INTERNSHIP PROJECT REPORT

SUBMITTED IN COMPLETE FULFILLMENT OF THE REQUIREMENTS STATED BY THE NATIONAL TECHNICAL RESEARCH ORGANIZATION(NTRO)

Submitted by:

KUMAR SHASWAT (2K21/SE/105)

Under the supervision of

SDC MR. ABHINAV MATHUR

DELHI TECHNOLOGICAL UNIVERSITY

(FORMERLY Delhi College of Engineering) Bawana Road, Delhi-110042

INDEX

DELHI TECHNOLOGICAL UNIVERSITY

(FORMERLY Delhi College of Engineering)

Bawana Road, Delhi-110042

<u>CANDIDATE'S DECLARATION</u>

We hereby certify that the work, which is presented in the Project entitled "" in fulfilment of the requirement of the National Technical Research Organisation and submitted to the Training and Placement Cell, Delhi Technological University, Delhi is an authentic record of my own.

The work presented in this report has not been submitted and not under consideration for the award for any other course/degree of this or any other Institute/University.

KUMAR SHASWAT
(2K21/SE/105)

DELHI TECHNOLOGICAL UNIVERSITY

(FORMERLY Delhi College of Engineering)

Bawana Road, Delhi-110042

<u>CERTIFICATE</u>

I hereby certify that the project Dissertation titled “” which is submitted by Kumar Shaswat (2K21/SE/105) comprises original work and has not been submitted in part or full for any Course/Degree to this university or elsewhere as per the candidate’s declaration. Delhi Technological University, Delhi in complete fulfilment of the requirement for the award of the degree of the Bachelor of Technology, is a record of the project work carried out by the students under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: Delhi

DELHI TECHNOLOGICAL UNIVERSITY

(FORMERLY Delhi College of Engineering)

Bawana Road, Delhi-110042

ACKNOWLEDGEMENT

In performing our project, I had to take the help and guideline of some respected persons, who deserve our greatest gratitude. The completion of this assignment gives me much pleasure. I would like to show our gratitude to SDC Mr. Abhinav Mathur, for giving me good guidance for reporting throughout numerous consultations. We would also like to extend our deepest gratitude to all those who have directly and indirectly guided us in writing this assignment.

Many people, especially Mr. Abhinav Mathur and other esteemed scientists, have made valuable comment suggestions on this project which gave me inspiration to improve my assignment. I thank all the people for their help directly and indirectly to complete my assignment.

In addition, I would like to thank the National Technical Research Organisation (NTRO), for giving me the opportunity to work on this project.

INTRODUCTION

The Ramer-Douglas-Peucker (RDP) algorithm is a method used to simplify curves or polylines by reducing the number of points that define them, while still preserving the overall shape and essential features. This technique is particularly valuable in fields such as geographic information systems (GIS), where it can simplify the representation of complex routes or boundaries, making them easier to store, process, and visualize. In cartography, for example, RDP is often used to reduce the complexity of coastline or road map data without losing critical details. It is also widely employed in computer graphics for compressing and smoothing lines in vector images, and in robotics for path simplification, helping robots and drones navigate more efficiently. Another common use is in time-series data, where RDP can be applied to reduce noise and simplify data patterns, making trends more apparent while minimizing the computational load. Overall, the RDP algorithm's ability to streamline large datasets while maintaining accuracy makes it highly applicable in areas requiring data reduction and analysis.

PYTHON

Python is a versatile, high-level programming language known for its emphasis on code readability, largely achieved through the use of significant indentation.

It is dynamically typed and uses garbage collection to manage memory. Python supports a range of programming paradigms, including procedural, object-oriented, and functional styles. It is frequently described as a "batteries included" language due to its extensive standard library.

Python consistently ranks among the most popular programming languages and is widely adopted in the machine learning field.

Python was invented in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC programming language, which was inspired by SETL, capable of exception handling and interfacing with the Amoeba operating system . Its implementation began in December 1989.Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018, when he announced his "permanent vacation" from his responsibilities as Python's "benevolent dictator for life" (BDFL), a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker (he's since come out of retirement and is self-titled "BDFL-emeritus"). In January 2019, active Python core developers elected a five-member Steering Council to lead the project.

Python 2.0 was released on 16 October 2000, with many major new features such as list comprehensions, cycle-detecting garbage collection, reference counting, and Unicode support. Python 3.0 was released on 3 December 2008, with many of its major features backported to Python 2.6.x and 2.7.x. Releases of Python 3 include the 2 to 3 utility, which automates the translation of Python 2 code to Python 3.

Python is a multi-paradigm programming language. Object-oriented programming and structured programming are fully supported, and many of their features support functional programming and aspect-oriented programming (including metaprogramming and metaobjects).Many other paradigms are supported via extensions, including design by contract and logic programming.

Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. It uses dynamic name resolution (late binding), which binds method and variable names during program execution.

Its design offers some support for functional programming in the Lisp tradition. It has filter,mapandreduce functions; list comprehensions, dictionaries, sets, and generator expressions. The standard library has two modules (itertools and functools) that implement functional tools borrowed from Haskell and Standard ML.

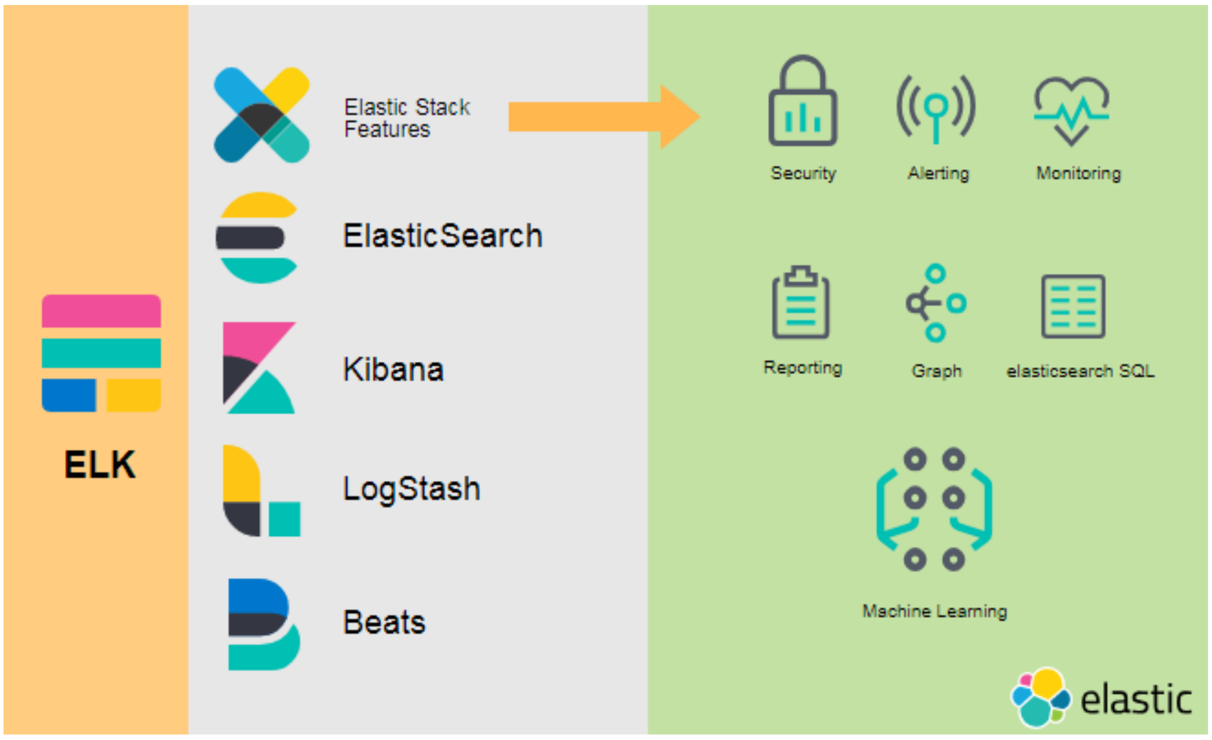
Python is meant to be an easily readable language. Its formatting is visually uncluttered and often uses English keywords where other languages use punctuation. Unlike many other languages, it does not use curly brackets to delimit blocks, and semicolons after statements are allowed but rarely used. It has fewer syntactic exceptions and special cases than C or Pascal.

