Compass-Based Navigation in Street Networks

Stefan Funke[1], Robin Schirrmeister[2], Simon Skilevic[2], and Sabine Storandt[2]

[1] FMI, University of Stuttgart (Germany)
funke@fmi.uni-stuttgart.de
[2] Department of Computer Science, University of Freiburg (Germany)
storandt@informatik.uni-freiburg.de

Abstract. We present a new method for navigating in a street network using solely data acquired by a (smartphone integrated electronic) compass for self-localization. To make compass-based navigation in street networks practical, it is crucial to deal with all kinds of imprecision and different driving behaviors. We therefore develop a trajectory representation based on so-called inflection points which turns out to be very robust against measurement variability. To enable real-time localization with compass data, we construct a custom-tailored data structure inspired by algorithms for efficient pattern search in large texts. Our experiments reveal that on average already very short sequences of inflection points are unique in a large street network, proving that this representation allows for accurate localization.

1 Introduction

Mobile devices are one of the primary tools for navigation nowadays. They more and more replace integrated navigation systems in cars, and can also be used when going by bicycle or foot. Typically, mobile devices rely on GPS, GSM or Wifi for localization. GPS allows for a rather precise determination of the actual position (up to few meters) if the GPS receiver gets signals from at least four satellites. Unfortunately, this might not always be possible due to signal blockage (e.g. by high buildings or foliage), furthermore signal reflections might induce imprecisions. To use GSM, one has to be connected to a cell phone network, with the signal strength of nearby base stations revealing the position (with a precision of $\approx$50m). Wifi localization works in a similar fashion. Here, companies like Google georeference wireless access point IDs, and as soon as signals are received, a certain geographic location can be estimated (with a precision of $\approx$20m). So for GSM and Wifi, interaction with a third party is required to self-localize. And even for GPS, to obtain a faster position lock, third party servers are contacted via GSM or 3G. This raises privacy issues, as the own position is revealed to these third parties. There are recent attempts, though, to crowd-source GSM and Wifi access points and make the data openly available[3]. But the coverage of mapped data is still poor. Moreover, GSM and Wifi is simply not

[3] https://location.services.mozilla.com/

available everywhere; especially in rural or sparsely inhabited areas one cannot expect precise localization based solely on those signals.

We propose a new way of navigating in street networks, by making use of the electronic compass present in most of the current smartphones and other mobile devices. We acquire sequences of absolute directions and use them to identify the trajectory the vehicle has taken in the network. So for a given road map and the measured absolute directions, we aim at identifying the path in the map that most likely is the one that led to those measurements. The problem of pinpointing measurements to a path in a map is commonly referred to as *map matching*. The advantage of our scheme is the ability to self-localize in a completely autonomous way. No interaction with third parties is required. Moreover we do not rely on any kind of distance measurements, which are typically imprecise if conducted with a mobile device.

We will describe in detail how to obtain, process and store compass data, and how to instrument this for precise and fast self-localization.

1.1 Related Work

In the classical *map matching* problem, we are given a sequence of possibly imprecise location measurements (obtained e.g. with GPS, GSM or WLAN). This setting is well-studied in different variations. The on-line version (measurements have to be processed the moment they are taken) is described e.g. in [1]. In the off-line case the best possible path in the map for a given measurement sequence is chosen as the optimal one according to some scoring function. The score might for example be the Frechet-Distance, see [2], or the objective function value of an integer program [3]. In [4], the authors have shown that even very imprecise GSM localization allows for a very accurate reconstruction of the route a mobile user has traveled along in a network if measurements for a long enough period can be gathered.

Alternative sources of information for localization (besides GPS, GSM or WLAN) have been investigated before. In [5] the authors introduce so called *path shapes* which describe the sequence of *relative* movements of a vehicle (e.g. '500m straight, 40 degree left turn, 200m straight, 90 degree right turn, $\cdots$'). Experiments show, that different paths quickly exhibit differing path shapes, which allows for high-precision self-localization. The relative movement data is acquired by reading information from the on-board computer of a car.

Smartphones do not have access to this data, as typically there is no open interface for communication with the on-board computer. Of course, most smartphones also have integrated gyro sensors and accelerometers, which allow to measure turning angles and (increase in) velocity. In theory, this yields the same kind of data as needed for the path shape localization scheme. In practice, though, due to the imprecision of this data, such methods only make sense to *complement*, not replace GPS localization. The latter is the goal of this work.

In [6] the concept of *elastic pathing* is introduced. The authors show that fine-grained speed information is also sufficient for self-localization. Every path in the street network exhibits a typical 'speed profile' to which actual measurements can be compared. In contrast to *path shapes*, this localization scheme requires knowledge about the starting location. We aim to be able to compute the actual position without the start position being known beforehand. Moreover we have the same problem here as with path shapes: While the car itself monitors the driving speed autonomously, speeds measured by a smartphone normally involve GPS usage. And even more severe, a huge amount of historical data is necessary to have good typical speed profiles at hand. This data is not easily available with sufficient coverage (especially for bicycles).

In [7], positions of pedestrians are determined using gyro sensors and a heuristic which mitigates direction errors by incorporating the underlying street network data. They do not use smartphones, though, but specially constructed devices attached to a shoe. Again, they require the user to provide the starting location. No large-scale study is conducted and no timings are given; hence, it is unclear whether this methods can be used for real-time localization.

