The Fréchet Distance Unleashed: Approximating a Dog with a Frog

Sariel Har-Peled
Department of Computer Science, University of Illinois, Urbana, IL, USA

Benjamin Raichel
Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA

Eliot W. Robson
Department of Computer Science, University of Illinois, Urbana, IL, USA

Abstract

We show that a variant of the continuous Fréchet distance between polygonal curves can be computed using essentially the same algorithm used to solve the discrete version. The new variant is not necessarily monotone, but this shortcoming can be easily handled via refinement.

Combined with a Dijkstra/Prim type algorithm, this leads to a realization of the Fréchet distance (i.e., a morphing) that is locally optimal (aka locally correct), that is both easy to compute, and in practice, takes near linear time on many inputs. The new morphing has the property that the leash is always as short as possible. These matchings/morphings are more natural, and are better than the ones computed by standard algorithms – in particular, they handle noise more graciously. This should make the Fréchet distance more useful for real world applications.

We implemented the new algorithm, and various strategies to obtain fast practical performance. We performed extensive experiments with our new algorithm, and released publicly available (and easily installable and usable) `Julia` and `Python` packages. In particular, the `Julia` implementation, for computing the regular Fréchet distance, seems to be **significantly faster** than other currently available implementations. See Table 2.2. Our algorithms can be used to compute the almost-exact Fréchet distance between polygonal curves.

Implementations and numerous examples are available here: frechet.xyz.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Curve similarity, Fréchet distance

Digital Object Identifier 10.4230/LIPIcs.SoCG.2025.54

Related Version *Full Version*: `https://arxiv.org/abs/2407.03101` [14]

Supplementary Material *Audiovisual (Examples)*: `https://frechet.xyz`
Software (Python Implementation): `https://github.com/eliotwrobson/FrechetLib` [17]
archived at `swh:1:dir:cf434ca17d61d3d43c67e7bc40fd2eb2b507c9d9`
Software (Julia Implementation): `https://github.com/sarielhp/FrechetDist.jl` [13]
archived at `swh:1:dir:9600673d8c20bc49890fd55c71bd6e5c841dc7df`

Funding *Sariel Har-Peled*: Work on this paper was partially supported by a NSF AF award CCF-2317241.
Benjamin Raichel: Work on this paper was partially supported by NSF AF Award CCF-2311179.
Eliot W. Robson: Work on this paper was partially supported by a NSF AF award CCF-2317241.

1 Introduction

1.1 Definitions

Given two polygonal curves, their Fréchet distance is the length of a leash that a person needs, if they walk along one of the curves, while a dog connected by the leash walks along the other curve, assuming they synchronize their walks so as to minimize the length of this leash. (I.e. they walk so as to minimize their maximum distance apart during the walk.) Our approach is slightly different than the standard approach, and we define it carefully first.

1.1.1 Free space diagram and morphings

▶ **Definition 1.** *For a (directed) curve $\pi \subseteq \mathbb{R}^d$, its* **uniform parameterization** *is the bijection $\pi : [0, \|\pi\|] \to \pi$, where $\|\pi\|$ is the length of π, and for any $x \in [0, \|\pi\|]$, the point $\pi(x)$ is at distance x (along π) from the starting point of π.*

▶ **Definition 2.** *The free space diagram of two curves π and σ is the rectangle $R = R(\pi, \sigma) = [0, \|\pi\|] \times [0, \|\sigma\|]$. Specifically, for any point $(x, y) \in R$, we associate the* **elevation function** *$\mathcal{e}(x, y) = \|\pi(x) - \sigma(y)\|$.*

The free space diagram R is partitioned into a non-uniform grid, where each cell corresponds to all leash lengths when a point lies on a fixed segment of one curve, and the other lies on a fixed segment of the other curve, see Figure 1.1. For a given value $\delta \geq 0$, the *sublevel set* of a real valued function consists of all inputs whose function value is $\leq \delta$. It is known that for any value $\delta \geq 0$, the sublevel set of the elevation function inside such a grid cell is a clipped ellipse.

▶ **Definition 3.** *A* **morphing**[1] *m between π and σ is a (not self-intersecting) curve $m \subseteq R(\pi, \sigma)$ with endpoints $(0, 0)$ and $(\|\pi\|, \|\sigma\|)$. The set of all morphings between π and σ is $\mathcal{M}_{\pi,\sigma}$. A morphing that is a segment inside each cell of the free space diagram that it visits, is well behaved*[2]*.*

Intuitively, a morphing is a reparameterization of the two curves, encoding a synchronized motion along the two curves. That is, for a morphing $m \in \mathcal{M}_{\pi,\sigma}$, and $t \in [0, \|m\|]$, this encodes the configuration, with a point $\pi\big(x(m(t))\big) \in \pi$ matched with $\sigma\big(y(m(t))\big) \in \sigma$. The ***elevation*** of this configuration is $\mathcal{e}(t) = \mathcal{e}(m(t)) = \big\|\pi\big(x(m(t)\big) - \sigma\big(y(m(t)\big)\big\|$.

