Dionysus

A brief overview

Dmitriy Morozov*

1 Introduction

Over the last decade, *persistent homology* [2] has matured from an elegant idea into a rich theory with century-old roots. Two pillars — algebra and fast algorithms — support its foundation. The former has guided its theoretical development; the latter has nurtured a range of applications. Software is the essential bridge between the two: assembling the growing collection of techniques in a single code-base makes the theory accessible to a wider audience.

Although Dionysus' original goal was to facilitate experiments with persistence algorithms, we won't discuss it here. Instead we focus on an accidental benefit. While the core library is written in C++, most of its functionality has been exposed to Python. This high-level interface is both rich and simple, allowing for many sophisticated techniques to be implemented in a few lines of code. Because it follows the C++ API very closely, translating a Python prototype into C++ becomes a straightforward exercise. The library is available at mrzv.org/software/dionysus.

Throughout the overview we assume familiarity with persistence — [1] is an excellent introduction — and give only cursory definitions.

2 Functions and filtrations

The main input to any persistence algorithm is a filtration of a simplicial complex, i.e., a nested sequence of complexes:

$$\emptyset = K_0 \subseteq K_1 \subseteq \ldots \subseteq K_{n-1} \subseteq K_n = K. \tag{1}$$

We can assume without loss of generality that subsequent complexes differ by a single simplex. So we can represent a filtration compactly by an ordered list of simplices $[\sigma_1, \sigma_2, \ldots]$, where complex $K_i = \cup_{j=1}^{i} \sigma_j$. It's rarely practical to construct such a filtration by hand. So, unless it is generated by an external tool, we rely on a few common constructions. Two of them cover many standard scenarios: distance functions and piecewise-linear functions.

2.1 Distance functions

In computational geometry, a distance function is a fruitful representation of the geometry of a compact set. Denoted by $d_P : \mathbb{R}^d \to \mathbb{R}$ for a compact set $P \subseteq \mathbb{R}^d$, it measures everywhere the distance to the closest point in P, $d_P(x) = \inf_{y \in P} \|x - y\|$.

Alpha shapes. The sublevel sets of a distance function are unions of balls, $d_P^{-1}(-\infty, r] = \cup_{p \in P} \mathrm{B}_r(p)$. When the set P is finite, we can clip these balls by the Voronoi regions of their centers to get a collection $\{\mathrm{B}_r(p) \cap \mathrm{Vor}(p)\}_p$. Because the sets are convex, the nerve of this collection, called an *alpha shape*, captures the homotopy type of the union:

$$\mathrm{AS}_r(P) = \mathrm{Nrv}_{p \in P}\{\mathrm{B}_r(p) \cap \mathrm{Vor}(p)\} \simeq \cup_{p \in P} \mathrm{B}_r(p).$$

As we increase the threshold r, the alpha shapes filter the Delaunay triangulation of the point set P. In particular, we can assign to each simplex the smallest value of the distance function on its dual Voronoi cell. In this context, $\mathrm{AS}_r(P)$ is the union of the simplices for which this value does not exceed r. In Python, we can compute the Delaunay triangulation together with the necessary values with the `fill_alpha_complex()` function. Each

*Lawrence Berkeley National Laboratory, dmitriy@mrzv.org. The author would like to acknowledge the support at LBNL of the DOE Office of Science, Advanced Scientific Computing Research, under award number KJ0402-KRD047, under contract number DE-AC02-05CH11231.

generated simplex stores a pair `(value,critical)` in its `data` field. Here `value` determines when the simplex enters the alpha shape, and `critical` is a flag indicating whether the simplex intersects its dual Voronoi cell. Consequently, we can use Python list comprehensions to extract any given alpha shape:

```
simplices = Filtration()
fill_alpha_complex(points, simplices)
alpha_shape = [s for s in simplices if s.data[0] <= alpha]
```

Or get a complete alpha-shape filtration by sorting the simplices with respect to the `data` field and `dimension`:

```
simplices.sort(data_dim_cmp)
```

Class `Filtration` is an auxiliary construction that both records the order of the simplices, and allows fast access to any given simplex.

