25th ITS World Congress, Copenhagen, Denmark, 17-21 September 2018

EU-TP1362

A Fast and Versatile Map Matching Engine

Jean-Sébastien Gonsette[1*]

1. AISIN AW – AWTC Europe SA, Belgium

Tel : +3 223 892 478, e-mail : jean-sebastien.gonsette@awtce.be

19 Avenue de l'Industrie, 1420 Braine l'Alleud, Belgium

Abstract

Map matching is a technology that is brought up to date because of the need to process the potential huge amount of connected car traces on a server, in the context of location-based services. Those positions are often a key resource to improve their relevance and accuracy. In this paper, we present the development of a map matching engine that is fast, flexible, scalable and versatile. It embeds a map matching library with state-of-the-art algorithms based on a Hidden Markov Model. A benchmark to test its performance and its accuracy is presented, along with a comparison with other popular map matching engines.

Keywords:

Map Matching, Road Map, Location

Introduction

Map matching algorithms date from the very beginning of the era of navigation systems. They are so fundamental and so embed in navigation stacks that they tend to be forgotten and not questioned anymore. Nevertheless, the value they bring is key to build reliable and performing localisation services on top. Historically, they have been very important for years on in-vehicle navigation systems, by determining the traversed vehicle road in real time. But today, map matching is as well important in two new domains. The first area is the one of the automated car, for which positioning at the *lane level* is required: finding the correct road is not enough anymore to take adequate control decisions. Then, the second context is related to the *connected car* to build various statistics on servers. Those statistics can indeed be processed to build useful services for the driver, like traffic jam modelling, dangerous road detection, route recommendations, etc. The more the quality of the data extracted from the map matching algorithm is, the more the reliability of the top services can be.

Basically, a map matching procedure aims at determining which road a vehicle is based on a provided localisation point. This point comes in a more or less noisy way, depending on the set of sensors used to build its estimation. GPS is always part of the game, but additional dead reckoning sensors can come into play in order to increase the level of its inherent accuracy. However, the map matching algorithm may not be aware of all the peculiarities that lead to the construction of the position estimation. This is typically the case when one map matches huge collection of traces off-line, while the exact source of the location positions is not certain. Therefore, a robust map matching implementation must be able to cope with various sources of localisation, with very different qualities. For example, those positions could be provided with a periodicity of the tenth of seconds to several minutes, and even vary inside a single trace. Regarding the localisation coordinates themselves, they are in the very best case polluted by a more or less unknown Gaussian noise. In practice, the most annoying GPS noises are due to multiple reflections against buildings in urban environment and they may tend to strongly bias the computed position for a while.

But another strong source of disturbance that may be encountered happens when incoming positions have been already map matched by another system. Even if it sounds strange to do the map matching process again in such circumstance, it is logical for two reasons. The main one comes from the fact that it may not be possible to know that a given trace has already been processed, typically by an embedded navigation system. Secondly, maps used by the two processes are unlikely to be the same or to match with each other. As floating car data is just raw coordinates, they don't embody any notion of link id with them anyway. Then, it is necessary to map match the trace again to get a solution fitting the desired map. But this previous map matching may be a strong source of highly correlated errors. It happens in case of discrepancies between maps, but is particularly relevant when previous map matcher did its job wrong. In this case, position errors, map errors and map matching errors can be badly accumulated without cancelling each other.

Background

Many algorithms have been proposed in the literature to solve the map matching problem. Applied techniques go from geometric or topological analysis to probabilistic algorithms. An extensive review of the different techniques is available in (1). Techniques based on Hidden Markov Model (HMM) (2) (3) have become popular because of their capability to evaluate multiple hypotheses at a time and to discriminate them later when more position samples are available. This model has been proven to be more tolerant against noisy environments. Many modern solutions are based on this technique. For instance, Barefoot (4) is a scalable cloud-based and open source solution whose map matching algorithm is derived from (5). OSRM is an open source project aimed at providing a powerful routing engine. It also embeds a map matching service derived from (5).

