using its Vehicle-ID

Features Offered

The following features were implemented in this project.

Scalability: This GPS toll-based system simulation can easily adapt to any road data, providing versatile applications across various road networks. This system can be implemented globally if the dataset and the system requirements are met.

Efficiency: With the new dataset and conditional toll calculations, the system is more efficient and accurate.

Uniqueness: Unique features such as unique toll calculation for each vehicle using vehicle ID, rush hour toll, time slot, etc. An administrator website is created to maintain and supervise the system.

Modularity: The code is structured into separate modules, each serving specific purposes to facilitate easy access and debugging. New modules can be created and integrated into the existing system without the need of the entire code.

Traffic Management: Rush hour tolls help in managing traffic flow and reducing congestion.

Security Features: Each user has a unique ID and password. The password is complex and should contain a number, uppercase character, lowercase character and a special symbol.

Results & Discussion

Popular classical Machine Learning algorithms from the `Python` library `sklearn` is used for model training and testing. Here our test dataset is the dataset containing the paths of the simulated vehicles. We use this dataset to train the model to calculate the toll.

Table 1: Testing GPS based toll system collection dataset

Model No.	Model Name	Training Time	Prediction Time	Mean Squared Error	R∧2 Score
1.	LightGBM	0.18 sec	0.01 sec	0.008758327	0.999920649
2.	Linear regression	0.03 sec	0.00 sec	39.97205449	0.637849551
3.	Random Forest Regresor	21.99 sec	0.42 sec	0.00500097	0.999954691
4.	Gradient Boosting Regressor	7.79 sec	0.02 sec	0.085283954	0.99922732
5.	Support Vector Regressor	194.50 sec	43.80 sec	101.8897264	0.076869813
6.	K-nearest Neighbors Regressor	0.08 sec	0.11 sec	47.66339141	0.568165339
7.	XGBoost Regressor	0.32 sec	0.01 sec	0.002159548	0.999980434
8.	Ridge Regression	0.01 sec	0.00 sec	39.98024338	0.637775359
9.	Lasso Regression	0.48 sec	0.00 sec	55.18411613	0.50002689

The Table 1, shows the summary of the training time, prediction time, mean square error and R2 score of the models tested on predicting the GPS dataset. From the Table 1, we can identify that the model LightGBM stands out as the optimal choice for toll calculation, achieving a high R∧2 score alongside minimal mean squared error, training time, and prediction time. It was used in calculating the toll. Therefore, LightGBM was utilized in calculating the toll.

We launched a website https://gps-toll.cubeshosting.com/#, which accurately calculates the toll and plots the latest journey travelled by the user. Each user has a unique login and password that grants them access to the website. The website enables users to view their last five journeys and includes a payment gateway feature. The website is user-friendly and secure which offers flawless toll prediction and payments.

An administrator website https://gps-toll.cubeshosting.com/admin/# was created to provide easy access and management of the Toll website for admins. The administrators can check and manage user profiles, view the vehicle journeys using vehicle ID and manage website traffic.

Conclusion

The development and simulation of a GPS toll-based system using `Python` to demonstrate the feasibility and effectiveness of leveraging modern geospatial technologies for automated toll collection is successfully implemented and executed. The project culminated in the successful development of a user-friendly website. This platform empowers users to effortlessly plan their journeys, access toll information, and complete payments seamlessly. This project has successfully demonstrated the potential of GPS-based toll systems, providing a robust framework for future enhancements and real-world applications. The system's performance was evaluated through unit and integration testing, demonstrating its reliability and accuracy. The analysis of different variables, such as vehicle speed and route changes, provides insights into their effects on toll collection, enabling further optimization of the system. The project effectively integrates GPS data with geofencing techniques to simulate vehicle movements and detect toll zone entries and exits. The simulation of vehicle routes using `Python` allows for the generation of realistic GPS data, which can be used to test and validate the toll system. The project successfully defines and detects toll zones using geofencing techniques.