In other navigation domains where vehicles do not have to follow streets but move around almost freely e.g. considering ships, planes, missiles or robots, navigation based on the movement alone is known as inertial navigation system (INS). Here, given the start point, the current position is calculated based on speed and direction of the moving object as well as the elapsed time since departure. Unfortunately, when using INS already small measurement errors translate into large positional errors accumulating over time, as the new position is always computed relative to the last one. Hence in regular intervals the actual position has to be corrected using e.g. GPS; therefore the autonomy of the system is compromised. An incarnation of INS for pedestrians was described in [8]. They use the built-in electronic compass of modern smartphones and employ an approach based on Bayesian networks, which combines GPS and compass information in a neat manner. As indicated before, they do not consider an underlying path or street network, but investigate free spaces and buildings where people can wander around. In contrast to this, a car or bicycle has to follow streets or paths. Therefore the effect of measurement errors is mitigated in our scenario. Even better, we gain information while driving around, hence the positioning becomes more and more accurate over time – quite the opposite of the INS paradigm.

1.2 Contribution

This paper presents a novel localization scheme purely based on absolute directions acquired by an electronic compass. We describe how to retrieve such data using a conventional smartphone, and how to deal with numerous sources of imprecision. We propose a new compass-based representation for trajectories, which is far more robust in particular against variations of driving speed than temporal subsampling of the absolute directions (e.g. by measuring every second). We show experimentally that already short paths in a network are characterizable by our compass representation, i.e. their representation is unique among all possible paths. As our framework does not rely on a known starting position, naively, we have to consider every node in the network as a potential starting point and then compute the one that most likely led to the observed measurements. An implementation of this naive approach scales very badly with the network size. To enable real-time self-localization, we therefore develop a custom-tailored data structure inspired by algorithms for efficient pattern search in large texts. This data structure allows for self-localization within fractions of a second even in large road networks. Finally, we provide an experimental study, including results on real-world data (collected by bicycle).

2 Wireless Acquisition of Compass Data

The first step in the pipeline is to acquire precise absolute direction information while driving around. In the following, we provide the details for collecting such data with the help of an electronic compass.

2.1 Electronic Compass

We implemented an Android app to gather electronic compass data. Five different methods based on different kinds of virtual and physical sensors provided by the android API[4] were employed:

Orientation Sensor. This used to be the standard way of acquiring compass data, but is officially deprecated now. The orientation sensor is a virtual sensor which directly returns the actual orientation. No parameters are required.

Magnetometer and Accelerometer to Rotation. This is one current standard way of getting absolute direction information. It returns the rotation matrix resulting from reading out the sensor values. No parameters are required.

Low Pass. This method is also based on the magnetometer and the accelerometer but additionally includes a low-pass filter to take care of short-term fluctuations. The higher the input parameter α, the less short-term fluctuations influence the resulting orientation.

Rotation Sensor. This sensor is similar to the orientation sensor but returns a rotation matrix and an estimation for the precision of the measured values.

Attitude Heading Reference System (AHRS)[5]. Apart from the accelerometer and the magnetometer, this method also uses the gyro sensor to estimate the orientation. There are several tuning parameters.

We stored direction values once per second for our real-world experiments. The data collected during an hour of measuring directions is below 300kB, so there is no problem with storing the measurements locally on the phone.

[4] `http://developer.android.com/guide/topics/sensors/sensors_overview.html`
[5] `http://www.x-io.co.uk/open-source-imu-and-ahrs-algorithms/`

2.2 Smoothing

Naturally, no sensor is flawless and the measured angles are perturbed by all kind of external factors. To take care of these fluctuations, we apply a smoothing technique. For that purpose we convert the measurements into a polyline. We do this by starting at $(0, 0)$ in a two-dimensional coordinate system, and then elongate the line by a straight segment in the direction of the measured angle. We always use the same length for each straight segment, i.e. we assume constant travel speed. Then we apply the Douglas-Peucker algorithm [9] to this polyline. Douglas-Peucker reduces the number of points on a polyline but faithfully preserves the overall shape at the same time, therefore we regard this algorithm as very useful in our scenario.

3 Compass Paths

Once the compass data is acquired, the challenge is to match these measurements to a path in the underlying street network. This means paths in the network and gathered measurements have to be made comparable by a common representation. A natural way of encoding a trajectory is just the sequence of absolute directions measured e.g. every second while driving. It turns out that this approach is too error-prone to be practical, though. In the following, we first discuss in detail why this is not the envisioned representation. Then we introduce a new representation, based on so called inflection points which is much more suitable for matching a compass-based trajectory to a path in the network.

3.1 Representation as Sequence of Absolute Directions

The problem of using the sequence of measurements received from the electronic compass directly for map matching, is that the descriptions of turns in the network and in a real trajectory might differ considerably. In a given network representation, a turn happens typically at a single point. So a turn is a sequence of two absolute directions, the one in which the vehicle headed before, and the one in which the vehicle headed after. For example, *30°,60°* describes a *30°* turn. But measuring the direction every second or even more fine-grained, we have to expect that the description of the trajectory for the very same turn looks more like this: *30°,37°,51°,60°*. In fact, every angle between the entrance and the exit angle might appear in real measurements. And depending on the sampling rate, even driving the same turn in the exact same way might lead to a new representation each time. From the pure sequence of absolute direction measurements, it is very difficult to tell which angles are artefacts of turns. Therefore we consider this kind of representation unsuitable for map matching.

3.2 Inflection Point Representation

We aim for a compass-based path representation which is robust against different driving styles and the problem of modelling (sharp) turns as described above. So let $\phi : [0, 1] \rightarrow$ be the function mapping points along the path (parametrized over $[0, 1]$) to an absolute direction. Driving the same path twice at different speeds, we get two different functions ϕ_1, ϕ_2 with different parametrizations of the path (here ϕ_2 could also be interpreted as the path in the underlying network). The question is in what respect could they be considered equal?