Contribution

Many solutions exist to perform map matching, being free to use or coming as paid on-line services. However, they suffer from different drawbacks that led us to discard them. Those inconveniences come in the different flavours of *performance*, *scalability*, *flexibility*, *versatility* and *evolving ability*. Some services are simply not designed to scale up or to handle *high rate*s of traces, like several tens of thousands points by second as we envision. Other solutions may be performing but also comes as a whole bundle of features heavily connected to each other. The main problem is therefore related to the *strong linkage* between the algorithms and the underlying map technology, forbidding to easily change the map provider if we would like to. Another difficulty we encountered with the available technologies is their inability to solve the *same map matching problem* in *multiple environments*: map matching a single trace in real time in the low resource environment of the car on one side, or map matching millions of doubtful quality traces with the full power of servers on the other side. Finally, all those solutions would have prevented us from improving the quality of the underlying algorithms by implementing our latest innovations, like the capacity to *map match a position at the lane level*. This latter capacity is becoming crucial for modern automated cars that need to know where they are and to plan accurate trajectories.

The aim of the work presented in this paper is to provide a solution meeting the criteria of performance, scalability, flexibility and versatility presented above. One key point to achieve all those goals is the factoring of all the algorithm-related parts into a separated map matching library. Such cleavage enables solving the same map matching problem in multiple environments, without any consideration or dependency on the map provider, the threading model, or any other memory, performance or implementation specific aspects. This library comes just as a lightweight yet powerful bunch of algorithms that can be easily integrated into any environment, like the navigation system of a car or on the cloud. This library brings intrinsic performance and flexibility, along as being a strong enabler for further scalability and versatility. It is presented in the first part of this article. Then, the second part of this work explains how this library has been incorporated into an on-line map matching engine capable of sustaining and process huge flows of connected vehicles data.

The map matching library

The map matching library is the angular stone on which different kinds of services can be built straight up from. Its implementation must therefore fulfil different qualities in addition to implement state of the art algorithms:

1. Portable implementation in various environments, including those with low memory capabilities;
2. Complete map agnosticism to avoid relying on any specific map interface or storage mean.
3. Support for various threading models without restrictions, typically to allow multiple map matching processes to run on multiple core while sharing memory;
4. Support of on-line and off-line map matching, either on small incrementally-built local map or on full map;
5. Robustness to any trace sampling periods or localisation accuracy;
6. Capacity to deal with various sets of optional sensors, in addition to the position: odometry, speed, heading direction, etc.

The first three points of this list have a direct impact on the software architecture, while the last three points are more related to the details of the algorithms running inside. Algorithms are described in the next section.

The library has been implemented in C++ for two reasons. First, it is still the language of choice to achieve universal portability, even on small embedded devices. In addition, it also enables reaching the best performance in terms of velocity and memory requirements. Map agnosticism means that the library is passive and needs to be fed with map content by an external provider. This postpones the responsibility of interfacing with a map database to the upper level and it certainly introduces some overhead in passing. The reason comes from the necessity to transfer data to the library instead of just accessing to it. However, this design decision makes it possible to get the best possible flexibility by lowering the dependencies to zero. Loading map data has a negligible cost when working with small local maps around a position. It certainly does not when working at the scale of several countries. However, this cost is mitigated by the possibility to save the library data structures on disk and to directly map them into memory at start up.

The map matching library is passive regarding its threading model. However, it has been designed so that it could support any upper level concurrent architecture. For instance, it enables instantiation of multiple map matching modules, each one accessing its own map data or sharing a single map with other modules. The library supports concurrent map matching requests, along with concurrent writing and reading accesses to the map data. As a result, any processing architecture can be envisaged on the exploiting process side.

Algorithms

The first part of the library consists of dealing with various positioning contents and to provide a consistent and augmented representation of the vehicle position to the map matching algorithm. This step is vital to standardise the format of the vehicle trajectory samples, whatever the source they come from. Concretely, its aim is to fuse all the available sensors information together in order to produce a vehicle state, along with the uncertainty of all its components. This state includes not only the position, but also the speed, the elapsed distance, the heading angle and all the covariance errors. For cloud use cases, the input data is often reduced to the GPS only. Anyway, provided data can also include other sensors like odometry, gyro, speedometer, etc. The heart of this processing is based on a Kalman filter that merges the different information, if they are available, and that can also predict the position of the car in the short term. The idea is to be the *as robust as possible*. If some sensor information is missing, it is deduced from the other data. The only symptom is then that the corresponding uncertainties on the vehicle state are increased.