Acknowledgments

We would like to express our heartfelt gratitude and appreciation to Intel© Corporation for providing an opportunity to do this project on GPS Toll-Based System Simulation using `Python`. First and foremost, we would like to extend our sincere thanks to our team mentor Er.Prince Abraham for his invaluable guidance and constant support throughout the project. We are deeply indebted to our college Saintgits College of Engineering and Technology for providing us with the necessary resources and sessions on the basics to help us in doing the project. We extend our gratitude to all the researchers, scholars, and experts in the field of machine learning, whose seminal work has paved the way for our project. We also acknowledge the contributions of the open-source community, whose development of libraries such as `geopy`, `shapely`, `folium`, `pandas`, `simpy`, `matplotlib`, and `plotly` have been essential for this project. We extend our sincere appreciation to Cubes Hosting:https://cubes.host/ for their hosting services to host our website. We acknowledge the mentors, institutional heads, and industrial mentors for their invaluable guidance and support in completing this foundation course in machine learning under Intel© - Unnati Programme whose expertise and support have played a crucial role in shaping our endeavors. []

References

[1] DALUMPINES, R., AND SCOTT, D. M. Making mode detection transferable: extracting activity and travel episodes from gps data using the multinomial logit model and python. *Transportation planning and technology 40*, 5 (2017), 523–539.

[2] FONSECA-CAMPOS, J., FONSECA-RUIZ, L., AND CORTEZ-HERRERA, P. N. Portable system for the calculation of the sun position based on a laptop, a gps and python. In *2016 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)* (2016), IEEE, pp. 1–5.

[3] 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.

[4] LAI, S., XU, H., LUO, Y., ZOU, F., HU, Z., AND ZHONG, H. Expressway vehicle arrival time estimation algorithm based on electronic toll collection data. *Sustainability 16*, 13 (2024), 5581.

[5] MELNIKOV, V., KRZHIZHANOVSKAYA, V. V., LEES, M. H., AND BOUKHANOVSKY, A. V. Data-driven travel demand modelling and agent-based traffic simulation in amsterdam urban area. *Procedia Computer Science 80* (2016), 2030–2041.

[6] MOONEY, P., AND CORCORAN, P. Accessing the history of objects in openstreetmap. In *Proceedings AGILE* (2011), vol. 353, pp. 1–3.

[7] MOYROUD, N., AND PORTET, F. Introduction to qgis. *QGIS and generic tools 1* (2018), 1–17.

[8] ÖSTERMAN, A. Map visualization in arcgis, qgis and mapinfo, 2014.

[9] TAN, J. Y., KER, P. J., MANI, D., AND ARUMUGAM, P. Development of a gps-based highway toll collection system. In *2016 6th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)* (2016), IEEE, pp. 125–128.

[10] TAN, J. Y., KER, P. J., MANI, D., AND ARUMUGAM, P. Gps-based highway toll collection system: Novel design and operation. *Cogent Engineering 4*, 1 (2017), 1326199.

[11] 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.

[12] WYCOFF, E., AND GAO, G. X. A python software platform for cooperatively tracking multiple gps receivers. In *Proceedings of the 27th International Technical Meeting of The Satellite Division of The Institute of Navigation (ION GNSS+ 2014)* (2014), pp. 1417–1425.

A Main Code Sections for the solution

A.1 Importing the necessary libraries for Version-1

Importing the necessary packages in Python needed for implementing the Version-1 of GPS toll based system.

```
# Import necessary modules
import geopandas as gpd
from shapely.geometry import LineString, Point
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime, timedelta
import math
```

A.2 Loading the shapefile

Data for this project is stored as a shapefile containing the coordinates of the roads. This code is used in both Version-1 and Version-2.

```
# Load shapefile
shapefile_path = 'D:\intel\mergedroad_part\mergedroad.shx'
gdf = gpd.read_file(shapefile_path)

# Ensure it's loaded correctly
print(gdf.head())
```

A.3 Extracting coordinates from the dataset

The coordinates required for calculating distance and toll are extracted from the dataset. This code is used in both Version-1 and Version-2.

```
# Extract all LineString geometries
lines = [geom for geom in gdf.geometry if isinstance(geom, LineString)]

# Extract coordinates from all LineString geometries
all_coordinates_utm = [list(line.coords) for line in lines]

def calculate_total_distance(coords_list):
    return sum
        (
        np.linalg.norm(np.array(coords[i]) - np.array(coords[i-1]))
        for coords in coords_list
        for i in range(1, len(coords))
        )

def calculate_bearing(start, end):
    start_lat, start_lon = start
    end_lat, end_lon = end
    delta_lon = end_lon - start_lon
    x = math.sin(delta_lon) * math.cos(end_lat)
    y = math.cos(start_lat) * math.sin(end_lat) - math.sin(start_lat) * math.cos(
                                          end_lat) * math.cos(delta_lon)
    initial_bearing = math.atan2(x, y)
    initial_bearing = math.degrees(initial_bearing)
    compass_bearing = (initial_bearing + 360) % 360
    return compass_bearing
```