A typical path includes both curves or turns to the left as well as to the right. So a possible characterization is to consider the sequence of angles where there is a change from increasing angles to decreasing angles or vice versa. This means in particular, that the sequence of angles resulting from a right/left turn is completely ignored. In fact, this is nice, as there is no way to predict how the subsampling of a turn will look like – and the chances it is the same as in the underlying network is miniscule. On the contrary, points that indicate a change of turn direction tend to be in the middle of almost straight segments or at least at sections where the directional change is not as pronounced as in sharp curves, see Figure 1 for an illustration. Hence restricting the measurements to such points

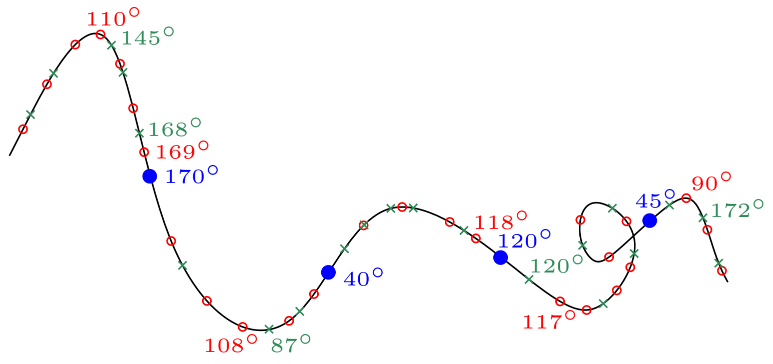

Fig. 1. Continuous trajectory with induced inflection points (blue). Two discrete measurement sequences conducted on this trajectory are indicated by the red circles and the green crosses, respectively. The labelled points near inflection points show that absolute directions are rather stable, while in curves even small sampling differences lead to drastic direction variations.

seems to be a more robust and clean way to compare ϕ_1, ϕ_2. In differential calculus these points of turn direction change are called *inflection points*. They are characterized by the second derivative changing sign. If we map directions to points in time, the inflection points turn out to be local maxima/minima. In our case, the function ϕ need not be differentiable, but we will still call the resulting representation the *inflection point representation (IPR)* or *compass path*. Obviously, the IPR of a path in the street network and of (smoothed) real-world measurements can be computed in linear time, by sweeping over the induced angles and extracting local extrema.

4 Map Matching

Having a common representation for paths in the network and compass paths, the next step is to find for a given compass path the network path that fits best, i.e. solving the map matching problem for our specific input. The notion of similarity for compass paths is yet to be defined. Ideally, we would like to declare two compass paths equal if and only if the sequence of inflection points is exactly the same. But obviously this will yield no match in the map most of the time. An electronic compass is hardly free from error; and even if it were, people do not drive exactly in the middle of the lane, heading in exactly the direction of the respective road segment. So we have to introduce some degree of fuzziness here.

4.1 Curve Matching

In computational geometry, the same problem arises when (polygonal) curves have to be matched. Transferred to our scenario, we are given a collection of polygonal curves (represented by the underlying graph) and want to select the one, that matches our reference curve (the given trajectory) best. One classical measure here is the Frechet-Distance which was already applied to planar maps in [2]. The Direction-Based Frechet-Distance [10] also allows for partial matchings which is beneficial in our envisioned scenario, as our trajectory naturally is only a small part of the whole graph. But these methods require integral calculus and have a runtime of $\Omega(ab)$ with a and b being the number of vertices on the two curves. With b denoting the number of all vertices in the network in our application, this is far from being practicable in a reasonable amount of time. Moreover curve similarity measures like the Frechet-Distance do not necessarily capture the similarity one aims for when considering trajectories. Especially if typical angle fluctuations are known for compass paths (due to experimental studies), there is no easy way to incorporate such knowledge in the measure.

4.2 Tolerance Ranges and Shape-Preserving Search

As argued before, the sequence of inflection points is never going to be *exactly* the sequence of inflection points on the respective path in the underlying map. So we have to allow inflection points to differ by at least some degrees. We realize that by introducing a tolerance value t. Hence, two compass paths $P = p_1, p_2, \cdots, p_a \quad P' = p'_1, p'_2, \cdots, p'_a$ are equal if $|p_i - p'_i| \leq t$ for $i = 1, \cdots, a$. The parameter t captures the imprecision of the electronic compass as well as differing absolute directions induced by individual driving behaviour.

Now the question is how to a extract matching compass paths from the network. Lets assume for the moment that the start location $s \in V$ is known. In [5], a shape-preserving Dijkstra algorithm (SPD) was introduced. Here, a Dijkstra run is started in s, but paths are only explored if their encoding is declared equal (according to our introduced comparison oracle) to the encoding of the reference path (aka the trajectory we want to match). So a node v is only looked at, if the respective path from s to v in the actual Dijkstra search tree yields a prefix of the reference encoding (including tolerance ranges). For an SPD to be as efficient as possible, a newly explored edge should be encodable in constant time. Therefore, in our scenario, we do not only store predecessor labels with every node, but also sign labels, that tell whether we are actually in a right bend, a left bend or in no bend at all. This information along with the difference of the absolute directions of the last two edges on the path is sufficient to decide whether a new inflection point arises.