Since 2003, Python has consistently ranked in the top ten most popular programming languages in the TIOBE Programming Community Index where as of December 2022 it was the most popular language (ahead of C, C++, and Java). It was selected as Programming Language of the Year (for "the highest rise in ratings in a year") in 2007, 2010, 2018, and 2020 (the only language to have done so four times as of 2020).

Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram, Spotify, and some smaller entities like Industrial Light & Magic and ITA. The social news networking site Reddit was written mostly in Python.

ELASTIC SEARCH

At its foundation, Elasticsearch is built on Apache Lucene, a powerful text search library. It enhances Lucene's functionality by providing a distributed, RESTful search engine that is easy to scale and manage. Elasticsearch indexes data efficiently, enabling near-instant search and retrieval, which makes it ideal for applications that require fast access to large datasets.



Key Features and Benefits

1. **Real-Time Search and Analytics**: Elasticsearch's real-time search is one of its standout features. Once data is indexed, it is immediately available for search and analysis, making it indispensable for applications like logging, monitoring, and analytics where timely access to data is critical.
2. **Distributed Architecture**: Elasticsearch's architecture is natively distributed, allowing it to scale horizontally across multiple nodes and clusters. This design ensures it can manage

large volumes of data and high query loads, making it suitable for both small-scale and enterprise-level operations.

3. **Full-Text Search**: Elasticsearch excels in full-text search capabilities, allowing users to perform advanced queries on text data. It supports features like fuzzy matching, partial matching, and relevance scoring, ensuring the most pertinent results are returned from large sets of text.
4. **RESTful API**: The RESTful API of Elasticsearch simplifies integration with other systems and applications. This API offers a clear, consistent interface for interacting with Elasticsearch, making it easy for developers familiar with HTTP and JSON to use.
5. **Scalability and Flexibility**: Elasticsearch can scale efficiently by adding more nodes, allowing it to handle growing data volumes and query demands. Its schema-free design also provides flexibility, supporting a range of data types from structured logs to unstructured text.
6. **Aggregation Framework**: Elasticsearch's aggregation framework offers robust data analysis capabilities. It allows users to perform complex aggregations and calculations, such as computing metrics, creating histograms, and generating statistical summaries, all within the search engine.

Common Use Cases

Elasticsearch is used across a variety of industries and applications. In e-commerce, it powers search functions on websites, enabling users to find products quickly and accurately. In log and event data management, Elasticsearch is widely utilized for aggregating and analysing logs from multiple sources, providing insights into system performance and security. In the media industry, it supports content discovery and recommendation engines, helping users find relevant articles, videos, and other media content.

Integration with the Elastic Stack

Elasticsearch is part of the Elastic Stack (previously known as the ELK Stack), which includes Logstash for data ingestion, Kibana for visualization, and Beats for lightweight data collection. Together, this suite of tools provides a comprehensive solution for data collection, storage, analysis, and visualization, making it easier to create end-to-end data solutions.

Conclusion

Elasticsearch is a powerful and versatile search and analytics engine, capable of handling large-scale data with speed and efficiency. Its real-time search, distributed architecture, and flexible data handling make it an essential tool for awide array of applications, from enterprise search to log analytics. As data continues to grow in size and complexity, Elasticsearch’s ability to deliver fast, relevant, and actionable insights remains a valuable asset for organizations aiming to maximize the potential of their data.

GraphQL

GraphQL is a data query and manipulation language for APIs, that allows a client to specify what data it needs ("declarative data fetching"). A GraphQL server can fetch data from separate sources for a single client query and present the results in a unified graph, so it is not tied to any specific database or storage engine.
The associated GraphQL runtime engine is open-source.
Facebook started GraphQL development in 2012 and released it as open source in 2015. In 2018, GraphQL was moved to the newly established GraphQL Foundation, hosted by the non-profit Linux Foundation.

On 9 February 2018, the GraphQL Schema Definition Language became part of the specification.

Many popular public APIs adopted GraphQL as the default way to access them. These include public APIs of Facebook, GitHub, Yelp, Shopify and Google Directions API.

GraphQL supports reading, writing (mutating), and subscribing to changes to data (real time updates – commonly implemented using WebSockets). A GraphQL service is created by defining types with fields, then providing functions to resolve the data for each field. The types and fields make up what is known as the schema definition. The functions that retrieve and map the data are called resolvers.