1.1.2 Fréchet distance

▶ **Definition 4.** *The* **width** *of a morphing m between π and σ is $\omega(m) = \max_{t \in [0, \|m\|]} \mathcal{e}(t)$. The* **Fréchet distance** *between the two curves π and σ is*

$$\mathsf{d}_{\mathcal{F}}(\pi, \sigma) = \min_{m \in \mathcal{M}^{+}_{\pi,\sigma}} \omega(m),$$

where $\mathcal{M}^{+}_{\pi,\sigma} \subseteq \mathcal{M}_{\pi,\sigma}$ is the set of all x/y-monotone morphings.

[1] A morphing induces a natural homotopy between the two curves.
[2] All the morphings we deal with in this paper are well behaved.

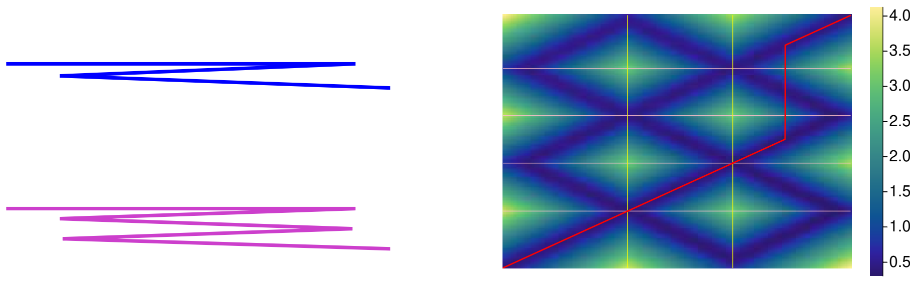
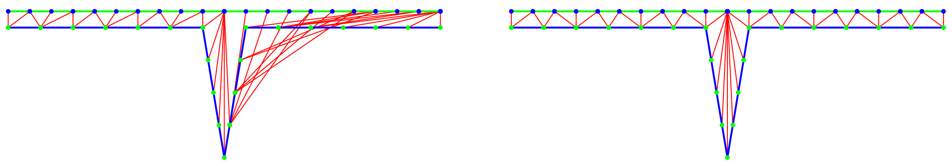


Figure 1.1 Two curves, their free space diagram (and the associated elevation function), and the optimal Fréchet morphing between the two curves encoded as an x/y-monotone curve. More illustrations and animations of this example are available here.



The classical (discrete) Fréchet morphing, caring only about the maximum leash length.

The retractable discrete Fréchet morphing, using the shortest leash possible at each point.

Figure 1.2 A comparison between the classical and retractable Fréchet distances. Observe that the morphing generated by the classical distance can be quite loose in many places. An animation of both morphings is available here.

Conceptually, the Fréchet distance is the problem of computing the minimum bottleneck matching between two curves, respecting the order and continuity of the curves.[3] Alternatively, it is an L_∞-norm type measure of the similarity between two curves. It thus suffers from sensitivity to outliers. Furthermore, even if only a small portion of the morphing requires a long leash, the measure, and the algorithms computing it, may use this long leash in large portions of the walk, generating a matching that is loose in many places, see Figure 1.2.

Observe that the Fréchet distance is the minimum value such that the sublevel set of the elevation function has an x/y-monotone path from $(0, 0)$ to $(\|\pi\|, \|\sigma\|)$ in R.

1.2 Background

Alt and Godau [1] presented a rather involved $O(n^2 \log n)$ time algorithm to compute the Fréchet distance using parametric search. The parametric search can be removed by using randomization, giving a simpler algorithm as shown in [12]. Buchin *et al.* [7] presented an alternative algorithm for computing the Fréchet distance that replaces the decision procedure by using a data-structure to maintain appropriate lower envelopes. Despite some simplifications, all these algorithms are somewhat involved.

Unfortunately, it is believed this problem requires quadratic time in the worst case, although a logarithmic speedup is possible; see [5] and references therein. The quadratic time can be improved for realistic inputs by assuming that the input is "nice", and introducing approximation, but the resulting algorithms are still not simple [9].

[3] Formally, since the reparameterization is not one-to-one, this is not quite a matching. One can restrict to using only such bijections, with no adverse effects, but it adds a level of tediousness, which we avoid for the sake of simplicity of exposition.

1.2.1 Variants of the Fréchet distance

Weak Fréchet distance. The ***weak Fréchet distance*** allows morphings where the agents are allowed to go back and traverse portions of the curve visited previously (i.e., the morphing does not have to be x/y-monotone). Since the weak Fréchet distance allows considering more parameterizations, it is potentially smaller, and we have that $\mathsf{d}_{\mathcal{F}}^{w}(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$, for any two curves π, σ. Elegantly, Alt and Godau [1] showed the weak Fréchet distance can be reduced to computing the minimum spanning tree of an appropriate graph. Unfortunately, there does not seem to be a natural way to overcome this non-monotonicity (and thus get the "strong" version).