Regarding the map matching problem, it consists in finding a sequence of map segments that maximises the likelihood to correspond with a sequence of vehicle states provided by the positioning algorithm described above. Assuming that a vehicle drives on such segments, this problem can be modelled with a *first order Hidden Markov Model* (HMM). Indeed, the decision to take one or other branches in the map tree only depends on the current segment the vehicle is on. This model is hidden because the true segment location is never observed directly. Such HMM problem is efficiently solved with a Dynamic Programming algorithm, such the *Viterbi algorithm* whose modern implementation is given in (6). It consists in finding the sequence of segments that maximises the joined probability of the observed coordinates.

An illustration of the Viterbi process is given in Figure 1. Identifying the most likely sequence of segments is done in three steps for each new incoming coordinate. First, potential map matched positions are generated by projecting the coordinate on the close segments. Each projection is given a probability according to some measurement criteria catching the likelihood it corresponds to the raw coordinate. Those are called the *emission* probabilities. In a second phase, those candidates are added to the tree describing all the possible states at each time step. Then, different *transition* probabilities are drawn from the likelihood that the vehicle trajectory actually followed the possible transitions between two consecutive states in this tree. Finally, the probability to be in each possible projected position is given by the combined probability of the most likely sequence of states in the tree.

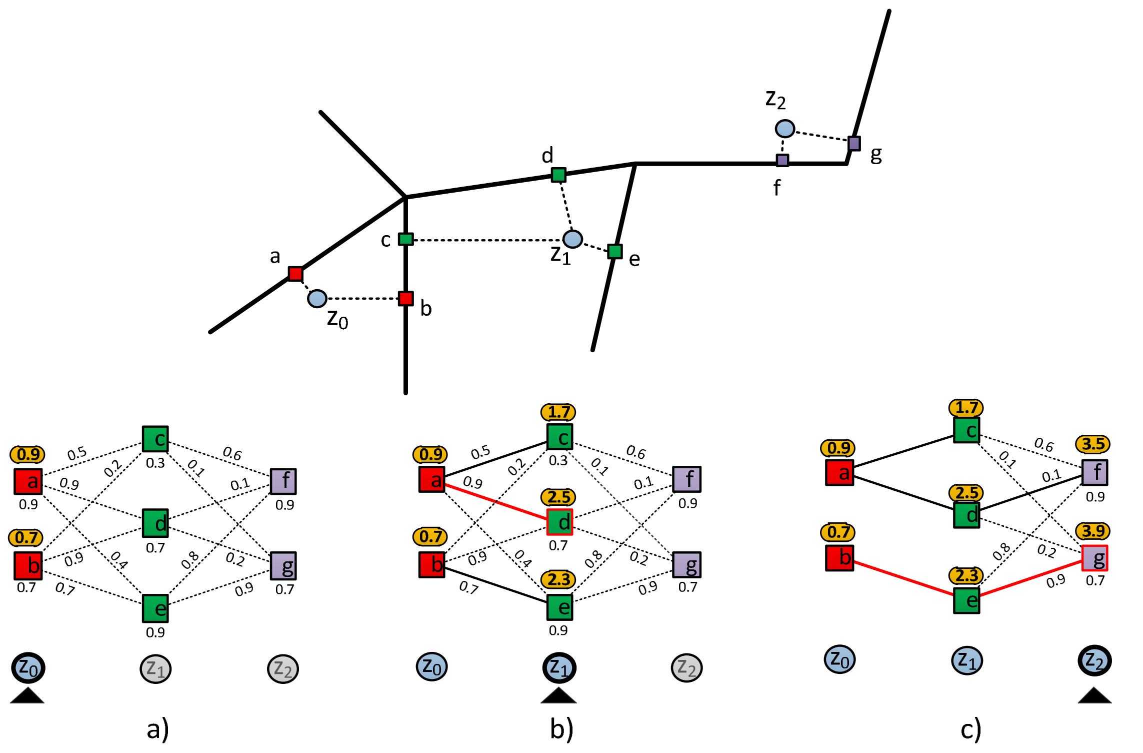

Figure 1: *Illustration of different steps of the Viterbi algorithms based on the road topology shown on the top of the figure. Three successive measurements z0 to z2 produce candidates (a, b), (c, d, e) and (f, g);* ***a)*** *The lattice represents the different candidates over time with their associated score, below. Possible transitions between candidates are shown with a dashed line. The number over this line is the score of the transition. Candidate a is the most probable candidate on the reception of z0;* ***b)*** *Candidates (c, d, e) are updated on the reception of z1. The solid lines represent the most probable paths that lead to each of the candidate (highest score). Paths are evaluated by summing scores of the sources, the destination and the transition. Candidate d becomes the most probable candidate with a score of 2.5;* ***c)*** *Candidates (f, g) are updated on the reception of z2. Candidate g*

becomes the most probable candidate with a total score of 3.9. The full path that reaches it is the sequence b-e-g, in red.

If the Viterbi algorithm forms the backbone of the solution to the map matching problem, it is still important to populate it with good emission and transition probabilities. Emission probabilities measure the likelihood of each map matched candidate against its corresponding raw position coordinate, ignoring the history of previous coordinates. Those probabilities are modular depending on the type of information coming from the vehicle trajectory samples:

- A first probability is drawn from the *distance* between the raw position and the nearby segment. It must take into account an estimate of the position coordinates error, as well as the possible map error on the exact location of the segment. The more the joined error, the more this probability can tolerate a greater distance. It must also consider an estimate of the road width as it can have a significant impact when it is large, especially if the error on the position is low.