With the help of this modified SPD, we can search for the longest match of the trajectory in the map that starts at s. As the number of paths that are compatible with the reference trajectory should be very small for reasonable values of t, a single SPD computation is typically very quick. But it is the starting point s itself that is unknown and that we want to discover. Hence, theoretically, we have to start an SPD in every single vertex of the network, as each of them might be the start location we are looking for. Obviously this scales very badly with the network size. Our experiments will reveal that in networks with millions of nodes and edges query times are in the order of minutes. This is absolutely impractical for navigation purposes. Therefore, in the next section we will describe a data structure which allows to speed up queries significantly.

5 A Data Structure for Fast Inflection Point Recognition

Checking for every node in the network if it is a valid starting point of the trajectory in question is far too time-consuming. Of course, once we have identified the correct start location, we can invoke SPD computations for updating the location of the vehicle if it moves on. But to get the initial correct match, we need an alternative approach.

To accomplish fast localization queries, we follow the idea presented in [5] to transfer the map matching problem to a pattern search problem in texts. So the encoding of the trajectory is regarded as a concatenated string. The text describing the network consists naively of all encodings of possible (shortest) paths in the map. Several preprocessing methods for large text corpora exist, which allow to find a pattern in time linear in the pattern length (so completely independent of the length of the text). One way to achieve this, is the creation of a generalized suffix tree (GST) on the text [11]. A GST requires only linear space in the length of the text, if the alphabet has bounded size. This is of course the case in our application, as our 'letters' correspond to absolute directions with a precision of one degree. Therefore our alphabet has 360 letters only.

In the following, we first describe the way a conventional GST is constructed and how queries are answered using this data structure. Then, we briefly review how the GST construction on *path shapes* works (as proposed in [5]). This GST construction scheme is based on some basic assumptions about the encoding, that are not fulfilled with our inflection point representation of trajectories. Therefore we subsequently describe how to adapt the GST creation to be applicable to compass paths as well.

5.1 Conventional Generalized Suffix Trees

For a set of strings $S_1, \cdots S_k$ a GST represents all suffixes of those strings, fulfilling the following characteristics: (1) Each tree edge is labelled with a non-empty string. (2) There is no inner node with degree 1. (3) Any suffix of a string corresponds to a unique path in the tree (starting from the root) with the concatenated edge labels along that path starting with this suffix.

A suffix is grafted into the GST by first identifying its longest prefix that already exists in the tree by tree traversal. If the path of this prefix ends in a node, a new edge and a new leaf are created, representing the last part of the suffix (if there are remaining characters). Also, the path could end implicitly, that means the edge label contains additional characters that are not in the suffix. Hence this edge has to be split together with its label, creating a new inner node that represents the longest prefix. Then one can proceed as described before. By performing this for every suffix occurring in $S_1, \cdots, S_k$ a GST of this set of strings is obtained. Note, that there are even more efficient ways of constructing GSTs, see e.g. [11]. But for the specific construction of the GST for the map matching application these are not applicable, as we will discuss later.

After the GST is constructed, the question whether a given pattern is contained in $S_1, \ldots, S_k$ can be answered in time linear in the size of this pattern, if the alphabet size is bounded, since a bounded alphabet size allows to associate an array with every node, which for each letter of the alphabet stores the edge (if any) whose label starts with this letter. So a query starts in the root of the GST, looking for an outgoing edge with a label, that begins with the first letter of the pattern. If such an edge exists, we have to compare the edge label to the respective pattern prefix element by element. If they are equal, we can go to the end point of the edge and repeat the search with the remaining elements of the pattern. If we always find a match, the pattern is contained in the underlying set of strings, otherwise it is not.

5.2 GSTs for Path Shapes

To construct the GST conventionally, all strings, i.e. all path shapes of all (shortest) paths in the network have to be explicitly available. But for larger networks this is clearly impractical. In fact, there are $\mathcal{O}(n^2)$ shortest paths in a network on n vertices. Even if their encoding length is rather small, the space consumption is far too high to store them all explicitly. Furthermore it is waste of time and space, to store all possible (long) paths. In fact, it suffices to store the prefixes of all paths until they are *unique* in the network. Because at this point, if a trajectory matches this prefix, the starting point can already be identified correctly, and the current location of the vehicle can be deduced. To determine efficiently whether a path encoding is unique in the network is non-trivial. Therefore the GST scheme as presented in [5] interleaves this classification with the GST creation. So the strings that are contained in the final GST are not available a priori, but are constructed in an online manner.

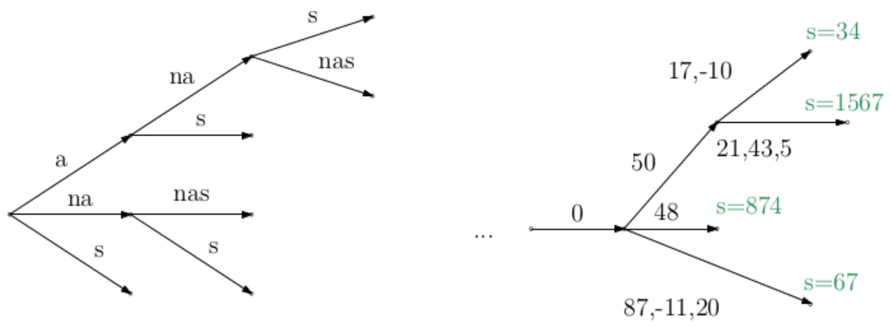

Fig. 2. Left: Conventional GST, here on the single string 'ananas'. Right: Small cutout of a GST on path shapes. The letters on the edges are now angles. Every leaf is labeled with the source vertex of the encoded path.