Discrete Fréchet distance. The complexity of these algorithms, together with sensitivity of the Fréchet distance to noise, lead to using "easier" related measures, such as the discrete version of the problem, and dynamic time-warping (discussed below). In the discrete version, you are given two sequences of points $p_1, \ldots, p_n$, and $q_1, \ldots, q_m$, and the purpose is for two "frogs" starting at p_1 and q_1, respectively, to jump through the points in the sequence until reaching p_n, q_m, respectively, while minimizing the maximum distance between the two frogs during this traversal. At each step, only one of the frogs can jump from its current location to the next point in its sequence (no jumping back). Computing the optimal distance under this measure can be done by dynamic-programming, similar to the standard approach to edit-distance. Indeed, the configuration space here is the grid $H = [\![n]\!] \times [\![m]\!]$, where $[\![n]\!] = \{1, \ldots, n\}$.

To use the discrete version in the continuous case, one sprinkles enough points along both input curves, and then solves the discrete version of the problem. Beyond the error this introduces, to get a distance that is close to the standard Fréchet distance, one has to sample the two curves quite densely in some cases.

For the (monotone) discrete Fréchet distance the induced graph on the grid H is a DAG, and the task of computing the Fréchet distance is to find a minimum bottleneck path from $(1,1)$ to (n,m), where the weights are on the vertices. Here, the weight on the vertex (i,j) is the distance $\|p_i - q_j\|$. In particular, a Fréchet morphing is an x/y-monotone path in H from $(1,1)$ to (n,m). The standard algorithm to do this traverses the grid, say, by increasing rows i, and in each row by increasing column j, such that the value at (i,j) is the maximum of the length of the leash of this configuration, together with the minimum solution for $(i-1,j)$ and $(i,j-1)$. This algorithm leads to a straightforward, $O(nm)$ time algorithm for the discrete Fréchet distance. However, the morphing computed might be inferior, see Figure 1.2 for such a bad example.

Retractable Fréchet. For simplicity, assume that the pairwise distances between all pairs of points in the two sequences are unique. We would like to imagine that we have a retractable leash, that can become shorter at times, and the leash "resists" being longer than necessary. It is thus natural to ask for a morphing where the leash as short as possible at any point in time.

Informally, the optimal *retractable Fréchet morphing* between the two sequences includes the bottleneck configuration, realizing the Fréchet distance, in the middle of its path, and the two subpaths from the endpoints to this configuration have to be also recursively optimal. This is formally defined and described in the Full version of the paper [14]. This concept was introduced by Buchin *et al.* [6]. Interestingly, they show that the discrete version can be computed in $O(nm)$ time, but unfortunately, the algorithm is quite complicated. They also show that the continuous retractable Fréchet can be computed in $O(n^3 \log n)$ time.

Buchin *et al.* [6] refers to this Fréchet distance as *locally correct*, but we prefer the *retractable* labeling. The term "retractable Fréchet" was used by Buchin *et al.* [7], but in a different (and not formally defined) context than ours.

(Continuous) Dynamic Time Warping. One way to get less sensitivity for noise is to compute the total area "swept" by the leash as the walk is being performed. In the discrete case, we simply add up the lengths of the leashes during the configurations in the walk. There is also work on extending this to the continuous setting [15, 2]. For the continuous case this intuitively boils down to computing (or approximating) an integral along the morphing. See full version of the paper [14] for more details.

1.2.2 Critical events

The standard algorithm for computing the Fréchet distance works by performing a "binary" search for the Fréchet distance. Given a candidate distance, it constructs a "parametric" diagram that is a grid, where inside each grid cell the feasible region is a clipped ellipse. The task is then to decide if there is an x/y-monotone path from the bottom left corner to the top right corner, which is easily doable. The critical values the search is done over are:

(I) *Vertex-vertex events*: The distance between two vertices of the two curves,
(II) *Vertex-edge events*: The distance between a vertex of one curve, and an edge of the other.
(III) *Monotonicity events*: This is the minimum distance between a point on one edge e of the curves, and (maximum distance to) two vertices u, v of the other curve. Specifically, it is realized by the point on e with equal distance to u and v.

The first two types of events are easy to handle, but the monotonicity events are the bane of the algorithms for the Fréchet distance.

1.2.3 Algorithm engineering the Fréchet distance

Given the asymptotic complexity and involved implementations of the aforementioned algorithms, there has been substantial work on practical aspects of computing the Fréchet distance. In particular, in 2017, ACM SIGSPATIAL GIS Cup had a programming challenge to implement algorithms for computing the Fréchet distance. See [19] for details.

More recently, Bringmann *et al.* [3] presented an optimized implementation of the decider for the Fréchet distance. Somewhat informally, Bringmann *et al.* [3] builds a decider for the Fréchet distance using a kd-tree over the free space diagram, keeping track of the reachable regions on the boundary of each cell, refining cells by continuing down the (virtual) kd-tree if needed.