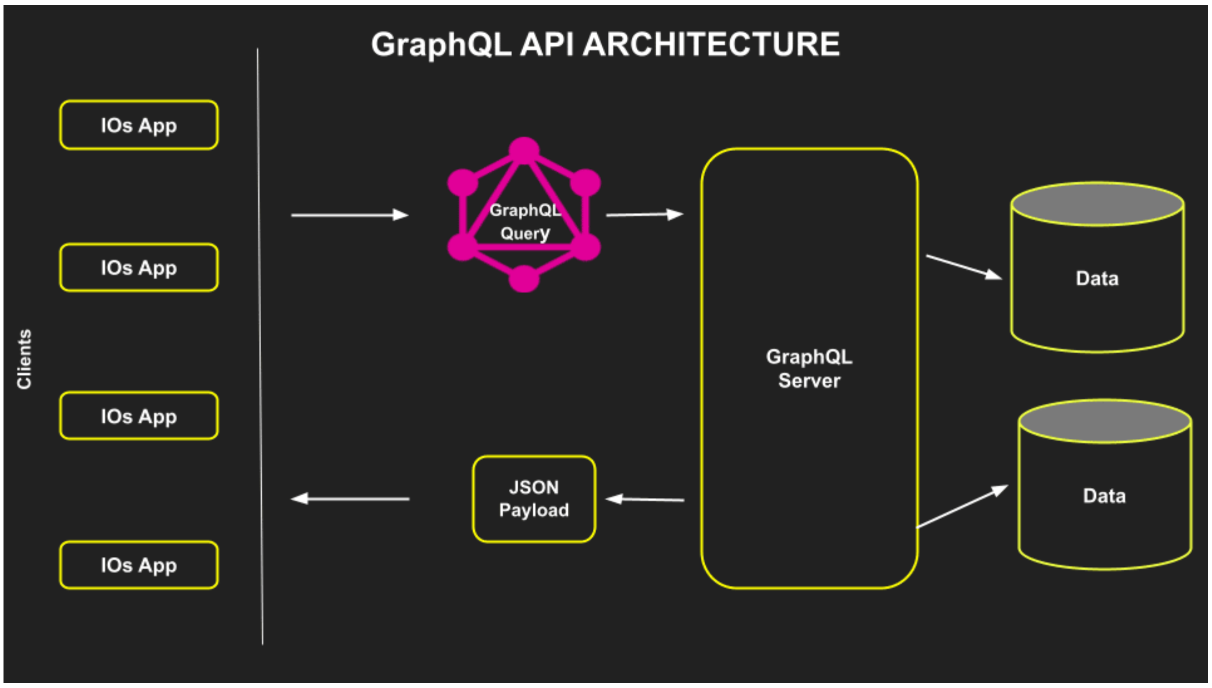
After being validated against the schema, a GraphQL query is executed by the server. The server returns a result that mirrors the shape of the original query, typically as JSON.

Testing

GraphQL APIs can be tested manually or with automated tools issuing GraphQL requests and verifying the correctness of the results. Automatic test generation is also possible.[12] New requests may be produced through search-based techniques due to a typed schema and introspection capabilities.

Some of the software tools used for testing GraphQL implementations include Postman, GraphiQL, Apollo Studio, GraphQL Editor, and Step CI.

GraphQL services can be written in any language. Since we can't rely on a specific programming language syntax, like JavaScript, to talk about GraphQL schemas, we'll define our own simple language. We'll use the "GraphQL schema language" - it's similar to the query language, and allows us to talk about GraphQL schemas in a language-agnostic way.



The language consists of the following :

Character - is a GraphQL Object Type, meaning it's a type with some fields. Most of the types in your schema will be object types.

Name and AppearsIn - are fields on the Character type. That means that name and AppearsIn are the only fields that can appear in any part of a GraphQL query that operates on the Character type.

String -is one of the built-in scalar types - these are types that resolve to a single scalar object, and can't have sub-selections in the query. We'll go over scalar types more later.

String!- means that the field is non-nullable, meaning that the GraphQL service promises to always give you a value when you query this field. In the type language, we'll represent those with an exclamation mark.

-Episode! represents an array of Episode objects. Since it is also non-nullable, you can always expect an array (with zero or more items) when you query the AppearsIn field. And since Episode! is also non-nullable, you can always expect every item of the array to be an Episode object.

ALGORITHM

The Ramer–Douglas–Peucker algorithm, also known as the Douglas–Peucker algorithm and iterative end-point fit algorithm, is an algorithm that decimates a curve composed of line segments to a similar curve with fewer points. It was one of the earliest successful algorithms developed for cartographic generalization. It produces the most accurate generalization, but it is also more time-consuming.

The starting curve is an ordered set of points or lines and the distance dimension $\varepsilon > 0$.

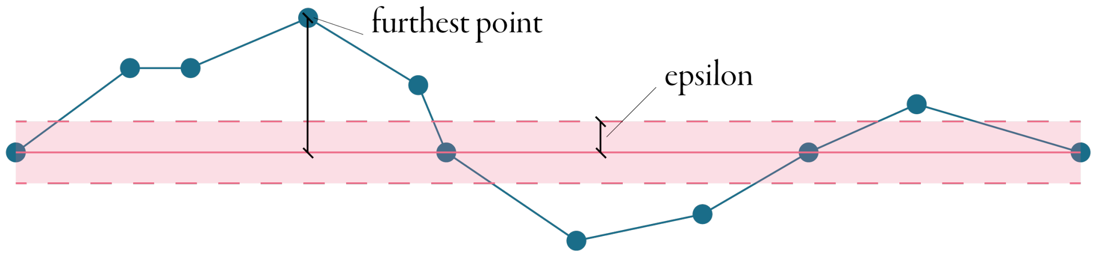
The algorithm recursively divides the line. Initially it is given all the points between the first and last point. It automatically marks the first and last point to be kept. It then finds the point that is farthest from the line segment with the first and last points as end points; this point is always farthest on the curve from the approximating line segment between the end points. If the point is closer than ε to the line segment, then any points not currently marked to be kept can be discarded without the simplified curve being worse than ε.

If the point farthest from the line segment is greater than ε from the approximation then that point must be kept. The algorithm recursively calls itself with the first point and the farthest point and then with the farthest point and the last point, which includes the farthest point being marked as kept.

When the recursion is completed a new output curve can be generated consisting of all and only those points that have been marked as kept.

Non-parametric Ramer–Douglas–Peucker

The choice of ε is usually user-defined. Like most line fitting, polygonal approximation or dominant point detection methods, it can be made non-parametric by using the error bound due to digitization and quantization as a termination condition.



STEPS

To implement the Ramer-Douglas-Peucker (RDP) algorithm using the perpendicular distance approach, we can calculate the perpendicular distance of a point from a line segment based on the concept of the area of a triangle and the height of that triangle. This method is geometrically intuitive and works as follows:

Given three points:

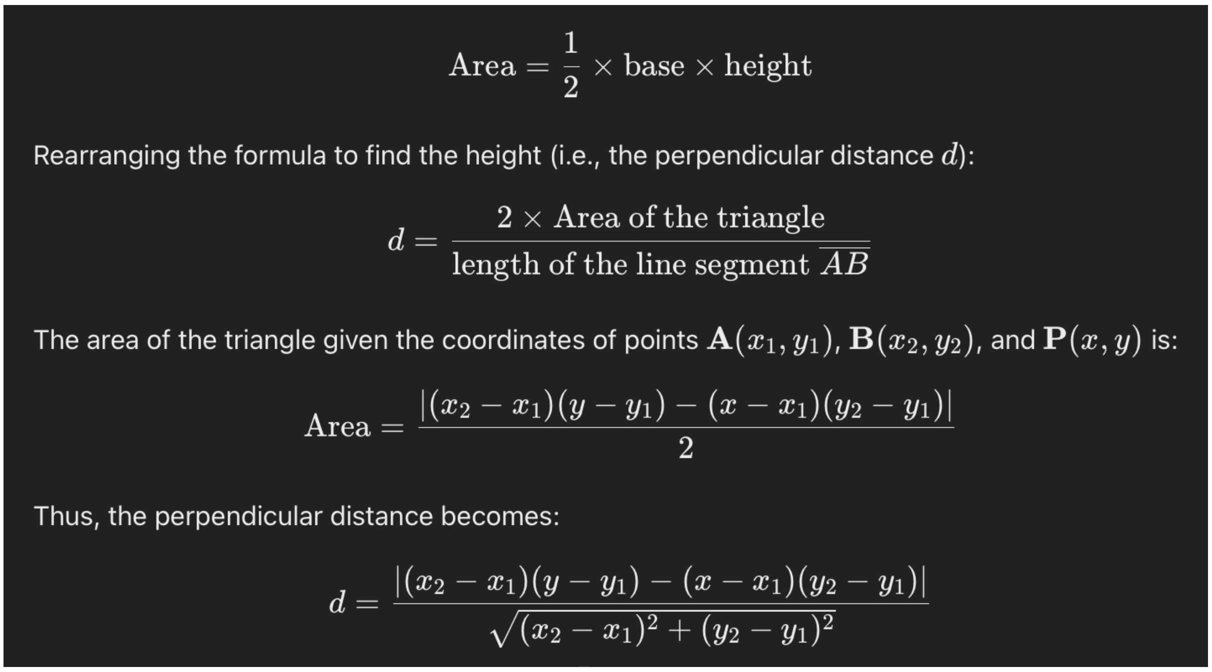
1. **Line start point** A(x1,y1)
2. **Line end point** B(x2,y2)
3. **Any point** P(x,y)

The perpendicular distance from point PP to the line segment AB can be calculated as the height of a triangle where:

- AA and BB are the base points of the triangle,
- PP is the vertex opposite the base AB.

The formula to calculate the perpendicular distance dd is derived from the area of the triangle formed by the points A,B,PA,B,P. The area of the triangle is:

Here's how you can implement the RDP algorithm using this perpendicular distance (area/height) method



$$\text{Area} = \frac{1}{2} \times \text{base} \times \text{height}$$

Rearranging the formula to find the height (i.e., the perpendicular distance d):

$$d = \frac{2 \times \text{Area of the triangle}}{\text{length of the line segment } \overline{AB}}$$

The area of the triangle given the coordinates of points $\mathbf{A}(x_1, y_1)$, $\mathbf{B}(x_2, y_2)$, and $\mathbf{P}(x, y)$ is:

$$\text{Area} = \frac{|(x_2 - x_1)(y - y_1) - (x - x_1)(y_2 - y_1)|}{2}$$

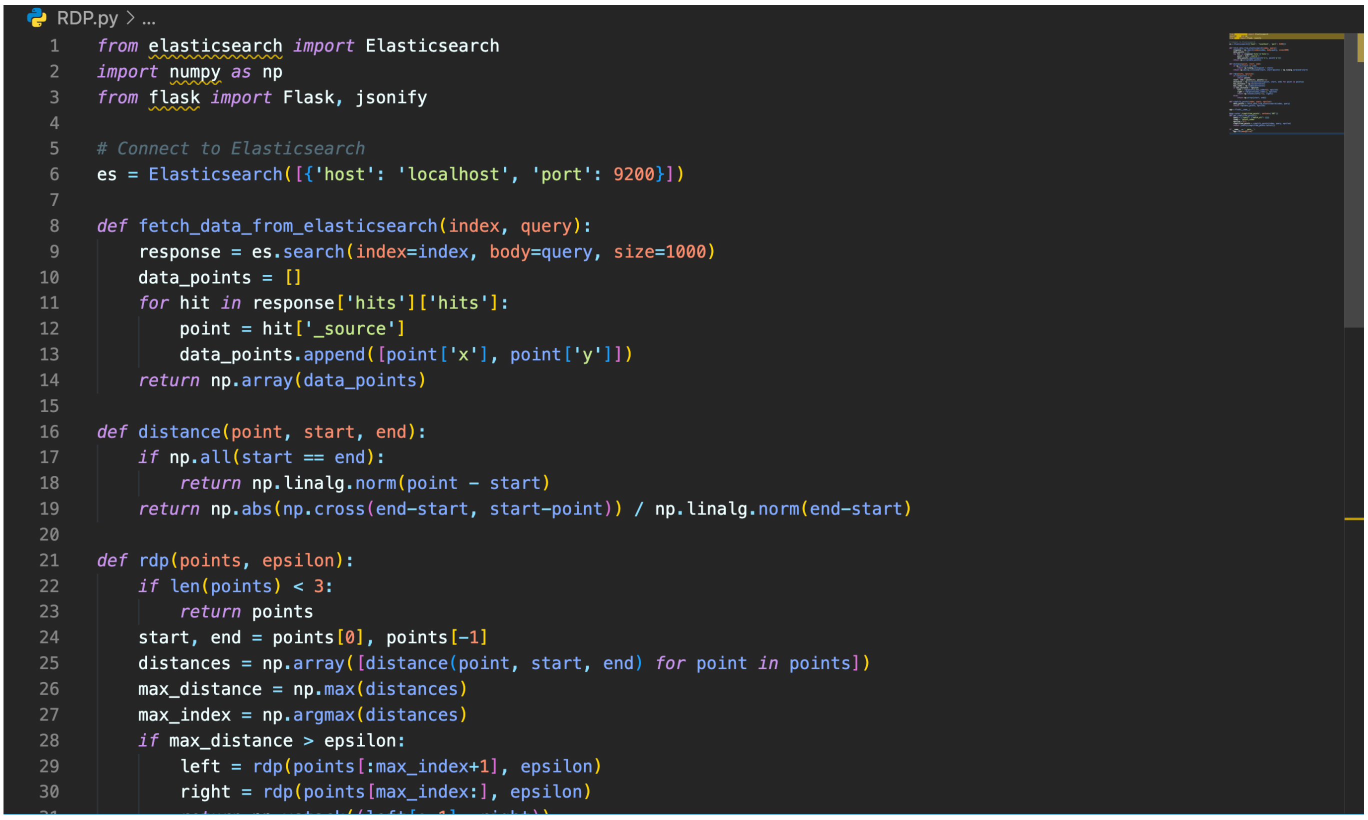
Thus, the perpendicular distance becomes:

$$d = \frac{|(x_2 - x_1)(y - y_1) - (x - x_1)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

:

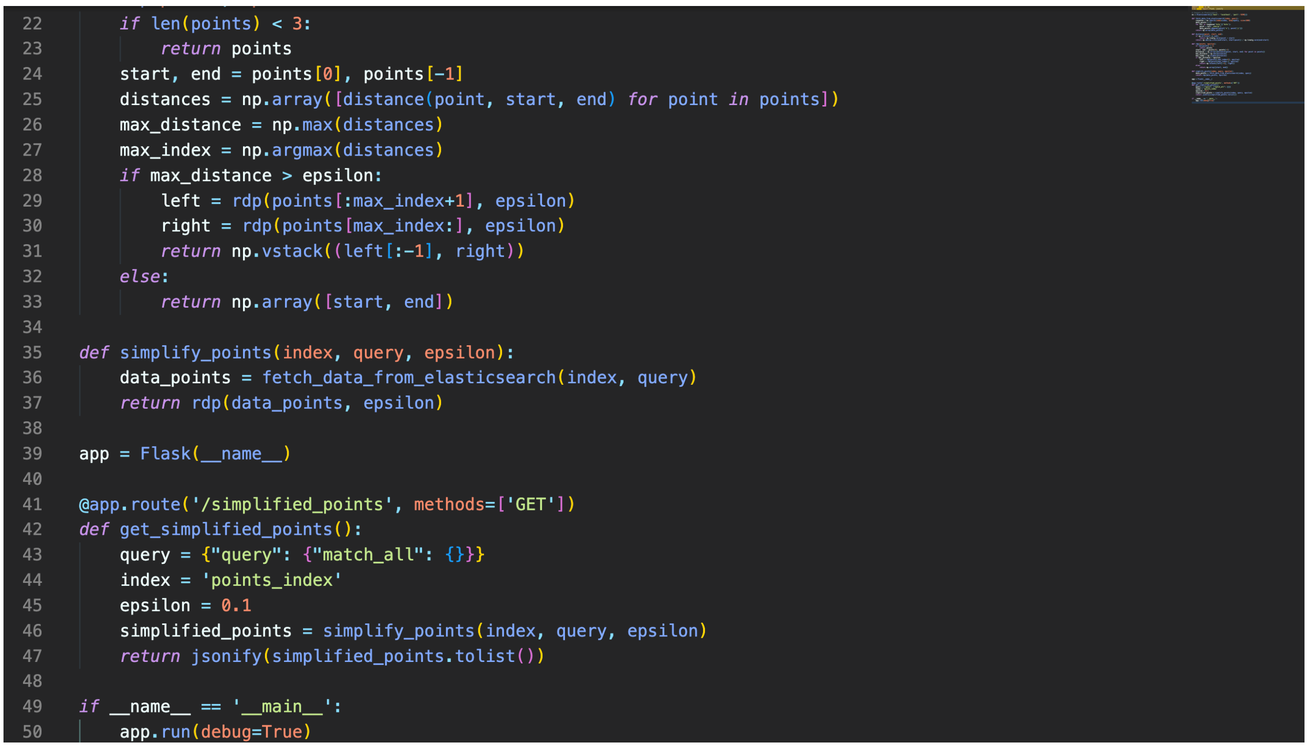
IMPLEMENTATION

Here’s a Python implementation of the Ramer-Douglas-Peucker (RDP) algorithm:



RDP.py > ...

```python
from elasticsearch import Elasticsearch
import numpy as np
from flask import Flask, jsonify

# Connect to Elasticsearch
es = Elasticsearch([{'host': 'localhost', 'port': 9200}])

def fetch_data_from_elasticsearch(index, query):
    response = es.search(index=index, body=query, size=1000)
    data_points = []
    for hit in response['hits']['hits']:
        point = hit['_source']
        data_points.append([point['x'], point['y']])
    return np.array(data_points)

def distance(point, start, end):
    if np.all(start == end):
        return np.linalg.norm(point - start)
    return np.abs(np.cross(end-start, start-point)) / np.linalg.norm(end-start)

def rdp(points, epsilon):
    if len(points) < 3:
        return points
    start, end = points[0], points[-1]
    distances = np.array([distance(point, start, end) for point in points])
    max_distance = np.max(distances)
    max_index = np.argmax(distances)
    if max_distance > epsilon:
        left = rdp(points[:max_index+1], epsilon)
        right = rdp(points[max_index:], epsilon)
        return np.vstack((left[:-1], right))
```

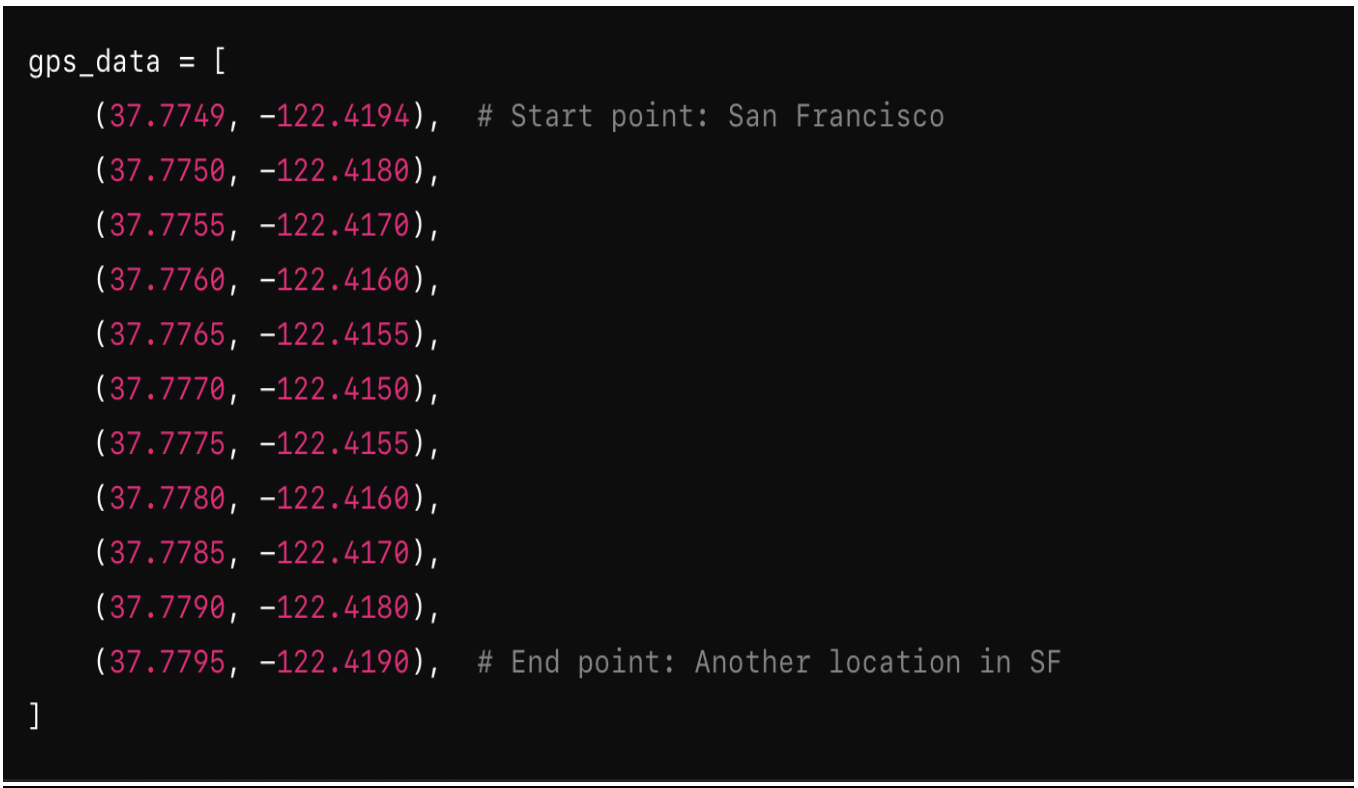


```
    if len(points) < 3:
        return points
    start, end = points[0], points[-1]
    distances = np.array([distance(point, start, end) for point in points])
    max_distance = np.max(distances)
    max_index = np.argmax(distances)
    if max_distance > epsilon:
        left = rdp(points[:max_index+1], epsilon)
        right = rdp(points[max_index:], epsilon)
        return np.vstack((left[:-1], right))
    else:
        return np.array([start, end])

def simplify_points(index, query, epsilon):
    data_points = fetch_data_from_elasticsearch(index, query)
    return rdp(data_points, epsilon)

app = Flask(__name__)

@app.route('/simplified_points', methods=['GET'])
def get_simplified_points():
    query = {"query": {"match_all": {}}}
    index = 'points_index'
    epsilon = 0.1
    simplified_points = simplify_points(index, query, epsilon)
    return jsonify(simplified_points.tolist())

if __name__ == '__main__':
    app.run(debug=True)
```