The detailed construction works as follows. A Dijkstra run is started in every vertex of the network, but with a given maximal distance. After each Dijkstra explored all nodes up to that distance (or the respective priority queue ran empty) it is frozen and all contained paths (implicitly given by the predecessor labels) are extracted and encoded. These encodings are grafted into the GST, with each node in the GST knowing the source and target vertex of the network, whose shortest path led to the creation of the node. Moreover every node has a boolean tag, that is initially set to true. If a path with different target vertex results the very same node in the GST, the tag is turned to false (this can be decided as a by-product of the grafting). Having done this for all vertices, the nodes in the GST with a true tag represent unique paths in the network. Hence the respective targets can be removed from the Dijkstra search tree of the respective source, pruning the search space. As long as not all priority queues of the Dijkstra have run empty, the process is repeated with an increased maximal distance. At the end, all necessary path prefixes are encoded in the GST. In Figure 2, examples of a conventional GST and our specialized GST-based data structure are provided.

When dealing with tolerance ranges, the construction has to be adapted slightly. In fact, it has to be checked whether a path prefix is unique with respect to that tolerance. If not, the respective search spaces cannot be pruned.

5.3 GSTs for Compass Paths in IPR

There are basically two requirements for the path shape GST construction scheme to work:

1. If an encoding of length l is unique among all other occurring encodings with length $\leq l$, a longer occurring encoding cannot destroy the uniqueness.
2. The length of a path encoding is equivalent to the path length.

The first condition is naturally fulfilled in our scenario, but using inflection points the code length does not have to be proportional to the path length at all. So the second condition is violated. Therefore we have to modify the suffix tree construction. To ensure that every path with an encoding length of a given value l is known, we proceed as follows: For every vertex we run Dijkstra with an upper bound on the number of polls (i.e. extractions of elements from the priority queue). After this number of polls is processed (or the priority queue ran empty), we extract the set UL of unsettled leaves in the Dijkstra search tree. For each node in UL we compute the respective path via predecessor-labels and encode it using IPR. If all encodings have a length equal to or exceeding l we are done. Otherwise we increase the poll limit and continue the Dijkstra run. After we ensured that every leave in the search tree is either settled or leads to a long enough encoding, we backtrack all paths from leaves nodes (i.e. all longest paths) and encode them. If an encoding exceeds l, we just use its l-long prefix. All resulting encodings can then be grafted into the suffix tree.

Note, that using inflection point encoding some of the Dijkstra runs might explore a very large search space in order to achieve the required encoding length for every path, e.g. in the case of long straight highways the code length will remain zero. As this increases runtime and space consumption significantly, it should be prohibited as possible. If the path $p = s, u, v$ of a settled node v from the related source s consists of two edges with the same absolute direction, then every elongation of this path will have the same encoding as the suffix of the path starting at node u. Hence there is no need to explore these paths starting in s and therefore in such a case we remove v from the search tree. We can proceed analogously, if we have a path $p = s, u, v, w$ with w being a settled node and the directions of the three edges $(s, u), (u, v), (v, w)$ increase or decrease strictly monotonous. Here again all path elongations will have the same encoding as the path's suffix starting at u and therefore the search space can be pruned.

5.4 Answering Queries

The complete pipeline for answering a map matching query on compass-based data looks like this:

- gather absolute directions via an electronic compass, e.g. every second
- smooth the data to get rid of artefacts
- extract the inflection point representation
- search for the resulting encoding in the GST in order to determine the source vertex in the street network
- if such a source vertex was found, run a SPD (with the trajectory as reference) from this node to determine the end point of the trajectory and hence the current location of the vehicle

Note, that we could also construct the GST backwards, i.e. on reversed paths. Then searching for the reversed inflection point representation of the trajectory, the GST would provide us with the actual position of the vehicle right away – without the necessity to run a SPD. But on the one hand, a single SPD is very efficient, and on the other hand, it allows to display the whole trajectory driven so far on the map. Hence we stick to the forwards approach anyway.

6 Experimental Results

To show the practicability of our approach, we implemented the described algorithms and methods and tested them on several input networks. Our implementation is written in C++, timings were taken on a single core of an AMD Opteron 6172 with 2.1GHz and 96 GB RAM. Table 1 shows on overview of the sizes of our networks (ST -Stuttgart, MA - Massachusetts, BW - Baden-Wuerttemberg, SG - Southern Germany, all extracted from OSM[6]), along with several characteristics. We included Massachusetts as it contains many grid-like substructures (especially in the area of Boston) which we consider challenging for our approach. We observe that the ratio of inflection points to all points on the average shortest path is rather high in larger graphs, in fact about 50%.

Table 1. Characteristics of the used test graphs. Averaged values are calculated on the basis of 1000 randomly chosen shortest paths (SPs). The number of inflection points (IPs) on a path equals its encoding length according to our model.