1.3 Our results

1.3.1 Result I: A new algorithm for retractable discrete Fréchet

We observe that a natural approach to compute the retractable Fréchet morphing is to modify Dijkstra's/Prim's algorithm so that it solves the minimum bottleneck path problem. This observation leads to a simpler (but log factor slower) algorithm for computing it. The only modification of Dijkstra necessary is that one always handles the cheapest edge coming out of the current cut induced by the set of vertices already handled. (In the discrete Fréchet case, the weights are on the vertices, but this is a minor issue.) This modified version of

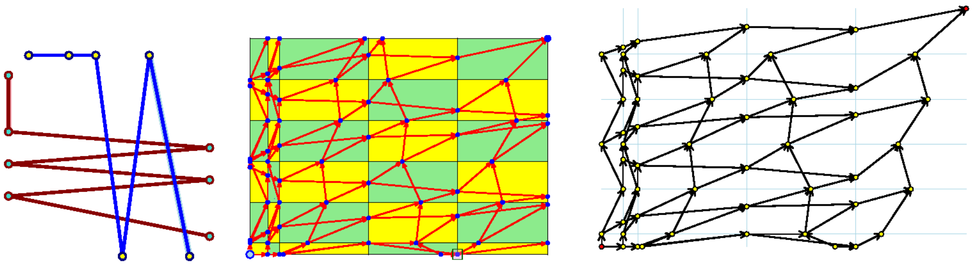


Figure 1.3 Two curves, and their associated VE-Fréchet graph. Note, that every internal edge of the grid contains a portal (i.e., vertex of the graph), but in many cases the portals of two edges are co-located on their common vertex (see the squared marked vertex in the middle figure) For the edges adjacent to the starting and ending corners we set their portals to lie at the corners themselves. See here for more info about a similar example.

Dijkstra is well known, but we include, for the sake of completeness, the proof showing that it indeed computes the recursively optimal path, which is also a retractable Fréchet morphing between the two sequences. This immediately leads to better and more natural morphings, see Figure 1.2.

Maybe more importantly, in practice, one does not need to explore the whole space of nm configurations (since we are in the discrete case, a configuration $(i, j) \in [\![n]\!] \times [\![m]\!]$ encodes the matching of p_i with q_j), as the algorithm can stop as soon as it arrives at the destination configuration (n, m). Informally, if the discrete Fréchet distance is “small” compared to the vast majority of pairwise distances (i.e., the two sequences are similar), then the algorithm only explores a small portion of the configuration space. Thus, this leads to an algorithm that is faster than the standard algorithm in many natural cases, while computing a significantly better output morphing.

1.3.2 Result II: A new distance and algorithm: VE-Fréchet

It is not clear how to extend the above to the continuous case. A natural first step is to consider the continuous Fréchet distance, where one restricts the solution inside each cell of the free space diagram to be a segment (which is already the case for the morphings computed by existing algorithms), but more importantly, insisting that the shape of this segment must be determined only locally, thus facilitating a greedy strategy compatible with the retractable approach. In practical terms, we throw away the (global) monotonicity events.

Traveling only through vertex-edge events. Because of the continuity and strict convexity of the elevation function, the function has a unique minimum on each edge of the free space diagram grid – geometrically, this is the minimum distance between a vertex of one curve, and an edge of the other curve (a *vertex-edge event*). We restrict our solution to enter and leave a cell only through these *portals* (which are easy to compute). The continuous configuration space now collapses to a discrete graph that is somewhat similar to the natural grid graph on $[\![n]\!] \times [\![m]\!]$. Indeed, a grid cell has four portals on its boundary edges. Specifically, there are directed edges from the portal on the bottom edge, to the portal on the top and right edges of the cell. Similarly, there are edges from the portal on the left edge, to the portals lying on the top and right edges. As before, the values are on the portals, and our purpose is to compute the optimal bottleneck path in this grid-like graph. Examples of this graph are depicted in Figure 1.3, Figure 1.4 and Figure 1.5.

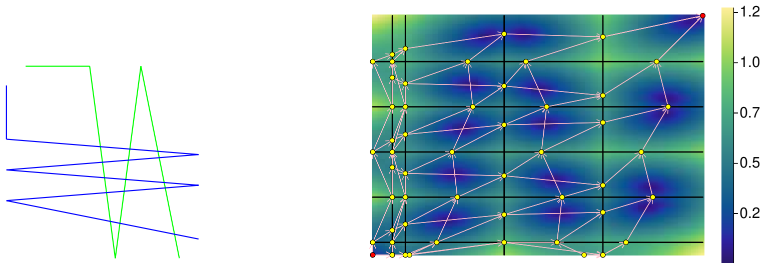


Figure 1.4 Left: Two curves. Right: Their elevation function, and the associated graph. See here for more info.