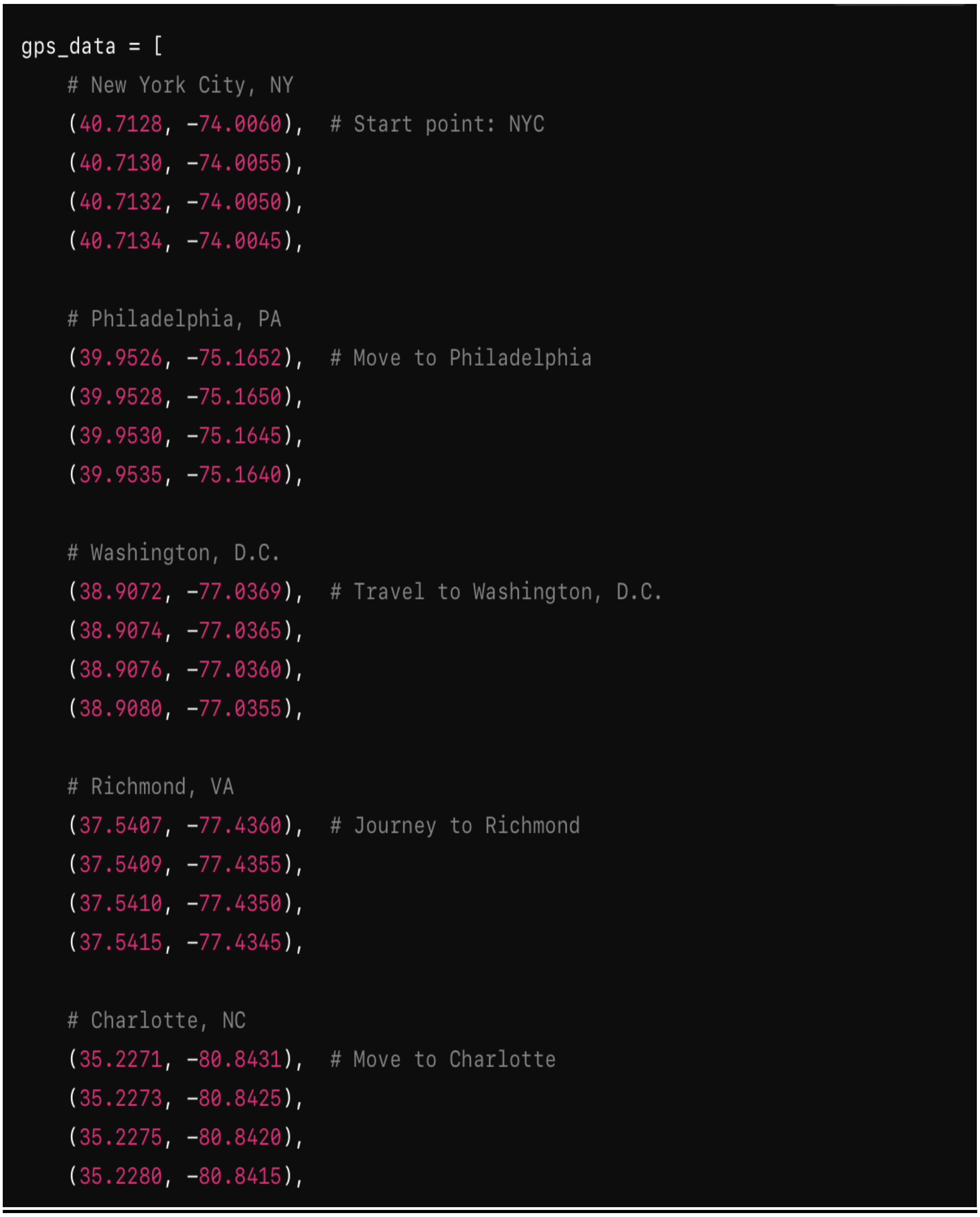
Steps of the Algorithm:

1. **Initial Input:** The algorithm starts with a set of points that define the curve. The first point P1and the last point Pn are considered critical and are always retained.
2. **Finding the Most Distant Point:** A straight line is drawn between P1 and Pn. The algorithm then checks every intermediate point Pi to find the point that is farthest from the line P1 Pn . This distance is referred to as the **perpendicular distance** or **orthogonal distance**.
3. **Tolerance Check:** If the maximum perpendicular distance found is greater than a given tolerance ϵ, that point is considered important for maintaining the shape of the curve. The polyline is split at this point, and the algorithm is recursively applied to the two sub-curves: from P1 to Pi , and from Pi to Pn
4. **Recursive Application:** This process repeats, breaking the curve into smaller and smaller segments, until all points within each segment have a perpendicular distance less than the tolerance ϵ.
5. **Termination:** Once no point's distance exceeds ϵ