	ST	MA	BW	SG
# nodes	122,334	294,345	999,591	5,588,146
# edges	243,593	731,874	2,131,490	11,711,088
avg. path length	15.9km	120.4km	78.2km	173.7km
avg. # IPs on SP	119	209	664	1373
avg. % IPs on SP	30.0	51.2	45.5	48.6

6.1 Characterizability of Street Networks

The first crucial step is to show that the inflection point representation really suffices for accurate localization. For that purpose, we conducted the following experiment: We randomly choose a vertex pair $s, t \in V$ and compute the shortest path π between them. Then we use our SPD for every possible start vertex to find the longest match of the inflection point encoding of π. Naturally, the longest match will be π itself or even a superpath, as depicted in Figure 3. To find out, if there is a path really different from π but sharing its encoding, we restricted the result to matches with at least 80% of the edges being different from the reference path. In Table 2 the average prefix lengths can be found, grouped by the number of inflection points we demand the match to contain at least. If we set this number to zero, the longest match simply is the longest shortest path in the network with no encoding, completely independent of the reference path. Increasing the number of inflection points that have to be equal, the prefix sizes decrease dramatically. Already using 5 IPs, the IPR becomes unique quite early, even when using an angle tolerance that allows the IPs to differ by 5 degrees ($t = 5$). The first 10 IPs are rarely matched by any other path in the map even

[6] `openstreetmap.org`

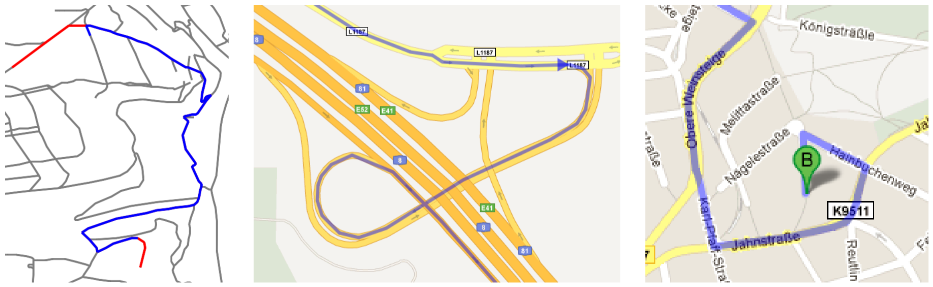

Fig. 3. Left image: Reference path (blue) and its longest match in the map. Without restrictions, the match (red+blue) is naturally a superset of the reference path (blue), as there are prefixes/suffixes with zero encoding length. The two images on the right show long pure right or left turns which do not exhibit encodings in IPR.

Table 2. Unique prefix length (in meters) in dependency of the number of inflection points (IPs) a match has to contain for exact queries (left table), and with an angle tolerance of $t = 5$ (right table). The respective query times (in seconds) denote the time that was necessary to finish a SPD computation for every vertex in the network. Values are averaged over 1000 random queries.

	ST	MA	BW	SG
exact	avg. prefix length (m)			
0 IP	4,822	28,430	4,289	91,941
1 IP	2,509	21,388	3,060	45,083
2 IP	582	9,085	1,269	31,757
5 IP	13	114	66	1,596
10 IP	0	2	1	4
time	7.86	17.18	36.73	245.63

	ST	MA	BW	SG
t= 5	avg. prefix length (m)			
0 IP	5,255	31,931	4,731	94,146
1 IP	4141	31,293	4,441	81,934
2 IP	3,047	23,445	3,778	62,309
5 IP	835	8,853	1,210	4,998
10 IP	5	175	14	3
time	8.03	19.01	42.81	287.12

for Southern Germany. There, 10 IPs correspond to less than one percent of the total number of IPs on an average path.

Hence IPR encoding for shortest paths in street networks seems to be a feasible way to solve the map matching problem accurately. But the running time of the naive approach is a drawback, increasing significantly with the network size – resulting from a SPD run started in *every* vertex. It is almost unaffected by the required number of IPs, as the paths with zero encoding have to be explored anyway, leading to a total runtime of over 4 minutes for a single query in Southern Germany.

6.2 GST Construction

We computed GSTs for all our test graphs, for an exact as well as an imprecision-tolerant comparison model. Table 3 contains the characteristics of the created GSTs. The depth of the GST – reflected by the maximal code length we had to consider – is quite small for all test graphs. For every path in Southern Germany that contains at least 17 inflection points, we can be sure to find a proper source node with the help of the GST. Moreover for exact queries this means that at most 17 comparisons are necessary to retrieve this source node. Having a look at the number of explored nodes per Dijkstra run, the search spaces are very small on average. Nevertheless some of the Dijkstra search trees contain very long path sections with zero encoding, leading to very long maximal distances in that tree, e.g. over 37 km for Massachusetts and over 155 km for Southern Germany. But even with the majority of the Dijkstra runs being very fast, the run time of the preprocessing scales badly with the network size and the introduction of an angles tolerance t. While the depth of the GST only doubles for $t = 5$, the runtime increases significantly. This is due on the one hand to the larger search spaces for the Dijkstra computations and on the other hand to the increased time for checking whether a node in the temporary GST is unique. On a single core we needed about one hour to preprocess BW with exact comparison and about a day with $t = 5$. It took already two days to preprocess Southern Germany without considering tolerances. Future work will include the parallelization of the Dijkstra computations to speed up the preprocessing and permit to use even larger graphs and higher tolerances. But the preprocessing step only has to be performed once – for the queries the resulting GST suffices. For Southern Germany the respective data structure is less than 4 GB in size and hence could be stored on a SD-card of a mobile device.