- An estimation of the vehicle *heading angle* is used to measure the likelihood to drive a given segment, based on the alignment difference. The more this heading deviates from the segment direction, the less probable this segment is. This probability must first consider the uncertainty on the vehicle heading and on the segment direction. It must also consider an estimate of the vehicle speed. Indeed, the higher this speed, the less those two directions can differ from each other. At very low speed, however, the vehicle heading must not have any impact.
- If available, the *segment speed limit* can be used to derive the likelihood to observe a given vehicle instantaneous speed. It is based on the (debatable) assumption that it is unlikely that the driver will infringe the Highway Code too much. Vehicle speed can be obtained directly from the sample data, or estimated based on the observed successive positions.

Transition probabilities describe the likelihood to connect two candidates at two successive time steps, just by judging the path that connects them together. As candidates can be on different segments not directly connected to each other, the geometrical attributes of each path is retrieved by computing the shortest path (in time or distance) joining each examined pair. Even if there is no guarantee that the vehicle would have followed this path, this remains a reasonable assumption if the sampling time is not unreasonably large.

- Plausibility of a path is first considered by comparing its length with the distance driven by the car, while taking uncertainty of both into account. This distance can be obtained directly from the sampled data if odometry is available, or estimated based on the observed successive positions.
- If the sampling rate is low, it may well happen that a transition concerns positions that are on separate segments. In this case, we can compute the minimum time needed to traverse the graph according to the segments allowed speed. This minimum time can then be compared with the true elapsed time from the data.
- When the path joins two positions that are on both sides of a crossroad, it is possible to consider

the maximum safe speed that allows taking the turn according to its curvature radius. Comparison of this safe speed with the observed vehicle speed may help to disambiguate multiple candidates.

Results

Performance of a map matching implementation can be analysed for the two criteria that are *speed* and *quality* of the prediction. Speed may not be too much relevant for embedded real-time map matching, where a single point is map matched once upon a while. However, it is crucial for the purpose of map matching a high volume of traces on a server. The analysis is performed on the basis of simulated vehicle traces. The reason for this choice is that they are de facto noise free and the associated ground truth is given straight. It is then easy to change the sampling time of those traces or to add noise from above. The logic of the fictive moving vehicle is kept simple, in the sense that when it reaches the end of a link, it randomly chooses the next one to continue its journey. However, the vehicle must obey the Highway Code, along with basic physical laws: acceleration and deceleration are kept in reasonable limits, and lateral acceleration is kept safe inside turn angles.

