A Practical Guide to Persistent Homology

Dmitriy Morozov
Lawrence Berkeley National Lab
A Practical Guide to Persistent Homology

(Dionysus edition)

from dionysus import *
from dionysus.viewer import *
from readers import *

Dmitriy Morozov
Lawrence Berkeley National Lab

Code snippets available at:
http://hg.mrzv.org/Dionysus-tutorial
Dionysus

• C++ library

• Implements various algorithms that I’ve found interesting over the years:

  – ordinary persistence
  – vineyards
  – image persistence
  – zigzag persistence
  – persistent cohomology
  – circular coordinates
  – alpha shapes
  – Vietoris-Rips complexes
  – bottleneck and wasserstein distances between diagrams

• To make life easier, added Python bindings

• This talk exclusively in Python
Python

- Good news: You already know Python! It's just like pseudo-code in your papers, but cleaner. ;-)

- Lists and list comprehensions
  
  ```python
  lst1 = [1,3,5,7,9,11,13]
  lst2 = [i for i in lst1 if i < 9]
  print(lst2)  # [1,3,5,7]
  ```

- Functions
  
  ```python
  def pow(x):
      def f(y):
          return y**x
      return f
  ```

- Loops and conditionals
  
  ```python
  for i in lst1:
      if i % 3 == 0 and i > 5:
          print(square(i))
  ```

- Lots of extra functionality in modules
  
  ```python
  from math import sqrt
  from dionysus import *
  ```
Persistent Homology

- Over a decade old now. Introduced as a way to detect prominent topological features in point clouds. Since then evolved into a rich theory with many applications.

What is the homology of this point cloud?
Persistent Homology

- Over a decade old now. Introduced as a way to detect prominent topological features in point clouds. Since then evolved into a rich theory with many applications.

What is the homology of this point cloud?

- “Squint our eyes”
Persistent Homology

- Over a decade old now. Introduced as a way to detect prominent topological features in point clouds. Since then evolved into a rich theory with many applications.

What is the homology of this point cloud?

- “Squint our eyes”
Persistent Homology

- Over a decade old now. Introduced as a way to detect prominent topological features in point clouds. Since then evolved into a rich theory with many applications.

What is the homology of this point cloud?

- “Squint our eyes”
Persistent Homology

- Over a decade old now. Introduced as a way to detect prominent topological features in point clouds. Since then evolved into a rich theory with many applications.

What is the homology of this point cloud?

- “Squint our eyes”
Persistent Homology

- Over a decade old now. Introduced as a way to detect prominent topological features in point clouds. Since then evolved into a rich theory with many applications.

What is the homology of this point cloud?

- “Squint our eyes”
Persistent Homology

• Over a decade old now. Introduced as a way to detect prominent topological features in point clouds. Since then evolved into a rich theory with many applications.

What is the homology of this point cloud?

• “Squint our eyes” no natural fixed scale → persistent homology
“Eye Squinting”

\[ P \text{ – point set in } \mathbb{R}^n \]

\[ P_r = \bigcup_{p \in P} B_r(p) \]
“Eye Squinting”

$P$ – point set in $\mathbb{R}^n$

$P_r = \bigcup_{p \in P} B_r(p)$
“Eye Squinting”

$P$ – point set in $\mathbb{R}^n$

$$P_r = \bigcup_{p \in P} B_r(p)$$
“Eye Squinting”

$P$ – point set in $\mathbb{R}^n$

$$P_r = \bigcup_{p \in P} B_r(p)$$
“Eye Squinting”

\[ P - \text{point set in } \mathbb{R}^n \]

\[ P_r = \bigcup_{p \in P} B_r(p) \]
“Eye Squinting”

\[ P \text{ – point set in } \mathbb{R}^n \]

\[ P_r = \bigcup_{p \in P} B_r(p) \]

\[ 0 \to H(P_{r_1}) \to H(P_{r_2}) \to \ldots \to H(\mathbb{R}^n) \]
“Eye Squinting”

$P$ – point set in $\mathbb{R}^n$

$$P_r = \bigcup_{p \in P} B_r(p)$$

$$0 \to H(P_{r_1}) \to H(P_{r_2}) \to \ldots \to H(\mathbb{R}^n)$$
“Eye Squinting”

\( P \) – point set in \( \mathbb{R}^n \)

\[
P_r = \bigcup_{p \in P} B_r(p)
\]

\[
0 \rightarrow H(P_{r_1}) \rightarrow H(P_{r_2}) \rightarrow \ldots \rightarrow H(\mathbb{R}^n)
\]
“Eye Squinting”

$P$ – point set in $\mathbb{R}^n$

$P_r = \bigcup_{p \in P} B_r(p)$

Squinting our eyes gives us a continuous function. Algorithms work with (discrete) simplicial complexes.