Vietoris–Rips complexes. Often it is more convenient to work with pairwise distances between points, rather than their explicit embedding in a Euclidean space. In this case, an alternative construction approximates the information captured by the distance function. *Vietoris–Rips complex for parameter r* contains all those simplices whose edge lengths do not exceed r, $\mathrm{VR}_r(P) = \{\sigma \subseteq P \mid \|u - v\| \leq r \ \forall\ u, v \in \sigma\}$.

In Python, we express distances between points through an auxiliary class that knows their number and how to measure a distance between a pair, see the example below. (`PairwiseDistances` provides a shortcut for points in a Euclidean space.) Given an instance `distances` of such a class, we can instantiate a class `Rips` whose method `generate` fills a simplicial complex up to prescribed threshold r and skeleton dimension k. Furthermore, its method `cmp` is suitable for sorting the simplices by increasing edge lengths, while `eval` determines for any simplex the length of its longest edge.

The following example constructs a `DictDistances` class that looks up edge lengths in a dictionary, returning infinity for missing information. It's then used as an input to the `Rips` class to generate a filtration of a 2-skeleton of a Vietoris–Rips complex for parameter $r = 3$.

```
class DictDistances:
    def __init__(self, dictionary, count = None):
        self.d = dictionary
        self.count = count or len(dictionary)

    def __call__(self,i,j):
        if i in self.d and j in self.d[i]:
            return self.d[i][j]
        elif j in self.d and i in self.d[j]:
            return self.d[j][i]
        else:
            return float('inf')

    def __len__(self):
        return len(self.d)


octahedron = {0: {1: 1, 3:1}, 2: {1:1, 3:1}, 4: {0:2, 1:2, 2:2, 3:2}, 5: {0:3, 1:3, 2:3, 3:3}}
distances = DictDistances(octahedron, 6)
rips = Rips(distances)
simplices = Filtration()
rips.generate(2, 3, simplices.append)       # 2-skeleton up to distance 3
simplices.sort(rips.cmp)

for s in simplices:
    print s, rips.eval(s)
```

2.2 Piecewise-linear functions

Given a simplicial complex K and a function $\hat{f} : \mathrm{Vert}\, K \to \mathbb{R}$ defined on the vertices of K, we linearly interpolate it on the interiors of the simplices. The result is a piecewise-linear function $f : K \to \mathbb{R}$. The sublevel sets of

f are not subcomplexes of K — a levelset may cut a simplex in half — so we need a scheme to construct a filtration with the same persistence as f. To solve this problem, we define a *lower-star filtration* of K by setting $K_a = \{\sigma \in K \mid \max_{v \in \sigma} \hat{f}(v) \leq a\}$. In words, the subcomplex K_a contains all the simplices on which the function f does not exceed value a. It's clear from the definition that K_a changes only as a passes the value $\hat{f}(v)$ of one of the vertices. It's not immediately obvious, but true that K_a is homotopy equivalent to $f^{-1}(-\infty, a]$.

Assuming list `values` contains the function $\hat{f}$, where `values[i]`$= \hat{f}(v_i)$, we construct the lower-star filtration of $f : K \to \mathbb{R}$ by sorting the simplices with respect to their highest vertex:

```
def max_vertex_cmp(values):
    def max_vertex(s):
        return max(values[v] for v in s.vertices)

    def compare(si,sj):
        return cmp(max_vertex(si), max_vertex(sj)) or cmp(si.dimension(), sj.dimension())

    return compare

filtration = Filtration(complex, max_vertex_cmp(values))
```

`max_vertex_cmp()` returns a comparison function that remembers the list of vertex `values`. Sorting the filtration, we get the lower-star filtration ordering of all the simplices.