Table 3. Experimental results of the online GST creation.

	exact				t=5			
	ST	MA	BW	SG	ST	MA	BW	SG
max code size	12	15	13	17	30	37	31	33
avg. #expl. nodes	191	279	202	154	428	498	359	215
max dist (m)	8,697	37,650	9,923	155,481	12,746	37,824	17,111	162,582
time (min)	5.75	38.02	52.83	2978.22	213.71	663.75	1733.35	5287.52
GST nodes	$5.2 \cdot 10^5$	$3.1 \cdot 10^6$	$5.1 \cdot 10^6$	$3.6 \cdot 10^7$	$2.5 \cdot 10^6$	$1.6 \cdot 10^7$	$2.3 \cdot 10^7$	$1.9 \cdot 10^8$

6.3 Queries

Having the GST at hand, we can now answer queries with a tree traversal followed by a single run of a shape-preserving Dijkstra. This is a dramatic improvement compared to n necessary SPD runs using the naive approach. The effect on the practical runtime is shown in Table 4. Using the combination of the GST and one SPD, all queries could be answered in less than half a decisecond. This results from the fact, that all GSTs have a very small depth, hence very few comparisons are needed to find a certain pattern and moreover the SPD run explores almost only the edges, that are part of the resulting trajectory. All in all our approach can answer map matching queries up to 8000 times faster than the naive approach and even for larger graphs this procedure might allow for real-time query answering.

Table 4. Query times (in seconds) for answering map matching queries using different approaches and comparison models. Timings are averaged over 1000 random queries.

		ST	MA	BW	SG
exact	naive	7.8665	17.1836	36.7329	245.6335
	GST+SPD	0.0011	0.0022	0.0045	0.0332
	speed-up	7151	7810	8126	7398
t=5	naive	8.0324	19.0172	42.8134	261.1443
	GST+SPD	0.0015	0.0038	0.0076	0.0458
	speed-up	5355	5004	5633	5701

6.4 Accuracy

The quality of a path p' resulting from a map matching procedure is conventionally measured by the percentage of edges of the correct path p, that are not matched by p' (called A_N), and the percentage of the length of p, that could not be covered by p' (called A_L). In our scenario there are two sources of errors for matching paths extracted from the map: Firstly, paths with a too short encoding length to be unique in the network cannot be matched at all, secondly a path with a unique encoding might still allow for a small range of different path beginnings (before the first inflection point) and path tails (after the last inflection point). But based on the density of inflection points on shortest paths, these disturbance sources have only a mild effect on the accuracy. For both quality measures we never observed an error value greater than 0.06 for $t = 5$ and 0.04 for exact queries, with the A_L value always being slightly better than A_N. So both error metric values are remarkably small for all considered graphs, even under imprecisions. To mitigate the imprecision even further when determining the current position of the vehicle, path prediction algorithms [4] can be used on the basis of the matched trajectory.

6.5 Real-World Data

To demonstrate the practicability of our approach on real-world data, we collected electronic compass measurements with our app (installed on a Nexus 4) over a period of a month on the same trajectory (so 20 measurements in total). The data was collected by bicycle, the travelled path includes streets and bicycle paths. Figure 4 provides a visualization of the trajectory in our web application. In Figure 5 we show a comparison of all introduced compass measurement techniques supported by our app. The results implied that using the average over all methods works best. (For AHRS, we tested several parameter combinations and finally considered three of them useful. There might be of course better ways to use the method.) As ground truth we used the representation of the trajectory in the underlying network. Conducting our compass measurements, we observed that the total travelled time and therefore the number of raw measurements vary significantly (by up to 20%), but the number of inflection points is very steady (11 for 18 out of 20 trajectories, 10 and 12 for the two remaining ones).

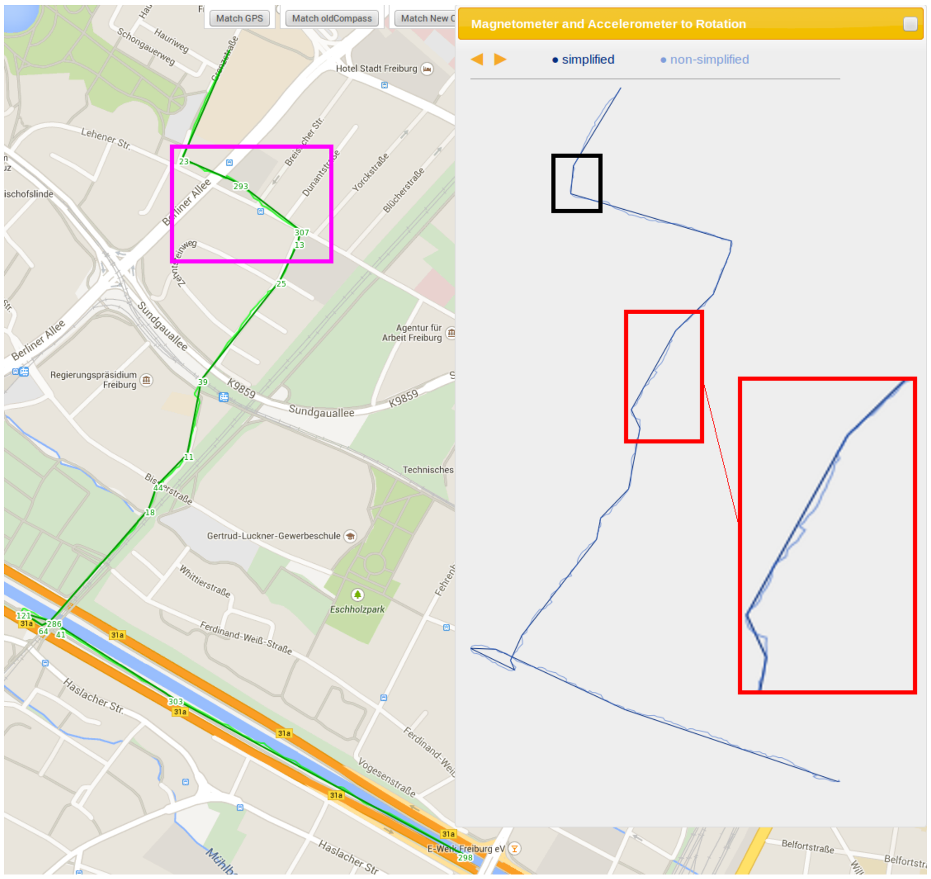

Fig. 4. GPS-based trajectory (green) visualized in the map on the left, and compass-based measurements on the same trajectory presented in a pop-up on the right side. The red box shows a zoomed-in version of part of the compass path to illustrate our line smoothing technique. The overall shape is very faithfully preserved. The pink box on the left shows a trajectory section where GPS measurements imply a curve, but the compass path truthfully reports a straight line there. The black square on the right shows a sharp turn which is not represented as single turning point in the compass path. But as we use inflection point representation, this does not affect the encoding.