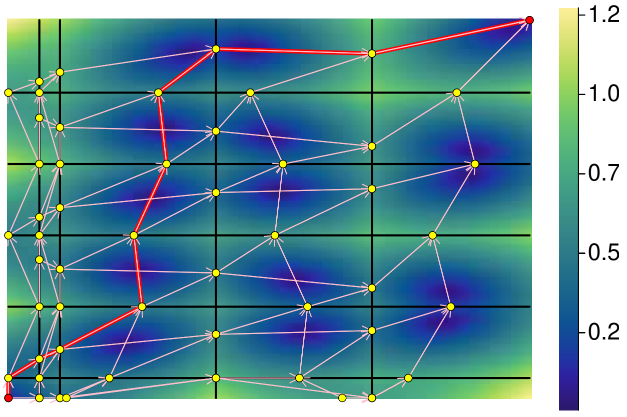


Figure 1.5 The VE-Fréchet morphing for the two curves from Figure 1.3. Note, that the solution is "slightly" not x/y-monotone. For animation of this morphing, follow the link.

▶ Remark 5. A somewhat similar idea was used by Munich and Perona [15], but they used it in the other direction – namely, in defining a better CDTW distance for two discrete sequences. However, this idea was already present (implicitly) in the original work of Alt and Godau [1] – indeed, their algorithm for the Weak Fréchet distance uses only the Vertex-Edge events (i.e., edges in the free space diagram). This boils down to solving the bottleneck shortest path problem in an undirected graph. In this case, this problem can be solved by computing the minimum spanning tree (e.g., by Prim's algorithm, which is a variant of Dijkstra's algorithm), as Alt and Godau do. For the directed case, one needs to use a variant of Dijkstra's algorithm [11] – see full version of the paper [14]. See also Buchin *et al.* [4] who also used a similar idea.

We can now run the retractable bottleneck shortest-path algorithm (i.e., the variant of Dijkstra described above) on this implicitly defined graph computing the vertices and edges of it, as they are being explored. For many natural inputs, this algorithm does not explore a large fraction of the configuration space, as it involves distances that are significantly larger than the maximum leash length needed. The algorithm seems to have near linear running time for many natural inputs. The ***VE-Fréchet*** morphing is the one induced by the computed path in this graph. Unfortunately, the VE-Fréchet morphing might allow the agents to move backwards on an edge, but importantly, the motion across a vertex is monotone. Namely, the VE-Fréchet is monotone for vertices, but not necessarily monotone on the edges. A vertex is thus a *checkpoint* that once passed, cannot be crossed back.

1.3.3 Result III: New algorithm for the regular Fréchet distance

The natural question is how to use the (easily computable) VE-Fréchet morphing to compute the optimal (regular) continuous Fréchet distance. We next describe how this can be done in practice.

The hunt for a monotone morphing. We denote the VE-Fréchet distance between two curves π and σ by $\mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$. Clearly, we have that $\mathsf{d}_{\mathcal{F}}^{w}(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$.

One can of course turn any morphing into a monotone one by staying put instead of moving back. This is appealing for VE-Fréchet, as the corresponding VE morphing $\boldsymbol{m}$, say between two curves π and σ, never backtracks over vertices (only edges), so we already expect the error this introduces to be relatively small. Let $\boldsymbol{m}^+$ denote the monotone morphing resulting from this simple strategy. Observe that

$$\omega(\boldsymbol{m}) = \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma) \leq \omega(\boldsymbol{m}^+).$$

In particular, if $\omega(\boldsymbol{m}) = \omega(\boldsymbol{m}^+)$, then $\mathsf{d}_{\mathcal{F}}(\pi, \sigma)$ is realized by $\boldsymbol{m}^+$, and we have computed the Fréchet distance between π and σ.

A less aggressive approach is to introduce new vertices in the middle of the edges of π and σ as to enforce monotonicity. Indeed, clearly, if we refine both curves by repeatedly introducing vertices into them, the VE-Fréchet distance between the two curves converges to the Fréchet distance between the original curves, as introducing a vertex in the middle of an edge does not change the regular Fréchet distance, while preventing the VE-Fréchet morphing from backtracking over this point.

We refer to this process of adding vertices to the two curves as ***refinement***. (See Figure 1.6.) In practice, in many cases, one or two rounds of (carefully implemented) refinement are enough to isolate the maximum leash in the morphing from the non-monotonicity, and followed by the above brute-force monotonization leads to the (practically) optimal Fréchet distance. Even for pathological examples, after a few more rounds of refinement, this process computes the almost-exact Fréchet distance. That is, the computed lower bound, which is the VE-Fréchet distance, is equal to the width of the computed monotone morphing, which is the Fréchet distance.

▶ Remark 6 (Almost-exact: Floating point issues). As we are implementing our algorithm using floating point arithmetic, and the calculation of the optimal Fréchet distance involves distances, impreciseness is unavoidable. A slight improvement in preciseness can be achieved by using squared distances (and also slightly faster code) – but for simplicity we have not used this idea in our code. In particular, we take the somewhat pragmatic view, that an approximation to the optimal up to a factor of (say) 1.00001 can be considered as computing the “optimal” solution. We refer to such solutions as being ***almost-exact***.

Note that Fréchet morphings are somewhat less sensitive to numerical issues than other geometric problems – indeed, once a morphing is computed, one can compute its width directly.