The test setup is as follows: 30 traces have been generated randomly over a country, each one having duration of 5 minutes. Traces have been further sub sampled with a periodicity of 1, 5 and 30s. An additive Gaussian noise of 1, 5 and 15m std. error is then combined with each trace coordinate. The quality of a map matched trace is defined as the fraction of map matched points falling on any segment of the true trajectory followed by the simulated vehicle. The underlying map used for the tests is OpenStreetMap as it is commonly available for the different platforms of this benchmark. The performance is measured by recording the time needed to map match 1M of trace coordinates, on a single thread of the processor (Intel Core i7-4810MQ @ 2.8GHz). Results of this analysis are given in Table 1 and Table 2 for our library, OSRM and Barefoot.

	Smp. time	**1s**			**5s**			**30s**		
Lib.	**Noise**	**1m**	**5m**	**15m**	**1m**	**5m**	**15m**	**1m**	**5m**	**15m**
AISIN AW		100	99.9	97.7	99.9	99.9	99.6	99.9	99.7	98.8
OSRM		99.8	98.7	93.8	99.8	99.1	96.6	99.9	99.5	96.3
Barefoot		99.5	99.5	99.5	98.2	98	97.9	98.4	96	92.4

Table 1: *Comparison of the map matching accuracy scores, in %, reached by our library, OSRM and Barefoot. The columns correspond to different sampling period, and to different levels of noise added to the coordinate samples.*

Smp time / Libraries	1s	5s	30s
AISIN AW	75.5K	51.0K	15.5K
OSRM	6.7K	5.2K	2.2K
Barefoot	0.2K	0.2K	0.2K

Table 2: *Comparison of the map matching performances, in samples by second, reached by our library, OSRM and Barefoot. The columns correspond to different sampling period. Those tests were conducted on a single thread of an Intel Core i7-4810MQ CPU @ 2.8GHz.*

The map matching engine

The goal of the map matching engine consists in the processing of streams of vehicle traces. Each trace is defined as a succession of position measurements given by a timestamp and a couple of coordinates. Other information like speed or odometry could be added easily but were not available in the data sets we had to deal with. From an interface point of view, the engine delivers one map matched trace for each raw trace given in input. From a design point of view, the engine wraps the map matching library described in the previous section in a scalable way. It is also responsible to deal with various map suppliers in order to feed the library with the map data it needs, and, of course, to connect to the outside world through its dedicated interface. This generic architecture is given in Figure 2.

The system is made up of one or several nodes interfacing with the external world through two dedicated FIFO queues. One of them is dedicated to convey all the incoming job requests, while the other one is used to deliver the responses. Different queue technologies are possible, as long as they implement the *competing consumer pattern*. This latter is needed to ensure a job is not lost if a node crashes while processing it. We have currently worked with RabbitMQ and ServiceBus. Internally, each node disposes of one map that is loaded into memory at start up and shared among all the instances running on this node. The engine is map agnostic as it only needs features commonly available across map providers: geometry, link ids and number of driving lanes. Therefore, it can accommodate different formats and providers, for now we have worked with OpenStreetMap and Here. Each map matching instance lives in its own thread in order to parallelise the load work.

Our map matching engine must process incoming traces quickly and being able to absorb bursts of work requests efficiently. A system is scalable if it guarantees a bounded response time whatever the increase of the load. This load is here given by the number of traces to map match by second, and by the number of samples that are contained in those traces. Vertical scaling is first achieved by increasing the capacity of a single node, in terms of the number of processor threads participating in the process. This scalability has no impact on the system memory as the map data can be shared among all the map matching instances. Horizontal scalability is then achieved by controlling the number of nodes in the process. Load balancing is simply achieved by having each map matching instance competing to grab a work item in the incoming queue.