In the end, we could match 13 out of 20 trajectories to the ground truth using an angle tolerance of 5°. For the remaining ones, at least one inflection point differed more from the ground truth, the maximum was at 22°. Nevertheless, considering the scenario where as soon as we found our position in the network (using e.g. the first four inflection points) and then only update our position as soon as new inflection points come in, we followed the correct trajectory in the map from beginning to end for 19 out of 20 trajectories.

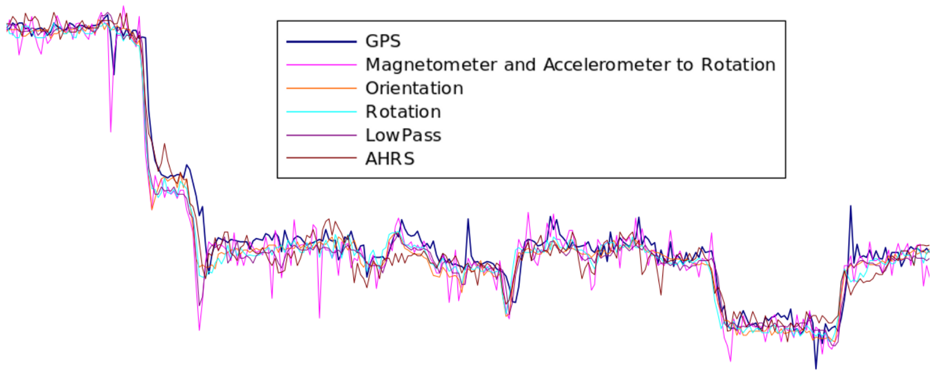

Fig. 5. Comparison of different compass measurement techniques and GPS based directions on part of our test trajectory. We observe that in general the reported values are very similar, with some outlier peaks for the Magnetometer and Accelerometer and AHRS.

So there is a clear indication that compass paths in IPR can work as stand-alone for precise localization (if the sensor quality is sufficient). Of course, experiments on a single trajectory are not that meaningful. In future work, a large scale study should be conducted to retrieve more information about compass-based navigation. But seeing that the scheme works for bicycles already gives hope that it might work even better for cars. As when going by bicycle one tends not to follow lanes exactly and one has a larger degree of freedom on bicycle tracks than cars have on streets.

7 Conclusions

We presented a complete pipeline to acquire compass data in a wireless manner, to process it, and to use the resulting data for localization of a vehicle moving in a street network. One important aspect to make this approach practical is our newly developed inflection point representation, which is robust against different driving styles and time-dependent data sampling. To allow for real-time localization, we designed a data structure based on generalized suffix trees, which encodes the whole street network compactly, and allows to search for a compass path in inflection point representation in fractions of a second. Future work should include more real-world experiments, better sensor value trade-offs (instead of the simple average we used), and compatibility tests with other (autonomous) information sources.

References

1. Quddus, M., Ochieng, W., Noland, R.: Current map-matching algorithms for transport applications: state-of-the art and future research directions. Transportation Research Part C: Emerging Technologies **15** (2007) 312 – 328
2. Alt, H., Efrat, A., Rote, G., Wenk, C.: Matching planar maps. J. Algorithms **49** (2003) 262–283
3. Yanagisawa, H.: An offline map matching via integer programming. In: Proc. 20th International Conference on Pattern Recognition (ICPR), IEEE (2010) 4206–4209
4. Eisner, J., Funke, S., Herbst, A., Spillner, A., Storandt, S.: Algorithms for matching and predicting trajectories. In: Proc. of the 13th Workshop on Algorithm Engineering and Experiments (ALENEX). (2011)
5. Funke, S., Storandt, S.: Path shapes: an alternative method for map matching and fully autonomous self-localization. In: Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM (2011) 319–328
6. Firner, B., Sugrim, S., Yang, Y., Lindqvist, J.: Elastic pathing: Your speed is enough to track you. CoRR **abs/1401.0052** (2014)
7. Aggarwal, P., Thomas, D., Ojeda, L., Borenstein, J.: Map matching and heuristic elimination of gyro drift for personal navigation systems in gps-denied conditions. Measurement Science and Technology **22**(2) (2011) 025205
8. Pei, L., Chen, R., Liu, J., Liu, Z., Kuusniemi, H., Chen, Y., Zhu, L.: Sensor assisted 3d personal navigation on a smart phone in gps degraded environments. In: Geoinformatics, 2011 19th International Conference on. (June 2011) 1–6
9. Douglas, D., Peucker, T.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. The Canadian Cartographer **10**(2) (1973) 112–122
10. de Berg, M., Cook IV, A.: Go with the flow: The direction-based frechet distance of polygonal curves. In: Proc. 1st Int. ICST Conf. on Theory and Practice of Algorithms in Computer Systems (TAPAS). (2011)
11. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14** (1995) 249–260 10.1007/BF01206331.