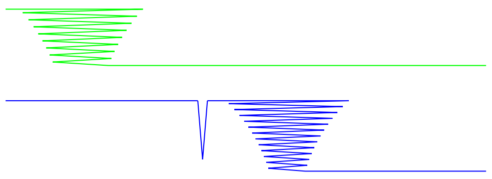


Figure 1.6 For these two curves, the solution involves an agent stopping at one point on one curve while the other agent traverses is zig-zag, and vice versa. The algorithm enforces monotonicity by refining the two curves by introducing new vertices. For the results, see here.

▶ Remark 7 (What if one wants the exact distance?). As pointed above, our algorithm computes the almost-exact Fréchet distance, and in practice there is no difference to the exact Fréchet distance (and in many cases they seem to coincide). Nevertheless, what if one insists on the exact Fréchet distance?

The refinement process can be modified to compute the monotonicity events on the regions on the curves where monotonicity is being violated. This would readily lead to an exact algorithm – from implementation point of view such a modification seems pointless, and we did not pursue it any further.

1.3.4 Result IV: Computing the Fréchet distance quickly for real inputs

The above leaves us with a natural strategy for computing the Fréchet distance between two given curves. Compute quickly, using simplification, a morphing between the two input curves, and maintain (using VE-Fréchet, for example) both upper and lower bounds on the true Fréchet distance. By carefully inspecting the morphing, (re)simplifying the curves in a way that is sensitive to their (local) Fréchet distance, and recomputing the above bounds, one can get an improved morphing. Repeat this process potentially several times till the upper and lower bounds meet, at which point the optimal Fréchet distance has been computed.

This seems somewhat overkill, but it enables us to compute (in practice) the almost-exact Fréchet distance between huge polygonal curves quickly.

1.3.5 Main contribution: Implementation in `Julia` and `Python`

We implemented the above algorithms as official packages in `Python` and `Julia`. The `Python` package is available at `https://github.com/eliotwrobson/FrechetLib`, and the `Julia` package is available at `https://github.com/sarielhp/FrechetDist.jl`. Animations and examples computed by the new algorithm are available at `https://frechet.xyz/`.

1.3.6 Additional results

Sweep distance. We demonstrate how one can convert our algorithm for computing VE-Fréchet to an algorithm that computes a variant of the CDTW distance, which we call the *sweep distance*. One can then use refinement to approximate the CDTW distance. One can also compute a lower bound on this quantity, and the two quantities converge. See the full version of the paper [14] for details.

Fast simplification. We show how to preprocess a curve π with n vertices, in $O(n \log n)$ time, such that given a query w, one can quickly extract a simplification of $\overline{\pi}$ of π, such that $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) \leq w$. Importantly, the time to extract $\overline{\pi}$ is proportional to its size (i.e., the extraction is output sensitive). More importantly, in practice, this works quite well – the size of $|V(\overline{\pi})|$ is reasonably close (by a constant factor) to the optimal approximation. Combined with greedy simplification, this yields a very good simplification result. See the full version of the paper [14] for details.

1.4 Significance

We demonstrate in this paper that a minor variant of the **continuous** Fréchet distance can be computed by a strikingly simple Dijkstra type algorithm. Furthermore, for many real world inputs it runs in near linear time (out of the box). Similarly, for many real world inputs, the computed morphing is monotone, thus realizing the (standard) Fréchet distance. More importantly, because of the *retractable* nature of the morphing computed, the matching computed is more natural, and can handle noise/outliers more gracefully than the matchings computed by the current algorithms for the Fréchet distance. Indeed, areas that are noise/outliers are isolated in the morphing to a small interval which can be easily identified and handled, see Figure 1.2. We believe that this makes the morphings computed by our new algorithms significantly better for real world applications than previous algorithms. There is also previous work on other variants of Fréchet distance that are more robust to uncertainty and outliers [10], and shortcutting [8].

We then show how to modify this algorithm to compute the (monotone) Fréchet distance (in cases when the morphing was not already monotone), and how to make it handle large inputs quickly via simplifications (handling all the technical difficulties this gives rise to).

Finally, we implemented our new algorithms in `Julia` and `Python`, and made them publicly available as standard packages, thus making the computation of the Fréchet distance accessible to a wider audience. Using such packages in `Python`/`Julia` is significantly easier than using any not pre-compiled code in `C++`.

Summary. The combination of simplification, new (and not so new) algorithmic ideas (such as retractablity via Dijkstra, and VE-Fréchet, among others), and careful implementation, leads to fast performance in practice beating other implementations. We discuss our implementation next – the algorithmic ideas described above are covered in detail in the appendices.

Full version. Due to space limitations, all the low-level details are omitted from this version. See the full version of the paper [14] (`https://arxiv.org/abs/2407.03101`) for these details.

2 Implementation and experiments

We include here some tables with information about the performance of our implementation. For details about the implementation, see the full version of the paper [14].

