

MEOS: An Open Source Library for Mobility Data Management

Esteban Zimányi
 Université libre de Bruxelles
 Brussels, Belgium
 esteban.zimanyi@ulb.be

Mariana Duarte
 Université libre de Bruxelles
 Brussels, Belgium
 mariana.machado.garcez.duarte@ulb.be

Víctor Diví
 Universitat Politècnica de Catalunya
 Barcelona, Spain
 victor.divi@upc.edu

ABSTRACT

The increasing prevalence of mobility data in diverse applications such as traffic management requires specialized tools for manipulating it. This paper introduces MEOS (Mobility Engine Open Source), a versatile C library designed explicitly for managing and processing mobility data. MEOS is a highly adaptable solution, providing a single code base that can be readily adapted to multiple programming languages, including Python and SQL, numerous computing environments, from edge to cloud computing, from batch to stream processing. We further extend our discussion to PyMEOS, a specialized Python library developed as a binding to MEOS, offering additional functionality that enhances the analytic capacity with data visualization and data frame integration. This demonstration comprises a series of queries using two different languages, namely, MEOS (C) and PyMEOS (Python).

1 INTRODUCTION

Advances in tracking technologies have led to the growing trend of utilizing mobility data in a myriad of applications, including traffic management. This requires specialized tools to handle the huge volumes of mobility data generated. This paper demonstrates the use of MEOS (Mobility Engine Open Source)¹ for trajectory manipulation. MEOS is an open-source C library engineered for mobility data management. It provides a single source code that can be readily adapted to different environments including, but not limited to, Python and SQL, multiple computing environments, from edge to cloud computing, from batch to stream processing. A single code simplifies maintenance processes, making code modification, updating and debugging straightforward. It can also reduce errors and offer faster implementation of new features. MEOS provides an alternative to existing moving objects libraries [2, 4, 6] while maintaining a data format with a lossless high compression rate that may reach up to 450% [7], depending on the data source. We present an example of MEOS' adaptability with an integration in Python called PyMEOS.² PyMEOS contains all functionalities present in MEOS but also enhances the analytic capacity with extra features such as data visualization and data frame integration.

2 CONTRIBUTIONS

We present MEOS, a new library for managing and analyzing moving objects. MEOS library leverages decoupling computing from storage. The decoupled architecture allows the balancing of storage and computing requirements based on workload demands. This strategy ensures that the library can be configured

¹<https://libmeos.org>

²<https://github.com/MobilityDB/PyMEOS>

© 2024 Copyright held by the owner/author(s). Published in Proceedings of the 27th International Conference on Extending Database Technology (EDBT), 25th March–28th March, 2024, ISBN 978-3-89318-095-0 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

to deploy the most suitable storage for the data requirements and the optimal computing resources for processing tasks, thereby enhancing overall system performance. Lastly, the separation of computing and storage enhances MEOS's adaptability. As a result, it can easily utilize different storage and computing resources based on the task requirements.

3 MEOS

The MEOS library provides functions in multiple programming languages and environments (Fig. 1). The library can be built on M1, ARM, and Intel hardware on Ubuntu, MacOS, and Windows operating systems. Furthermore, it can be adapted to various programming languages such as C, JavaScript, Java, and Python.



Figure 1: MEOS compatibility layers. Available languages, compatible systems, operating systems, and hardware

3.1 MEOS Components

MEOS' main components are presented as follows:

- **Input/Output Functions:** MEOS supports input and output operations for various formats, including GeoJSON, Well-Known Text (WKT) and Well-Known Binary (WKB). It also implements expandable data structures, enabling data streaming and batch processing.
- **Data Structure/Operations:** The system augments data operations with additional support for temporal properties, lifted predicates, and aggregations. It implements the types tbool, tfloat, tint, and tsequence. These data types represent the temporal values of each base type unfold over time. For example, a tbool encapsulates a boolean that changes over time.
- **Predicates:** MEOS implements temporal-aware predicates such as intersects, touches and equals.