3 Persistence algorithms and diagrams

Given a filtration (1), we can compute its homology as a sequence of vector spaces and linear maps connecting them (we assume homology is computed with coefficients in a field):

$$0 \to \mathsf{H}(K_1) \to \ldots \to \mathsf{H}(K_{n-1}) \to \mathsf{H}(K_n).$$

Persistence keeps track of cycles as they appear and disappear in this filtration and computes a collection of intervals that represent the cycles' lifetimes. We can think of the input to a persistence algorithm as the boundary matrix D, where the columns and rows are ordered with respect to the filtration. In turn, we can interpret standard persistence algorithms as computing a matrix decomposition $R = DV$, where matrix R is reduced, meaning the lowest non-zero entries in its columns fall in unique rows, and matrix V is full-rank and upper-triangular.

The columns of these auxiliary matrices have an immediate interpretation. If the column $R[i] = 0$, then the column $V[i]$ is, by definition, a cycle; it's born with the addition of σ_i in the filtration. If $R[i] \neq 0$, then it's a cycle that dies with the addition of simplex σ_i. Indeed, it's the boundary of the chain $V[i]$, which contains σ_i. The row j of the lowest non-zero entry in column $R[i]$ tells us when the cycle was born — namely, with the addition of simplex σ_j. Notice that matrix R contains all the pairing information.

3.1 Algorithms

In Dionysus, `StaticPersistence` expresses the original algorithm [2], which computes matrix R. If we are interested in both matrices, R and V, we can use the class `DynamicPersistenceChains`. Both have a method `pair_simplices`, which performs the actual computation[1].

```
p = StaticPersistence(filtration)
p.pair_simplices()
```

or

```
p = DynamicPersistenceChains(filtration)
p.pair_simplices()
```

Once the matrices are processed, we can examine their result by iterating over the instances of these classes. An auxiliary map `smap` converts the iterators into the filtration simplices.

[1] `StaticPersistence.pair_simplices()`, by default, uses an optimization [2] that stores only positive simplices in matrix R. This behavior can be turned off by passing `True` to the method.

```
smap = p.make_simplex_map(filtration)
for i in p:
    if i.sign():
        sigma = smap[i]
        if i.unpaired():
            print sigma.dimension(), sigma.data, "inf"
        else:
            tau = smap[i.pair()]
            print sigma.dimension(), sigma.data, tau.data
```

In the `for`-loop, the `sign()` of a simplex determines whether it created a homology class or destroyed it. In the former case, we recover the simplex `sigma` responsible for the birth. We check whether it is `unpaired()`, i.e., whether the homology class it created ever died. If it didn't, we output the `dimension()` and `data` associated with `sigma` as well as `"inf"` to indicate the infinite lifetime. If `sigma` is paired, we recover its `pair()` `tau`, and output `sigma`'s `dimension()` and `data` as the birth time, plus `tau`'s `data` as the death time.

The above snippet outputs the persistence diagram — in fact, all the non-empty persistence diagrams. But sometimes we want more. To recover explicit cycles that were born and later died in the filtration, i.e., the columns of matrix R, we use the `cycle` attribute of the persistence elements. The following snippet outputs the matrix R, one column per line.

```
for i in p:
    for ii in i.cycle:
        print smap[ii],
    print
```

If `p` is an instance of `DynamicPersistenceChains`, its elements have an attribute `chain`, which gives access to columns of matrix V. Thus we can, for example, recover the cycles that never die:

```
for i in p:
    if i.sign() and i.unpaired():
        for ii in i.chain:
            print smap[ii],
        print
```

Remark. Dionysus also provides `ZigzagPersistence` and `CohomologyPersistence` classes with slightly different semantics. We omit them here for lack of space.