2.1 Discussion

Overall, it is clear that our `Julia` implementation is faster than other currently available implementations for computing the Fréchet distance. Beyond that, `Julia` seems like a great programming language to develop geometric algorithms – providing a combination of a safe, multi-threaded, high-level programming language with the performance of `C++`, without the pain involved in working in `C++`.

Table 2.1 The inputs tested and where they are taken from, `birds` referring to the stork migration dataset [18], the GeoLife dataset [20], and `Pigeons` referring to a dataset from [16].

Input	Description
1	`birds:  1787_1 / 1797_1`
2	`birds:  2307_3 / 2859_3`
3	`birds:  2322_2 / 1793_4`
4	`GeoLife 20080928160000 / 20081219114010`
5	`GeoLife 20090708221430 / 20090712044248`
6	`Pigeons RH887_1 / RH887_11`
7	`Pigeons C369_5 / C873_6`
8	`Pigeons C360_10 / C480_9`

Table 2.2 `Julia`/`C++` comparison, demonstrating that the `Julia` implementation is significantly faster. The `C++` implementation is from Bringmann *et al.* [3]. Running times are in seconds. `Julia` MT stands for the multi-threaded implementation – since multi-threading is easy in Julia, and the task at hand is easily parallizable, we tested this, but of course this should not be taken as a direct (or remotely fair) comparison to the `C++` implementation, and is provided for the reader amusement. In particular, the multi-threading done is pretty naive, and better performance should be possible by better fine-grained partition of the tasks. The tests were performed on a Linux system with 64GB memory with Intel `i7-11700` CPU, with a decent 16 threads CPU, but far from the fastest hardware currently available. The "# pairs" column is the number of pairs of curves that had their Fréchet distance compared during this test. The "avg # points" is the average number of vertices per curve in this input set – underlying the benefit of simplification for the GeoLife input set.

			Total RT in seconds			
Dataset	# pairs	avg # points	`C++`	`Julia`	`Julia` MT	`C++ / Julia`
sigspatial	322, 000	247.8	145	102	25	**1.42**
Characters	253, 000	120.9	90	72	18	**1.25**
Geolife	322, 000	1080.4	646	70	30	**9.22**
All tests combined			881	244	73	**3.61**

One optimization used by the `Julia` code, that should be generally useful, is the following. As a preprocessing step, precompute a hierarchy of simplified curves for each input curve. This improves the query process, but makes the preprocessing more expensive. Thus, storing such precomputed hierarchies might be a good idea if input curves are going to be used repeatedly for performing distance queries. We emphasize that the reported running times include this (light) preprocessing stage (interestingly, the SIGSPTIAL [19] competition allowed such preprocessing to not be included in the overall running time).

References

1 H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *Int. J. Comput. Geom. Appl.*, 5:75–91, 1995. `doi:10.1142/S0218195995000064`.

2 Milutin Brankovic, Kevin Buchin, Koen Klaren, André Nusser, Aleksandr Popov, and Sampson Wong. (k, l)-medians clustering of trajectories using continuous dynamic time warping. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '20, pages 99–110, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3397536.3422245`.

Table 2.3 `Julia` performance on various inputs. All running times are is seconds. The # columns specify the number of vertices in each input. The single missing running time is for a case where the program ran out of memory. The quality of approximation is in the title of the column. The VER column is for running the VE-Fréchet (retractable) algorithm on the original input curves, which is much slower than the exact algorithm, which computes the exact Fréchet distance, but uses simplification internally for speed.

Input	P #	Q #	≈4	≈1.1	≈1.01	≈1.001	Exact	VER
1	10,406	11,821	0.106	0.256	4.164	16.639	0.476	27.743
2	16,324	14,725	0.114	0.428	0.985	4.762	3.522	19.734
3	22,316	4,613	0.151	0.651	1.207	3.262	0.882	7.138
4	56,730	91,743	0.578	1.207	4.215	28.109	4.259	--
5	6,103	9,593	0.043	0.040	0.305	0.519	0.484	5.678
6	2,702	1,473	0.034	0.084	1.367	1.387	0.160	1.081
7	1,068	1,071	0.026	0.066	0.048	0.049	0.044	0.011
8	864	1,168	0.018	0.044	0.142	0.124	0.069	0.080

Table 2.4 Python performance in seconds on the inputs given in Table 2.1. The missing runtime is for a case where the test code ran out of memory.

Input	P #	Q #	≈4	≈1.1	≈1.01	Exact	VER
1	10,400	11,815	0.071	0.250	5.372	0.593	92.869
2	16,318	14,719	0.091	0.240	1.142	69.525	66.844
3	22,310	4,607	0.087	0.232	1.041	49.426	25.032
4	56,730	91,743	0.307	0.563	1.892	2.390	--
5	6,103	9,593	0.042	0.032	0.175	3.657	19.618
6	2,696	1,467	0.077	0.294	4.913	2.946	283.538
7	1,062	1,065	0.145	0.275	0.455	0.666	382.455
8	858	1,162	0.034	0.100	1.537	0.159	0.709

3 Karl Bringmann, Marvin Künnemann, and André Nusser. Walking the dog fast in practice: Algorithm engineering of the fréchet distance. *J. Comput. Geom.*, 12(1):70–108, 2021. `doi:10.20382/JOCG.V12I1A4`.