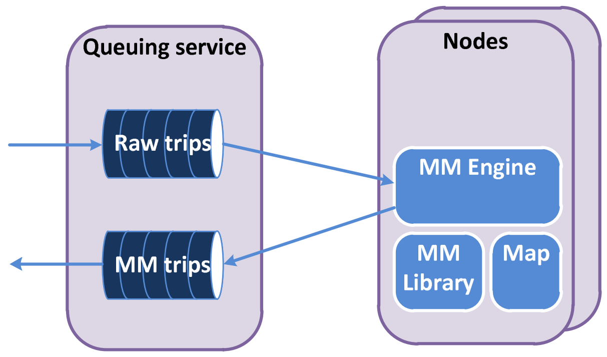

Figure 2: *System architecture of our map matching engine. Multiple nodes compete with each other to grab work items to process. Each work is a trace made up of a variable size of sampling points.*

Results

The complete implementation of the map matching engine has been assessed on a data set of roughly 150 million samples spread over 85,000 trips in Europe. Points were sampled every 45s in average. The test ran on a single Azure standard E8 machine (i7 6820 HQ, 2.7GHz, 8 vcpu, 64 GB memory). Trips were delivered to the engine through a Service Bus message queue in JSON format. Map matched traces were similarly outputted in JSON format through another dedicated queue. All traces were successfully map matched in 30 minutes, achieving overall performances of *80K samples per second*.

Conclusion

Map matching has always been a key component of navigation systems. But the arrival of connected cars and related connected services pushes its usage further. Location-based services are built on the ground of big amount of map matched probe data. They thus require to process efficiently those streams of vehicle coordinates, while dealing with potentially disparate contents or accuracy.

Automated cars relying on HD maps and requiring map matching at the lane level are another reason to push those algorithms further. It requires embracing, not only the positioning information, but also all the sensors that can reduce the uncertainty on this measurement.

The work presented in this paper explained how we could isolate the map matching problem into a separate component to avoid solving this very same problem each time a new context is encountered. This library reaches state-of-the-art performance and accuracy while focusing entirely on the algorithmic part of the problem. This work goes beyond standard map matching services thanks to the possibility to fuse different sensor information to increase positioning accuracy. This capability allows us to now focus on the integration of lane matching algorithms. On the other hand, we showed how we used this library to build a full map matching engine running on a server. This later development could be realised quickly thanks to the flexibility of the underlying library. Despite this, our production ready implementation is scalable and shows outstanding performances by being able to process more than 80K sample points by second and by node.

Next development we would like now to consider is related to the essential emerging field of machine learning. Nowadays, this topic penetrates all areas of the society and could certainly offer further improvement to the subject of map matching.

References

1. *Current map-matching algorithms for transport applications: State-of-the art and future research directions.* **Quddus M., Ochieng W. & Noland R.** s.l.: Transportation Research Part C: Emerging Technologies, 2007, pp. 312–328.

2. *On-line map-matching based on hidden Markov model for real-time traffic sensing applications.* **C.Y. Goh, J. Dauwels, and co.** 15th international IEEE Conference on, s.l.: Intelligent Transportation Systems (ITSC), 2012.

3. *Map-matching for low-sampling-rate gps trajectories.* **Yu Zheng Yin Lou, Chengyang Zhang and co.** s.l.: Proceedings of 18th ACM SIGSPATIAL Conference on Advances in Geographical Information Systems, 2009.

4. *Putting the car on the map: A scalable map matching system for the open source community.* **Mattheis, Sebastian, Khaled Al-Zahid, Kazi and Engelmann, Birgit.** s.l.: INFORMATIK, 2014.

5. *Hidden Markov map matching through noise and sparseness.* **Krumm., P. Newson and J.** s.l.: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2009, pp. 336–343.

6. *On-line Viterbi Algorithm and Its Relationship to Random Walks.* **Sramek, Rastislav, Brejova, Brona and Vinar, Tomas.** s.l.: Algorithms in Bioinformatics: 7th International Workshop (WABI), 2007, Vol. 4645 volume of Lecture Notes in Computer Science.