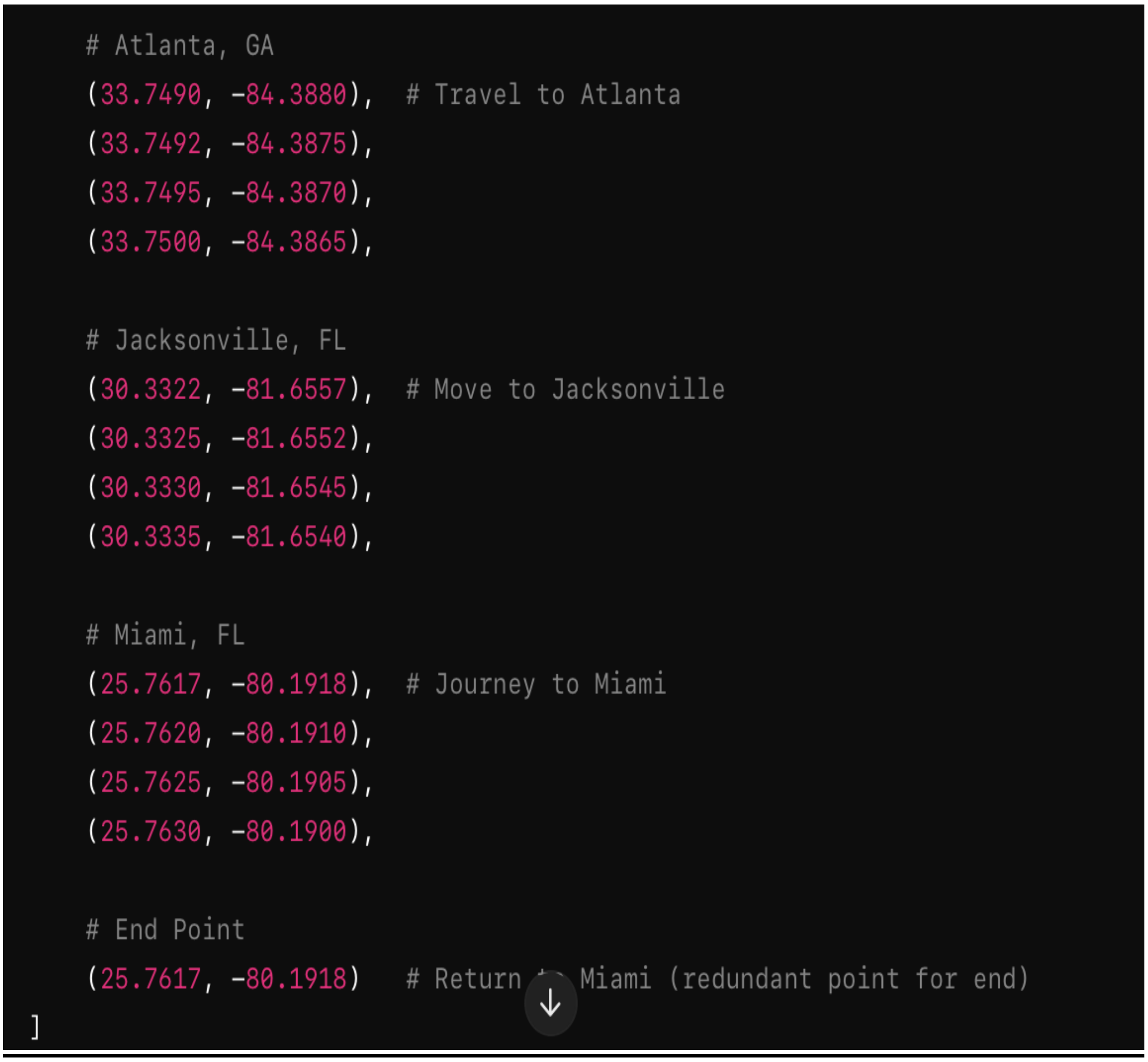
DATA



```
gps_data = [
    (37.7749, -122.4194),  # Start point: San Francisco
    (37.7750, -122.4180),
    (37.7755, -122.4170),
    (37.7760, -122.4160),
    (37.7765, -122.4155),
    (37.7770, -122.4150),
    (37.7775, -122.4155),
    (37.7780, -122.4160),
    (37.7785, -122.4170),
    (37.7790, -122.4180),
    (37.7795, -122.4190),  # End point: Another location in SF
]
```

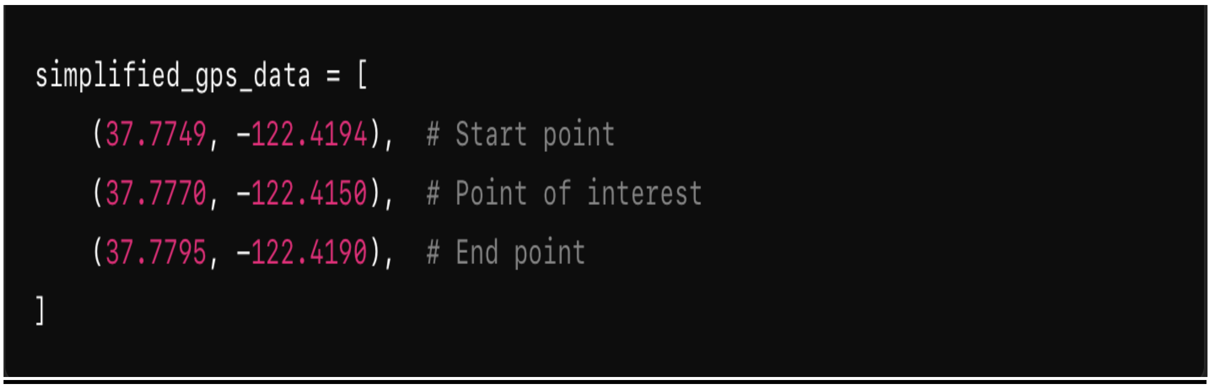


```
gps_data = [
    # New York City, NY
    (40.7128, -74.0060),  # Start point: NYC
    (40.7130, -74.0055),
    (40.7132, -74.0050),
    (40.7134, -74.0045),

    # Philadelphia, PA
    (39.9526, -75.1652),  # Move to Philadelphia
    (39.9528, -75.1650),
    (39.9530, -75.1645),
    (39.9535, -75.1640),

    # Washington, D.C.
    (38.9072, -77.0369),  # Travel to Washington, D.C.
    (38.9074, -77.0365),
    (38.9076, -77.0360),
    (38.9080, -77.0355),

    # Richmond, VA
    (37.5407, -77.4360),  # Journey to Richmond
    (37.5409, -77.4355),
    (37.5410, -77.4350),
    (37.5415, -77.4345),

    # Charlotte, NC
    (35.2271, -80.8431),  # Move to Charlotte
    (35.2273, -80.8425),
    (35.2275, -80.8420),
    (35.2280, -80.8415),
```