- **Geometry Model:** To track the temporal evolution of a moving object, MEOS employs temporal types such as `tgeopoint` and `tgeogpoint`. Furthermore, it implements `tboxes`, i.e., temporal bounding boxes.
- **Compression Functions:** MEOS implements a lossless compression algorithm when creating a trajectory. In addition, it offers single-pass Douglas-Peucker (DP) line simplification [3] and recursive DP.
- **Analysis Functions:** MEOS has the ability to extract trajectory properties such as speed, length, angle and trajectory similarity measures.

3.2 PyMEOS Components

PyMEOS has a three-layer architecture, with every layer hiding implementation details that improve the usability of the library. The layers, from inner-most to outer-most, are:

- (1) **C Foreign Function Interface (CFFI):** This layer is automatically generated by the CFFI library and is a shared library (.so file in Linux) that is imported in Python and can call MEOS' functions directly.
- (2) **CFFI bindings:** This layer wraps all the functions presented in the CFFI layer and manages most of the required transformations of values to interact with the inner layer. For example, it transforms from Python's `string` to C's `char *`.
- (3) **Outer bindings:** This layer combines MEOS' functions into a set of classes in an object-oriented interface that the end user is meant to use.

These layers are divided into two different Python packages:³ `pymeos_cffi`, which contains the two inner layers, and `pymeos`, containing the outer-most one. Although both can be used to interact with MEOS, the latter provides a more user-friendly API.

In addition to all MEOS' functionality, PyMEOS provides extra functionality:

- **Database Connections:** PyMEOS offers direct database connectors enabling read/write operations for PyMEOS' objects from and to a MobilityDB database. Connectors are available for `psycopg` and `asyncpg`.⁴
- **Python Integration and Visualization:** PyMEOS can be installed via `pip` and is compatible with Jupiter Notebook.⁵ A variety of functions are available within PyMEOS to facilitate visualization of temporal types. This is accomplished through the use of `matplotlib`.⁶

4 DEMONSTRATION SCENARIO

This demonstration focuses on the use of the MEOS library for trajectory manipulation. The scenario is presented with an implementation in both MEOS and PyMEOS. This demo presents a scenario consisting of operations managing mobility data. The scenario is an urban setting with Grab-Posisi, a GPS trajectory dataset from Singapore city [5]. The dataset covers over 1 million kilometers from April 2019 and contains over 88 million points in CSV files.⁷ The proposed operations include:

- Input GPS data and generate trajectories (Fig. 2, Query 1).
- Create a spatiotemporal bounding box (`stbox`) around two top-visited places, namely 'Singapore Airport' and



Figure 2: Trajectories generated by MEOS

- 'Gardens by the Bay,' (Query 2) and filter trajectories from '08-04-2019' to '22-04-2019' (Fig. 3, Query 3).
- Segment the trajectories into single-leg journeys (Query 4). Each single-leg journey represents a direct travel segment from one location to another without return. For instance, a trip from 'Singapore Airport' to 'Gardens by the Bay' constitutes a single-leg journey.
- Extract metrics for each trip, including total distance traveled (Query 5), speed (Query 6), average bearing and travel time.
- Identify instances where vehicles came within 10 meter from each other, using vehicle identification and GPS coordinates to determine proximity (Fig. 4, Query 7).

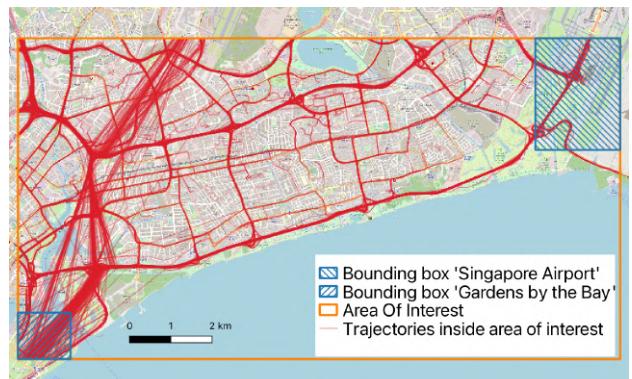


Figure 3: Trajectories and bounding boxes encompassing 'Singapore Airport' and 'Gardens by the Bay'



Figure 4: Trajectories where two vehicles (orange and blue) were within 10 meter from each other