A.4 Calculating time increment

To calculate the time by dividing the distance (in meters) and speed (in meter per seconds). This code is used in both Version-1 and Version-2.

```
def calculate_time_increment(distance, speed_mps):
    # Calculate time in seconds based on distance (in meters) and speed (in meters
                                          per second)
    return distance / speed_mps
```

A.5 Simulation of the vehicle using the coordinates in Version-1

Using the coordinates extracted, we simulate the vehicle path using the python in Version-1. The code is given below:

```
# Defining a function to simulate the vehicle path
def simulate_vehicle_movement(gdf, all_coordinates_utm, num_vehicles=100,
                                          num_steps=1000, max_speed_kmph=60,
                                          start_time=None):
    if start_time is None:
      start_time = datetime.now()
    all_vehicle_data = []
    total_distance = calculate_total_distance(all_coordinates_utm)
    max_speed_mps = max_speed_kmph / 3.6  # Convert max speed to meters per second

    for vehicle_id in range(num_vehicles):
      vehicle_data = []
      current_distance = np.random.uniform(0, total_distance)
      speed_mps = np.random.uniform(0.5, max_speed_mps)  # Speed in meters per
                                                      second
      timestamp = start_time

      for step in range(num_steps):
         accumulated_distance = 0.0

         for coords in all_coordinates_utm:
            for i in range(1, len(coords)):
               start = np.array(coords[i-1])
               end = np.array(coords[i])
               segment_distance = np.linalg.norm(end - start)

               if accumulated_distance + segment_distance >= current_distance:
                   ratio = (current_distance - accumulated_distance) /
                                                            segment_distance
                   point_utm = start + ratio * (end - start)

                   # Transform point back to the original CRS
                   point_utm_geometry = gpd.GeoSeries([Point(point_utm)], crs=gdf.
                                                            crs)
                   point = point_utm_geometry.to_crs(original_crs).iloc[0].coords[
                                                            0]
```

```
                    speed_kmph = speed_mps * 3.6  # Convert speed back to km/h for
                                                           recording
                    direction = calculate_bearing(start, point)
                    time_increment = calculate_time_increment(segment_distance,
                                                           speed_mps)
                         timestamp += timedelta(seconds=time_increment)

                    # Append data with formatted timestamp (hour and minute only)
                    formatted_timestamp = timestamp.strftime('%H:%M')
                    vehicle_data.append((vehicle_id, step, point[0], point[1],
                                                           speed_kmph,
                                                           formatted_timestamp,
                                                           direction))

                    current_distance += speed_mps / num_steps
                    break
                accumulated_distance += segment_distance

             if accumulated_distance >= current_distance:
                break

     all_vehicle_data.extend(vehicle_data)

    df = pd.DataFrame(all_vehicle_data, columns=['vehicle_id', 'step', 'x', 'y', '
                                        speed_kmph', 'timestamp', 'direction'
                                        ])
    return df
```

A.6 Defining the vehicles in Version-1

In Version-1 of GPS toll based system, only a single common type of vehicle is considered as given below.

```
num_vehicles = 30  # Increased number of vehicles
num_steps = 80  # Number of steps in the simulation
max_speed_kmph = 60  # Maximum speed of vehicles in km/h
start_time = datetime(2024, 1, 1, 0, 0, 0)  # Define a specific start time if
                                          needed

vehicle_data = simulate_vehicle_movement(gdf, all_coordinates_utm, num_vehicles,
                                          num_steps, max_speed_kmph, start_time)
print(vehicle_data.head())
```

A.7 Displaying the simulated vehicle path in Version-1

The simulated vehicle path is then stored in an output csv file using `matplotlib.pyplot` for Version-1 of GPS toll based system. The code is shown below:

```
plt.figure(figsize=(12, 8))
for vehicle_id in range(num_vehicles):
    vehicle_path = vehicle_data[vehicle_data['vehicle_id'] == vehicle_id]
    plt.plot(vehicle_path['x'], vehicle_path['y'], linestyle='-', marker='', label
                                        =f'Vehicle {vehicle_id}')
plt.title('Simulated Vehicle Paths')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.grid(True)
plt.show()

output_path = 'simulated_vehicle_paths_org.csv'
vehicle_data.to_csv(output_path, index=False)
print(f"Simulated vehicle paths saved to {output_path}")
```