4 Kevin Buchin, Maike Buchin, David Duran, Brittany Terese Fasy, Roel Jacobs, Vera Sacristán, Rodrigo I. Silveira, Frank Staals, and Carola Wenk. Clustering trajectories for map construction. In Erik G. Hoel, Shawn D. Newsam, Siva Ravada, Roberto Tamassia, and Goce Trajcevski, editors, *Proc. 25th ACM SIGSPATIAL Int. Conf. Adv. Geog. Info. Sys., GIS*, pages 14:1–14:10. ACM, 2017. `doi:10.1145/3139958.3139964`.

5 Kevin Buchin, Maike Buchin, Wouter Meulemans, and Wolfgang Mulzer. Four soviets walk the dog: Improved bounds for computing the Fréchet distance. *Discrete Comput. Geom.*, 58(1):180–216, 2017. `doi:10.1007/s00454-017-9878-7`.

6 Kevin Buchin, Maike Buchin, Wouter Meulemans, and Bettina Speckmann. Locally correct fréchet matchings. *Comput. Geom. Theory Appl.*, 76:1–18, 2019. `doi:10.1016/J.COMGEO.2018.09.002`.

7 Kevin Buchin, Maike Buchin, Rolf van Leusden, Wouter Meulemans, and Wolfgang Mulzer. Computing the fréchet distance with a retractable leash. *Discret. Comput. Geom.*, 56(2):315–336, 2016. `doi:10.1007/S00454-016-9800-8`.

8 A. Driemel and S. Har-Peled. Jaywalking your dog – computing the Fréchet distance with shortcuts. *SIAM Journal on Computing*, 42(5):1830–1866, 2013. `doi:10.1137/120865112`.

9 A. Driemel, S. Har-Peled, and C. Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *Discrete Comput. Geom.*, 48:94–127, 2012. doi:10.1007/s00454-012-9402-z.

10 Emily Fox, Amir Nayyeri, Jonathan James Perry, and Benjamin Raichel. Fréchet edit distance. In *40th International Symposium on Computational Geometry (SOCG)*, volume 293 of *LIPIcs*, pages 58:1–58:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.SOCG.2024.58.

11 Harold N. Gabow and Robert Endre Tarjan. Algorithms for two bottleneck optimization problems. *J. Algorithms*, 9(3):411–417, 1988. doi:10.1016/0196-6774(88)90031-4.

12 S. Har-Peled and B. Raichel. The Fréchet distance revisited and extended. *ACM Trans. Algo.*, 10(1):3:1–3:22, 2014. doi:10.1145/2532646.

13 Sariel Har-Peled. FrechetDist.jl. Software, version 2.0., swhId: swh:1:dir:9600673d8c20bc49890fd55c71bd6e5c841dc7df (visited on 2025-04-24). URL: https://github.com/sarielhp/FrechetDist.jl, doi:10.4230/artifacts.22985.

14 Sariel Har-Peled, Benjamin Raichel, and Eliot Wong Robson. The fréchet distance unleashed: Approximating a dog with a frog. *CoRR*, abs/2407.03101, 2024. doi:10.48550/arXiv.2407.03101.

15 Mario E. Munich and Pietro Perona. Continuous dynamic time warping for translation-invariant curve alignment with applications to signature verification. In *Proceedings of the International Conference on Computer Vision, Kerkyra, Corfu, Greece, September 20-25, 1999*, pages 108–115. IEEE Computer Society, 1999. doi:10.1109/ICCV.1999.791205.

16 E Pollonara, T Guilford, M Rossi, VP Bingman, and A Gagliardo. Data from: Right hemisphere advantage in the development of route fidelity in homing pigeons, 2017. doi:10.5441/001/1.245kb7r6.

17 Eliot Robson. FrechetLib. Software, version v0.1.1., swhId: swh:1:dir:cf434ca17d61d3d43c67e7bc40fd2eb2b507c9d9 (visited on 2025-04-24). URL: https://github.com/eliotwrobson/FrechetLib, doi:10.4230/artifacts.22984.

18 S Rotics, M Kaatz, SF Turjeman, D Zurell, M Wikelski, N Sapir, U Eggers, W Fiedler, F Jeltsch, and R Nathan. Data from: Early arrival at breeding grounds: causes, costs and a trade-off with overwintering latitude, 2018. doi:10.5441/001/1.v8d24552.

19 Martin Werner and Dev Oliver. ACM SIGSPATIAL GIS cup 2017: range queries under fréchet distance. *ACM SIGSPATIAL Special*, 10(1):24–27, 2018. doi:10.1145/3231541.3231549.

20 Yu Zheng, Hao Fu, Xing Xie, Wei-Ying Ma, and Quannan Li. *Geolife GPS trajectory dataset – User Guide*, geolife gps trajectories 1.1 edition, July 2011. URL: https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/.