³<https://pypi.org/project/pymeos-cffi/>, <https://pypi.org/project/pymeos/>

⁴<https://pypi.org/project/asyncpg/>, <https://www.psycopg.org/psycopg3/docs/>

⁵https://pypi.io/en/stable/cli/pip_install/, <https://jupyter.org>

⁶<https://matplotlib.org/>

⁷<https://engineering.grab.com/grab-posisi>

4.1 Implementation in MEOS

We start by reading GPS data files containing coordinates and object identifiers (carId). We collect these coordinates and link points based on their carId. Using these joint coordinates, we create a trajectory. We utilize a TSequence, a data structure that encapsulates a sequence of temporal values to structure these trajectories. As the data from the file is processed, points are systematically appended to their respective TSequences.

We filter the GPS trajectories that intersect the areas: ‘Singapore Airport’ and ‘Gardens by the Bay.’ We create an stbox, a spatiotemporal bounding box by joining these two areas’ maximum and minimum coordinates and setting a period of time: ‘08-04-2019’ to ‘22-04-2019’, namely Area of Interest (Fig. 3).

We then segment these collected trajectories into single-leg journeys, such as a direct trip from ‘Singapore Airport’ to ‘Gardens by the Bay.’ First, we identify trips intersecting both areas (Code 1). Considering only the resulting trips, we segment them based on the timestamp indicating when the car was first detected at the origin and last detected at the destination with AtGeometry.

We apply analytical functions to determine key metrics of each car’s journey. To store the results of these metrics, we utilize a tfloat data type. A tfloat is a data structure that represents a float value over time. The total distance traveled is achieved by accumulating the distance between each GPS point in the trip. Additionally, we compute the average speed for each trip. The average bearing of travel is also determined based on the angular data associated with the vehicle’s path. Furthermore, we generated a list of instances where cars were within 10 meters from each other by comparing the position of each car and timestamp. In Fig. 4, we show an example with two cars.

```

1   for (size_t i = 0; i < numberCarsinAreaofInterest; i++){
2       if (eintersects_tpoint_geo(
3           cars[i].trip, places[0].geom)
4           && (eintersects_tpoint_geo(
5               cars[i].trip, places[1].geom))){
6               cars[no_cars].carID = cars[i].carID;
7               cars[number_cars++].trip = cars[i].trip; }}
```

Code 1: Identifying car trips inside the areas of interest

4.2 Implementation in PyMEOS

The procedures outlined above can be equivalently implemented using PyMEOS. In addition, PyMEOS provides visualization functionalities using matplotlib.

First, we visualize the trajectories using a heatmap, which allows us to identify the more transited parts. To do so, we first compute the bounding box containing all the trajectories and tile it into boxes of $500m \times 500m$. Then, we compute, for each tile, how many trajectories intersect it and plot it using Plotly (Fig. 5).⁸

Initially, we employ a general bounding box encompassing both the ‘Singapore Airport’ and ‘Gardens by the Bay’ to exclude any cars that do not operate within these areas. Afterward, we introduce bounding boxes for each location to identify the cars whose trajectories intersect these areas. We can see the result of this process in Fig. 6. Specifically, Fig. 6a depicts trips that do not cross into the area of interest. On the other hand, figure 6b shows trips that traverse the area of interest. In Fig. 6c, the focus