A.8 Generating vehicle ID for different vehicles in Version-2

Since more than one type of vehicle is considered in Version-2 of GPS toll based system, a unique ID is generated for each vehicle.

```
def generate_vehicle_id(vehicle_type, vehicle_count):
    return f'{vehicle_type}-{vehicle_count}'
```

A.9 Simulation of the vehicle using the coordinates in Version-2

Using the coordinates extracted, we simulate the vehicle path using the python in Version-2. The code is given below:

```
# Defining a function to simulate the vehicle path
def simulate_vehicle_movement(gdf, all_coordinates_utm, num_heavy=10,
                                          num_two_wheeler=10, num_medium=10,
                                          num_special=5, num_steps=1000,
                                          max_speed_kmph=60, start_time=None):
   if start_time is None:
      start_time = datetime.now()
   all_vehicle_data = []
   total_distance = calculate_total_distance(all_coordinates_utm)
   max_speed_mps = max_speed_kmph / 3.6  # Convert max speed to meters per second

   vehicle_counts = {'H': num_heavy, 'T': num_two_wheeler, 'M': num_medium, 'S':
                                             num_special}

   for vehicle_type, num_vehicles in vehicle_counts.items():
      for vehicle_count in range(1, num_vehicles + 1):
         vehicle_id = generate_vehicle_id(vehicle_type, vehicle_count)
         vehicle_data = []
         current_distance = np.random.uniform(0, total_distance)
         speed_mps = np.random.uniform(0.5, max_speed_mps)  # Speed in meters per
                                                  second
         timestamp = start_time

         for step in range(num_steps):
            accumulated_distance = 0.0
            speed_multiplier = 1.0  # Multiplier for speed adjustments

            for coords in all_coordinates_utm:
                for i in range(1, len(coords)):
                   start = np.array(coords[i-1])
                   end = np.array(coords[i])
                   segment_distance = np.linalg.norm(end - start)

                   if accumulated_distance + segment_distance >= current_distance:
                       ratio = (current_distance - accumulated_distance) /
                                                              segment_distance
```

```
                        point_utm = start + ratio * (end - start)

                        # Transform point back to the original CRS
                        point_utm_geometry = gpd.GeoSeries([Point(point_utm)], crs=
                                                          gdf.crs)
                        point = point_utm_geometry.to_crs(original_crs).iloc[0].
                                                          coords[0]

                        # Check if vehicle exceeds the max speed limit sometimes
                        if np.random.uniform() < 0.1:  # Adjust probability as
                                                          needed
                            speed_multiplier = np.random.uniform(1.1, 1.5)  #
                                                              Randomly
                                                              increase speed
                        adjusted_speed_mps = speed_mps * speed_multiplier

                        speed_kmph = adjusted_speed_mps * 3.6  # Convert speed back
                                                           to km/h for
                                                          recording
                        direction = calculate_bearing(start, point)
                        time_increment = calculate_time_increment(segment_distance,

                                                          adjusted_speed_mps
                                                          )
                        timestamp += timedelta(seconds=time_increment)

                        # Append data with formatted timestamp (including date)
                        formatted_timestamp = timestamp.strftime('%Y-%m-%d %H:%M')
                        vehicle_data.append((vehicle_id, step, point[0], point[1],
                                                          speed_kmph,
                                                          formatted_timestamp
                                                          , direction))

                        current_distance += adjusted_speed_mps / num_steps
                        break
                    accumulated_distance += segment_distance

                if accumulated_distance >= current_distance:
                    break

        all_vehicle_data.extend(vehicle_data)

    df = pd.DataFrame(all_vehicle_data, columns=['vehicle_id', 'step', 'x', 'y', '
                                          speed_kmph', 'timestamp', 'direction']
                                          )
    return df
```

A.10 Defining the vehicles in Version-2

In Version-2 of GPS toll based system, there are 4 types of vehicles considered as given below.

```
num_heavy = 12  # Number of heavy vehicles
num_two_wheeler = 15  # Number of two-wheeler vehicles
num_medium = 20  # Number of medium vehicles
num_special = 14  # Number of special vehicles
num_steps = 80  # Number of steps in the simulation
max_speed_kmph = 100  # Maximum speed of vehicles in km/h
```

```
start_time = datetime(2024, 1, 1, 0, 0, 0)  # Define a specific start time if
                                          needed

vehicle_data = simulate_vehicle_movement(gdf, all_coordinates_utm, num_heavy,
                                          num_two_wheeler, num_medium, num_special,
                                           num_steps, max_speed_kmph, start_time)
print(vehicle_data.head())
```