3.2 Diagram comparison

It's convenient to record a collection of all persistence pairs (b_i, d_i) in a *persistence diagram*, denoted by $\mathrm{Dgm}_p(f)$. Function `init_diagrams()` returns all such (non-zero) diagrams, each expressed in a class `PersistenceDiagram`.

```
diagrams = init_diagrams(p, filtration)
```

A fundamental property of persistence is its stability [3], which says that if two functions are close, then so are their persistence diagrams:

$$\mathrm{W}_\infty(\mathrm{Dgm}_p(f), \mathrm{Dgm}_p(g)) \leq \|f - g\|_\infty.$$

Here W_∞ denotes the bottleneck distance between the two diagrams. To define it, we compute a bijection $\gamma : \mathrm{Dgm}_p(f) \to \mathrm{Dgm}_p(g)$ that minimizes the longest distance between a point and its image:

$$\mathrm{W}_\infty(\mathrm{Dgm}_p(f), \mathrm{Dgm}_p(g)) = \inf_\gamma \sup_x \|x - \gamma(x)\|_\infty.$$

Rephrased algorithmically, we find the smallest value ε such that the bipartite graph on the two diagrams with an edge for every pair of points closer than ε contains a perfect matching. In Dinoysus, given two diagrams `dgm1` and `dgm2`, we can find their `bottleneck_distance(dgm1, dgm2)`.

Stronger stability results hold for more restrictive classes of functions. Specifically, for Lipschitz functions (with certain conditions on their domains), the persistence diagrams are stable with respect to the Wasserstein distance [4]:

$$\mathrm{W}_q(\mathrm{Dgm}_p(f), \mathrm{Dgm}_p(g)) \leq C \cdot \|f - g\|_\infty^k,$$

where constants C and k depend on the domain of the functions. Here the Wasserstein distance finds the matching γ that minimizes the sum of powers of its edge lengths:

$$\mathrm{W}_q(\mathrm{Dgm}_p(f), \mathrm{Dgm}_p(g)) = \left(\inf_{\gamma} \sum_{x} \|x - \gamma(x)\|_\infty^q \right)^{1/q}.$$

The seemingly small change makes a major difference in practice: W_q is much more sensitive because it accounts for the full diagrams rather than just the longest edge in a matching. In Dionysus, we can find it by calling `wasserstein_distance(dgm1, dgm2, q)`.

4 Code

Derived metric. To illustrate some of the above features, consider the following problem. Given a collection of 3-dimensional point clouds, stored in a list `point_clouds`, we want to find the persistence diagrams of the distance functions they induce on $\mathbb{R}^3$. Treating these diagrams as points, we measure the Wasserstein distances between them. These measurements define a (finite) metric space, and we compute the persistence of its Vietoris–Rips filtration. To put it briefly: what is the homology of the collection of point clouds?

First we compute the persistence diagrams for the alpha shape filtration for every point set in the collection:

```
def alpha_persistence_diagrams(points):
    f = Filtration()
    fill_alpha_complex(points, f)
    f.sort(data_dim_comparison)
    p = StaticPersistence(f)
    p.pair_simplices()
    diagrams = init_diagrams(p, f, lambda s: s.data[0])
    return diagrams

diagrams = [alpha_persistence_diagrams(cloud) for cloud in point_clouds]
```

Armed with the diagrams, we create a `WassersteinDistances` class suitable for the construction of a Vietoris–Rips filtration. We pick arbitrary thresholds `k` and `r`; their real choice is very application-dependent.

```
class WassersteinDistances:
    def __init__(self, diagrams, q, dim = 1):
        self.diagrams  = diagrams
        self.dimension = dim
        self.q         = q

    def __len__(self):
        return len(self.diagrams)

    def __call__(self, i, j):
        dgmi = self.diagrams[i][self.dimension]
        dgmj = self.diagrams[j][self.dimension]
        return wasserstein_distance(dgmi, dgmj, self.q)

distances   = WassersteinDistances(diagrams, q = 3)
rips        = Rips(distances)
simplices   = Filtration()
rips.generate(k, r, simplices.append)      # k-skeleton up to distance r
simplices.sort(rips.cmp)
```

Having generated the Vietoris–Rips filtration `simplices`, we compute its persistence diagrams:

```
p = StaticPersistence(simplices)
p.pair_simplices()
rips_diagrams = init_diagrams(p, simplices, rips.eval)
```

Besides pure curiosity, the knowledge of homology helps us choose candidate target spaces for dimensionality reduction. For example, if we find a 1-dimensional homology class, we can translate it into a map to a circle [5].