⁸<https://plotly.com/python/>

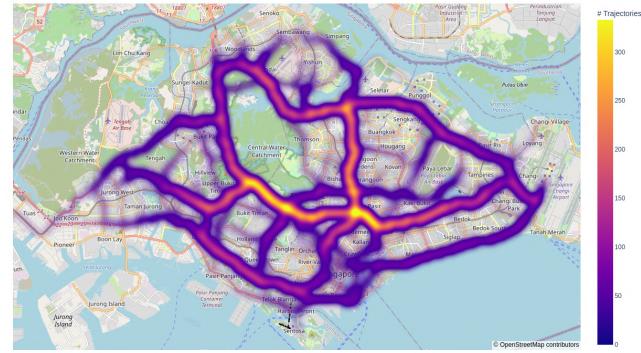


Figure 5: Heatmap of cars’ trips generated in Python/PyMEOS

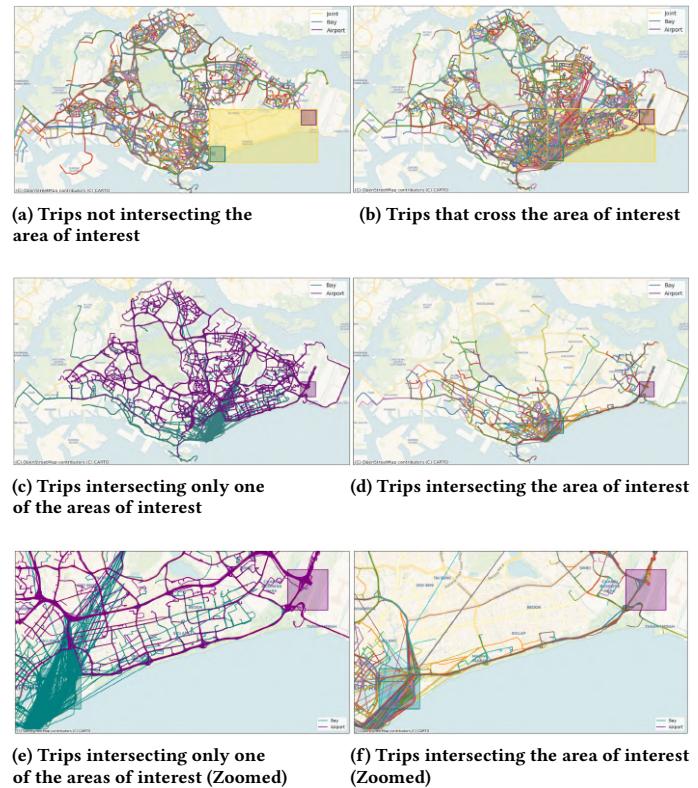


Figure 6: Comparative analysis of trips to the areas of interest

is on trips that intersect only one of the two designated areas, while Fig. 6d presents trips that pass through both specified areas. Fig. 6e and Fig. 6f show a zoomed-in version of the previous two Figures, showing in detail the trajectories that cross the areas.

Next, we see the method used to split the trajectories into single-leg trajectories (Fig. 7, Code 2). For every car, it first computes whether the car is in ‘Singapore Airport’ or in ‘Gardens by the Bay’ at every moment, storing this information as tbool, a temporal boolean (Lines 1–5). Then, we compute the periods in which the car is outside, removing the first and last segments (Lines 6–9) and uses them to clip the original trajectories into



Figure 7: Geometries of single-leg trips

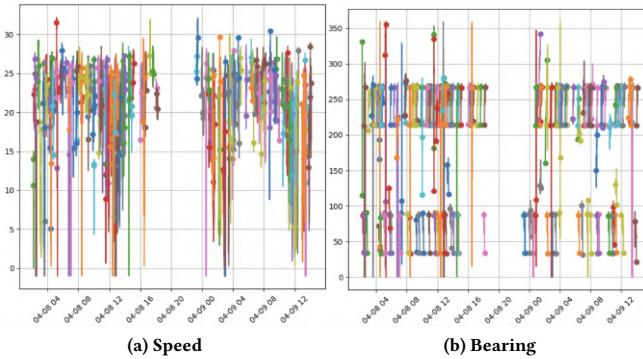


Figure 8: Speed and bearing of single-leg trips

parts that go from one area to the other (Lines 10–11). In addition, we compute the metrics for each single-leg trajectory, including speed (Fig. 8a) and bearing (Fig. 8b).

```

1 st['Is Inside'] = st['Trajectory'].map(lambda t:
2     trip.is_spatially_contained_in(bay)
3     .temporal_or(trip.is_spatially_contained_in(airport)))
4 st['Time Inside'] = st['Is Inside'].map(lambda t:
5     t.when_true())
6 st['Period'] = st['Time Inside'].map(lambda t:
7     t.to_period())
8 st['Time Outside'] = st.apply(lambda row:
9     row['Is Inside'].at(t['Period']).when_false(), axis=1)
10 st['Single Trips'] = st.apply(lambda row:
11     row['Trajectory'].at(['Time Outside']), axis=1)

```

Code 2: Trajectory split using bounding boxes

5 PERFORMANCE

In this Section, we measure the performance of MEOS and PyMEOS. Table 1 relates the number of records, size, and compression rate of the geometry column. When the trajectory is created, a lossless compression is applied. That is, MEOS drops points that can be extrapolated by the points before and after. The number of records is reduced by 99% and in size by 7%. The reduction in size rate can be affected by the amount of significant events, i.e., points that present a change of speed and bearing. Next, we apply a recursive spatiotemporal Douglas-Peucker (RDP) algorithm with thresholds of 10, 50 and 100 meters.