A.11 Displaying the simulated output in Version-2

The simulated vehicle path is then stored in an output csv file in Version-2 of GPS toll based system. The code is shown below:

```
output_path = 'simulated_vehicle_paths_hour_org_org.csv'
vehicle_data.to_csv(output_path, index=False)
print(f"Simulated vehicle paths saved to {output_path}")
```

A.12 Importing the necessary libraries for Version-2

Importing the necessary packages in Python needed for implementing the Version-2 of GPS toll based system.

```
# Import necessary modules
import pandas as pd
import simpy
from geopy.distance import geodesic
import geopandas as gpd
import networkx as nx
from shapely.geometry import Point
from shapely.ops import nearest_points
```

A.13 Loading the necessary data for testing of Version-2

For the testing phase of Version-2, several datasets are loaded. The code is shown below:

```
# Load simulated vehicle paths CSV file
try:
    df = pd.read_csv(r'D:\intel\datasets\26_06simtest.csv')
except FileNotFoundError:
    print("Error: CSV file not found. Please make sure the file path is correct.")
    exit()

# Load shapefile containing road network
try:
    roads = gpd.read_file(r'D:\intel\mergedroad_part\mergedroad.shx')
except FileNotFoundError:
    print("Error: Shapefile not found. Please make sure the file path is correct."
                                          )
    exit()

# Load shapefile containing main roads
try:
    main_roads = gpd.read_file(r'D:\intel\mainroad_part\mainroads_part1.shx')
except FileNotFoundError:
    print("Error: Main roads shapefile not found. Please make sure the file path
                                            is correct.")
    exit()
```

A.14 Creating a graph from the road network in Version-2

The roads considered are now being plotted as a graph for faster implementation using tnetworkx library.

```
# Create a graph from the road network
G = nx.Graph()
for _, road in roads.iterrows():
    line = road.geometry
    for i in range(len(line.coords) - 1):
        start = (line.coords[i][0], line.coords[i][1])
        end = (line.coords[i + 1][0], line.coords[i + 1][1])
        distance = geodesic((start[1], start[0]), (end[1], end[0])).meters
        G.add_edge(start, end, weight=distance)
```

A.15 Defining specific tolls for different vehicles in Version-2

Since there are different types of vehicles, each vehicle requires a different toll fee. The vehicle type serves as the key, paired with its corresponding toll value, forming a dictionary data structure. The code for the same is given below:

```
# Define the fee per kilometer for different vehicle types
FEE_PER_KILOMETER =
    {
    'H': 6,   # Heavy vehicles
    'T': 0.5,   # Two-wheelers
    'M': 2,   # Medium vehicles
    'S': 0   # Special vehicles
    }
```