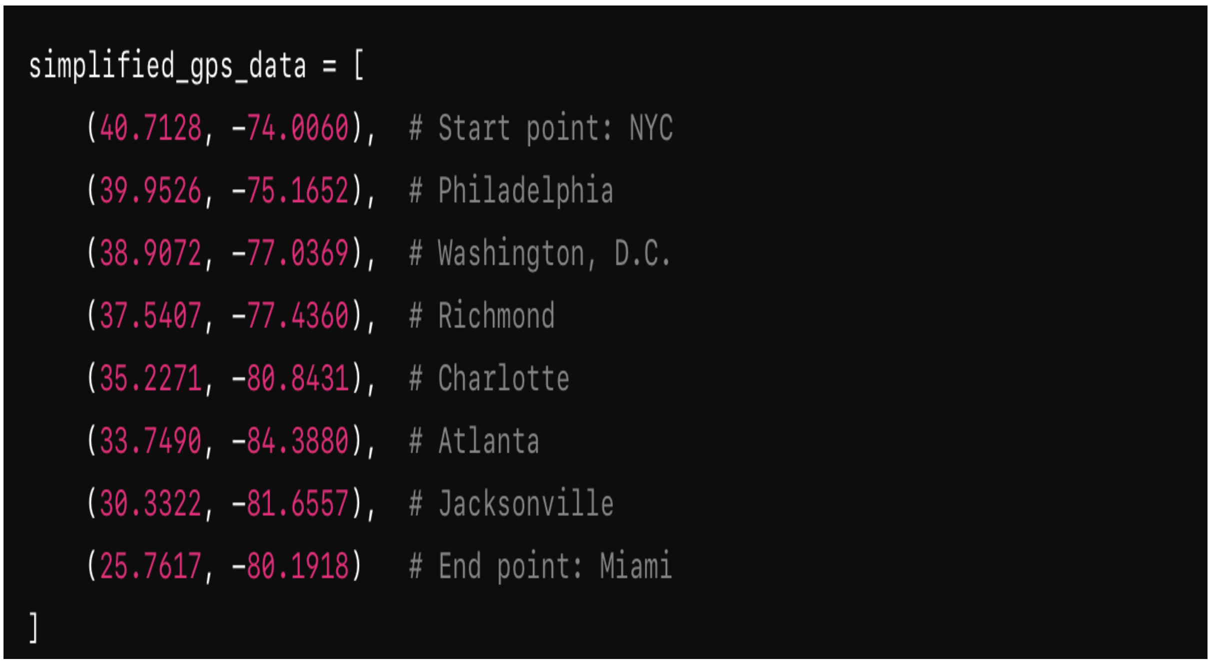


```
    # Atlanta, GA
    (33.7490, -84.3880),  # Travel to Atlanta
    (33.7492, -84.3875),
    (33.7495, -84.3870),
    (33.7500, -84.3865),

    # Jacksonville, FL
    (30.3322, -81.6557),  # Move to Jacksonville
    (30.3325, -81.6552),
    (30.3330, -81.6545),
    (30.3335, -81.6540),

    # Miami, FL
    (25.7617, -80.1918),  # Journey to Miami
    (25.7620, -80.1910),
    (25.7625, -80.1905),
    (25.7630, -80.1900),

    # End Point
    (25.7617, -80.1918)   # Return to Miami (redundant point for end)
]
```

OUTPUT



```
simplified_gps_data = [
    (37.7749, -122.4194),  # Start point
    (37.7770, -122.4150),  # Point of interest
    (37.7795, -122.4190),  # End point
]
```



```
simplified_gps_data = [
    (40.7128, -74.0060),  # Start point: NYC
    (39.9526, -75.1652),  # Philadelphia
    (38.9072, -77.0369),  # Washington, D.C.
    (37.5407, -77.4360),  # Richmond
    (35.2271, -80.8431),  # Charlotte
    (33.7490, -84.3880),  # Atlanta
    (30.3322, -81.6557),  # Jacksonville
    (25.7617, -80.1918)   # End point: Miami
]
```

EXPLANATION

The **input data** consists of a series of closely spaced GPS coordinates representing a path in San Francisco.

After running the **RDP algorithm**, the **output** includes only the essential points that define the path's shape, effectively reducing the total number of points while still capturing the trajectory's important features.

Input Data: The original dataset consists of GPS coordinates representing a journey through several major cities in the eastern United States, with many closely spaced points.

Output Data: After applying the RDP algorithm, the simplified set captures only the critical points that define the journey's path, resulting in a much smaller dataset while maintaining the essential geographical transitions.

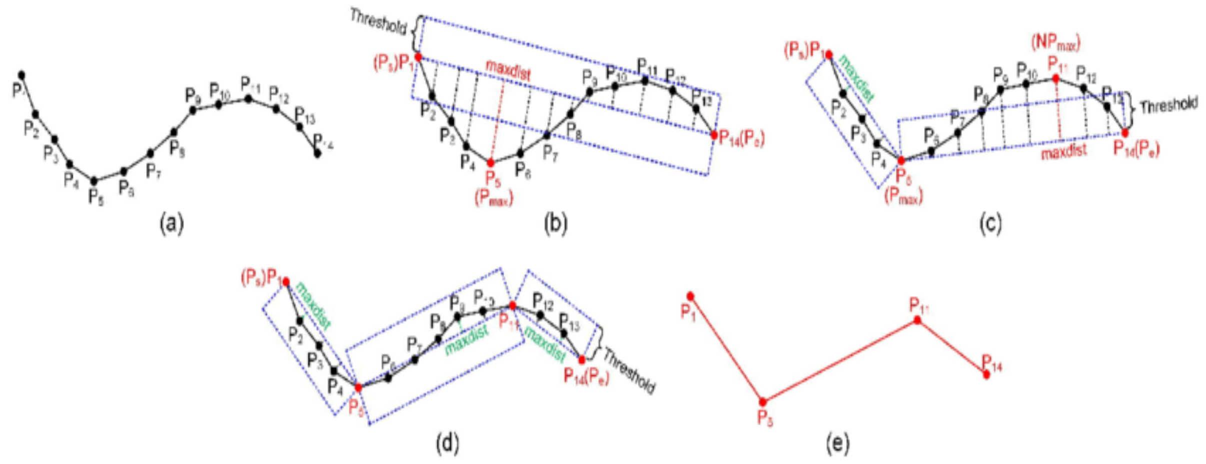
VISUALIZATION

If you were to visualize both the original and simplified paths on a map, you would see that the simplified path still follows the general trajectory of the original but with significantly fewer points, making it clearer and more manageable for display purposes.

This simplification is especially beneficial for applications like mapping services or navigation systems, where clarity and efficiency in data representation are paramount.

CONCLUSION

In conclusion, the Ramer-Douglas-Peucker (RDP) algorithm plays a crucial role in efficiently processing and visualizing large datasets, particularly in the context of GPS simplification. By integrating the algorithm with modern data retrieval systems like Elasticsearch, it can fetch complex GPS data and simplify it, reducing the number of data points while preserving essential features and the overall shape of the trajectory. This is particularly valuable for applications in navigation and mapping, where GPS data often consists of a high volume of points that can lead to performance issues and cluttered visualizations.



Once the simplification process is complete, the refined GPS data is sent to the front end of applications for display. This integration enhances the performance and responsiveness of user interfaces, allowing users to interact with clear and coherent representations of

GPS trajectories. By combining the strengths of Elasticsearch for efficient data retrieval and the RDP algorithm for GPS data simplification, organizations can deliver rich, informative visualizations that facilitate better navigation and decision-making. As the demand for efficient handling of location-based data continues to grow, this synergy becomes increasingly vital in various applications, from real-time tracking systems to advanced mapping services, ultimately improving user experience and operational effectiveness.