Figure 9 relates the run time for queries specified in Section 4 in MEOS and PyMEOS, on a logarithmic scale. Due to the implemented language, MEOS has a higher performance than PyMEOS. Nevertheless, PyMEOS exhibits strong performance in Python, mainly due to its C backend.

	Records	Size	Percentage
Input	9096633	252MB	-
Trajectories	28000	234 MB	7,14%
RDPS10	28000	67MB	73,41%
RDPS50	28000	28MB	88,89%
RDPS100	28000	20 MB	92,06%

Table 1: Number of records, column size, percentage of reduction for input, after trajectory creation, and application of recursive Douglas-Peucker (RDP) with varying thresholds of 10, 50 and 100 meters

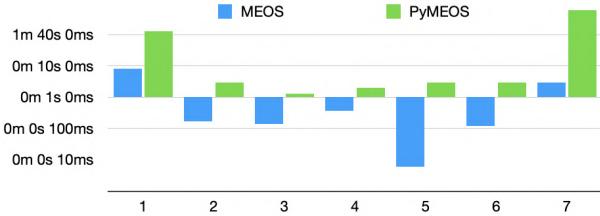


Figure 9: MEOS/PyMEOS performance for queries (Section 4)

6 CONCLUSION

In this demonstration, we introduce MEOS, a new versatile library for manipulating and analyzing mobility data. The library offers data management functions and algorithms in batch and streaming environments, including trajectory creation, interpolation, normalization, and trajectory analysis. Notably, MEOS achieves separation between data storage and processing while maintaining the same functionalities as MobilityDB [1].

We plan to extend the programming language bindings available for MEOS, including Javascript, Java and C#. Furthermore, we will implement in-memory indexing capabilities in MEOS, a necessary enhancement for effectively manipulating large volumes in streaming environments. With these additions, MEOS opens up new possibilities for real-time analysis.

REFERENCES

- [1] M. Bakli, M. Sakr, and E. Zimányi. 2020. Distributed Spatiotemporal Trajectory Query Processing in SQL. In Proc. of the 28th International Conference on Advances in Geographic Information Systems. 87–98.
- [2] B. Custers, M. Kerkhof, W. Meulemans, B. Speckmann, and F. Staals. 2021. Maximum Physically Consistent Trajectories. ACM Trans. Spatial Algorithms Syst. 7, 4, Article 17 (2021), 33 pages.
- [3] D. Douglas and T. Peucker. 1973. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization* 10, 2 (1973), 112–122.
- [4] A. Graser. 2019. MovingPandas: Efficient Structures for Movement Data in Python. *GIForum* 1 (2019), 54–68.
- [5] X. Huang, Y. Yin, S. Lim, G. Wang, B. Hu, J. Varadarajan, S. Zheng, A. Bulusu, and R. Zimmermann. 2019. Grab-Posisi: An Extensive Real-Life GPS Trajectory Dataset in Southeast Asia. In Proc. of the 3rd ACM SIGSPATIAL PredictGIS’19. 1–10.
- [6] L. Pappalardo, F. Simini, G. Barlačchi, and R. Pellungrini. 2022. Scikit-mobility: A Python Library for the Analysis, Generation, and Risk Assessment of Mobility Data. *Journal of Statistical Software* 103, 4 (2022), 1–38.
- [7] E. Zimányi, M. Sakr, and A. Lesuisse. 2020. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *ACM Trans. Database Syst.* 45, 4 (Dec. 2020).

ACKNOWLEDGMENTS

This work was funded by the EU’s Horizon Europe research and innovation program under Grant No. 101070279 MobiSpaces.