A.16 Tracking the vehicle and predicting the toll in Version-2

A vehicle class is created that is used to check the path travelled and estimating the fee appropriately based on the type of travel and type of vehicle.

```
class Vehicle:
    def __init__(self, env, vehicle_id):
        self.env = env
        self.vehicle_id = vehicle_id
        self.vehicle_type = vehicle_id[0]
        self.segments = []
        self.data = df[df['vehicle_id'] == vehicle_id]
        self.action = env.process(self.run())

    def run(self):
        for i in range(len(self.data) - 1):
            current = self.data.iloc[i]
            next_point = self.data.iloc[i + 1]
            duration = (next_point['timestamp'] - current['timestamp']).
                                                total_seconds()
```

```
# Get start and end points
start = Point(current['x'], current['y'])
end = Point(next_point['x'], next_point['y'])

# Check if the vehicle is on the main road
if self.is_on_main_road(start) and self.is_on_main_road(end):
    # Find the nearest points on the road network
    nearest_start = self.nearest_road_point(start)
    nearest_end = self.nearest_road_point(end)

    # Add nearest points to the graph if they do not exist
    if nearest_start not in G:
        self.add_node_to_graph(nearest_start)
    if nearest_end not in G:
        self.add_node_to_graph(nearest_end)

    # Calculate the distance along the road network
    distance = self.calculate_road_distance(nearest_start, nearest_end
                                             )

    # Calculate the fee for this segment
    fee = 0 if distance < 30 else (distance / 1000) *
                                             FEE_PER_KILOMETER[self.
                                             vehicle_type]  # Set fee
                                             to 0 if distance < 30
                                             meters

    # Add rush-hour fee if the start time is within rush-hour periods
    if self.is_rush_hour(current['timestamp']):
        fee += RUSH_HOUR_FEE

    # Calculate realistic travel time based on speed
    if 'speed_kmph' in current and current['speed_kmph'] > 0:
        travel_time = (distance / 1000) / current['speed_kmph'] * 3600
                                                 # Convert km/h to
                                                 seconds
        average_speed = current['speed_kmph']  # Use the provided
                                                 speed
    else:
        # Estimate travel time based on distance and average speed
                                                 assumptions
        average_speed = 30  # Assume an average speed of 30 km/h if
                                                 speed data is not
                                                 available
        travel_time = (distance / 1000) / average_speed * 3600

    end_time = current['timestamp'] + pd.Timedelta(seconds=travel_time
                                             )

    # Store segment details
    self.segments.append({
        'vehicle_id': self.vehicle_id,
        'start_time': current['timestamp'],
        'end_time': end_time,
        'start_x': start.x,
        'start_y': start.y,
        'end_x': end.x,
        'end_y': end.y,
                    'distance': distance,
                    'fee': fee,
                    'average_speed': average_speed
                })

                print(f"Vehicle {self.vehicle_id} moving from {start} to {end}.
                                                    Distance: {distance:.2f}
                                                    meters. Fee: {fee:.2f} Rs
                                                    . Travel Time: {
                                                    travel_time:.2f} seconds.
                                                     Average Speed: {
                                                    average_speed:.2f} km/h."
                                                    )

            yield self.env.timeout(duration)

        print(f"Vehicle {self.vehicle_id} total distance traveled on main roads: {
                                            sum(s['distance'] for s in self.
                                            segments):.2f} meters. Total fee:
                                             {sum(s['fee'] for s in self.
                                            segments):.2f} Rs.")

        # Append results for this vehicle to the results list
        results.extend(self.segments)

    def nearest_road_point(self, point):
        # Find the nearest road segment to the given point
        nearest_geom = nearest_points(point, roads.geometry.unary_union)[1]
        return (nearest_geom.x, nearest_geom.y)

    def add_node_to_graph(self, node):
        # Find the nearest existing node in the graph
        nearest_existing_node = min(G.nodes, key=lambda n: geodesic((n[1], n[0]),
                                            (node[1], node[0])).meters)
        distance = geodesic((nearest_existing_node[1], nearest_existing_node[0]),
                                            (node[1], node[0])).meters
        G.add_edge(nearest_existing_node, node, weight=distance)

    def calculate_road_distance(self, start, end):
        try:
            # Find the shortest path along the road network
            path = nx.shortest_path(G, source=start, target=end, weight='weight')
            path_edges = zip(path[:-1], path[1:])
            distance = sum(G[u][v]['weight'] for u, v in path_edges)
            return distance
        except nx.NetworkXNoPath:
            # If no path is found, fall back to geodesic distance
            return geodesic(start, end).meters

    def is_on_main_road(self, point):
        # Check if the given point is within the buffer of main roads
        return any(point.within(buffer) for buffer in main_roads_buffer)

    def is_rush_hour(self, timestamp):
        # Check if the timestamp is within rush-hour periods
        morning_rush_hour = timestamp.time() >= pd.Timestamp('08:00:00').time()
                                            and timestamp.time() <= pd.
                                            Timestamp('10:00:00').time()
        evening_rush_hour = timestamp.time() >= pd.Timestamp('16:00:00').time()
                                                  and timestamp.time() <= pd.
                                                  Timestamp('18:00:00').time()
        return morning_rush_hour or evening_rush_hour
```

A.17 Creating vehicles in Version-2

Setting up the simulation environment and creating the vehicles in Version-2.

```
# Set up the simulation environment
env = simpy.Environment()

# Create vehicles
vehicles = [Vehicle(env, vehicle_id) for vehicle_id in df['vehicle_id'].unique()]

# Run the simulation
env.run()
```

A.18 Saving the dataframe in Version-2

A dataframe is created using the results in Version-2. This dataframe is then saved as a csv file.

```
# Create a DataFrame from the results
results_df = pd.DataFrame(results)

# Save the results to a CSV file
results_df.to_csv(r'D:\intel\datasets\26_06_simorg_pt_test5.csv', index=False)
env.run()
```