Noisy domain. Stability of persistence diagrams helps us cope with the noise in the function values when the domain is fixed. But sometimes even the domain can be noisy. For example, instead of a function $f : X \to \mathbb{R}$ on a perfect compact set X, we may be given its noisy sampling P, where each point has a value $\hat{f}(p)$. With mild assumptions on the function — most importantly, it needs to be Lipschitz — and the quality of the sampling, we can recover the persistence diagram of the implicit original $f : X \to \mathbb{R}$ using *image persistence*. Two papers propose the same procedure [6, 7]: the first using alpha-shapes, the second using Vietoris–Rips complexes.

Denoting the Vietoris–Rips complex $\mathrm{VR}_y(P)$ with P^y, the key idea is that a map on homology from P^α to P^β, for suitably chosen α and β, filters out noisy homology classes while preserving the real cycles of X. (It's exactly the idea behind homology inference [3].) If now we filter these sublevel sets with respect to the function value f and look at the images of the former filtration in the latter, we get an image persistence diagram that provably approximates the persistence diagram of the original function. Denoting by P^y_x the Vietoris–Rips complex for parameter y constructed on the points p with $\hat{f}(p) \leq x$, i.e., $P^y_x = \mathrm{VR}_y(\hat{f}^{-1}(-\infty, x])$, we get a pair of persistence modules:

$$\begin{array}{ccccccccc}
\mathsf{H}(P^\beta_{a_1}) & \to & \mathsf{H}(P^\beta_{a_2}) & \to & \dots & \to & \mathsf{H}(P^\beta_{a_{n-1}}) & \to & \mathsf{H}(P^\beta_{a_n}) \\
\uparrow & & \uparrow & & & & \uparrow & & \uparrow \\
\mathsf{H}(P^\alpha_{a_1}) & \to & \mathsf{H}(P^\alpha_{a_2}) & \to & \dots & \to & \mathsf{H}(P^\alpha_{a_{n-1}}) & \to & \mathsf{H}(P^\alpha_{a_n})
\end{array}$$

The persistence diagram of the image of the lower module in the upper approximates the persistence diagram of f. To express this pair of filtrations in Dionysus, we must filter P^β with respect to the sampled function $\hat{f}$, and indicate when a simplex falls into P^α.

```
# Assume points and values are given as well as hdim, alpha, and beta
distances = PairwiseDistances(points)
rips      = Rips(distances)
simplices = Filtration()
rips.generate(hdim + 1, beta, simplices.append)
simplices.sort(max_vertex_cmp(values))              # Just like a PL-function

img_persistence = ImagePersistence(simplices, lambda s: rips.eval(s) <= alpha)
img_persistence.pair_simplices()
```

The `lambda`-function passed to `ImagePersistence` indicates for every simplex whether it belongs to the subcomplex P^α.

References

[1] Herbert Edelsbrunner and John Harer. *Computational Topology. An Introduction.* American Mathematical Society, Providence, Rhode Island, 2010.

[2] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological Persistence and Simplification. *Discrete and Computational Geometry,* **28**:511–533, 2002.

[3] David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. Stability of Persistence Diagrams. *Discrete and Computational Geometry,* **37**:103–120, 2007.

[4] David Cohen-Steiner, Herbert Edelsbrunner, John Harer, and Yuriy Mileyko. Lipschitz Functions Have L_p-Stable Persistence. *Foundations of Computational Mathematics*, **10**:127–139, 2010.

[5] Vin de Silva, Dmitriy Morozov, Mikael Vejdemo-Johansson. Persistent Cohomology and Circular Coordinates. *Discrete and Computational Geometry*, **45**:737–759, 2011.

[6] David Cohen-Steiner, Herbert Edelsbrunner, John Harer, and Dmitriy Morozov. Persistent Homology for Kernels, Images, and Cokernels. Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, pages 1011–1020, 2009.

[7] Frédéric Chazal, Leonidas J. Guibas, Steve Y. Oudot, Primoz Skraba. Scalar Field Analysis over Point Cloud Data. *Discrete and Computational Geometry*, **46**:743–775, 2011.