$0 \to \text{H}(P_{r_1}) \to \text{H}(P_{r_2}) \to \ldots \to \text{H}(\mathbb{R}^n)$
Simplices and Complexes

(Geometric) \( k \)-simplex: convex hull of \((k + 1)\) points.

(Abstract) \( k \)-simplex: subset of \((k + 1)\) elements of a universal set.

Boundary: \( \partial[v_0, \ldots, v_k] = \sum_i (-1)^i [v_0, \ldots, \hat{v}_i, \ldots, v_k] \)

```python
s = Simplex([0,1,2])
print "Dimension:", s.dimension

print "Vertices:"
for v in s.vertices:
    print v

print "Boundary:"
for sb in s.boundary:
    print sb
```

Dimension: 2
Vertices:
0
1
2
Boundary:
<1, 2>
<0, 2>
<0, 1>
Simplices and Complexes

(Geometric) \( k \)-simplex: convex hull of \( (k + 1) \) points.

(Abstract) \( k \)-simplex: subset of \( (k + 1) \) elements of a universal set.

Boundary: \( \partial [v_0, \ldots, v_k] = \sum_i (-1)^i [v_0, \ldots, \hat{v}_i, \ldots, v_k] \)

Simplicial complex: collection of simplices closed under face relation.

not a simplicial complex:
Simplices and Complexes

(Geometric) $k$-simplex: convex hull of $(k + 1)$ points.

(Abstract) $k$-simplex: subset of $(k + 1)$ elements of a universal set.

Boundary: $\partial[v_0, \ldots, v_k] = \sum_i (-1)^i [v_0, \ldots, \hat{v}_i, \ldots, v_k]$

Simplicial complex: collection of simplices closed under face relation.

complex = [Simplex(vertices) for vertices in [[0], [1], [2], [3], [4], [5], [0,1], [0,2], [1,2], [0,1,2], [1,3], [2,4], [3,4]]]
Simplices and Complexes

(Geometric) $k$-simplex: convex hull of $(k + 1)$ points.

(Abstract) $k$-simplex: subset of $(k + 1)$ elements of a universal set.

Boundary: $\partial [v_0, \ldots, v_k] = \sum_i (-1)^i [v_0, \ldots, \hat{v}_i, \ldots, v_k]$

Simplicial complex: collection of simplices closed under face relation.

Complex = [Simplex(vertices) for vertices in
[[0], [1], [2], [3], [4], [5], [0,1], [0,2], [1,2], [0,1,2], [1,3], [2,4], [3,4]]]

simplex9 = Simplex([0,1,2,3,4,5,6,7,8,9])
sphere8 = closure([[simplex9]], 8)
print len(sphere8)
1022
Homology

A $k$-chain is a formal sum of $k$-simplices.

A $k$-cycle is a chain without a boundary.

A $k$-boundary is the boundary of an $(k+1)$-dimensional chain.

$Z$ is the cycle group.

$B$ is the boundary group.

$H = Z/B$ is the homology group.

Two cycles are homologous if they differ by a boundary over $\mathbb{Z}_2$, a set of simplices.

Homology counts cycles up to differences by boundaries.

Diagram:

- A $k$-chain is represented by a formal sum of $k$-simplices.
- A $k$-cycle is a chain without a boundary.
- A $k$-boundary is the boundary of an $(k+1)$-dimensional chain.
- $Z$ is the cycle group.
- $B$ is the boundary group.
- $H = Z/B$ is the homology group.
- Two cycles are homologous if they differ by a boundary over $\mathbb{Z}_2$.
Homology in Dionysus

Dionysus doesn't compute homology directly, but we can get it as a by-product of persistent homology.

```python
complex = sphere8

f = Filtration(complex, dim_cmp)
p = StaticPersistence(f)
p.pair_simplices()

dgms = init_diagrams(p,f, lambda s: 0)

for i, dgm in enumerate(dgms):
    print "Dimension:", i
    print dgm
```
Persistent Homology (pipeline)

Filtration of a simplicial complex:

\[ K_1 \subseteq K_2 \subseteq \ldots \subseteq K_n \]

(w.l.o.g. assume \( K_{i+1} = K_i + \sigma_i \)).

so, really, an ordering of simplices
Persistent Homology (pipeline)

Filtration of a simplicial complex:

\[ K_1 \subseteq K_2 \subseteq \ldots \subseteq K_n \]

(w.l.o.g. assume \( K_{i+1} = K_i + \sigma_i \)).

so, really, an ordering of simplices

simplices = [([0], 1), ([1], 2), ([0,1], 3), ([2], 4), ([1,2], 5), ([0,2], 6)]

f = Filtration()
for vertices, time in simplices:
    f.append(Simplex(vertices, time))
f.sort(dim_data_CMP)
for s in f:
    print s, s.data    # s.data is the time
Persistent Homology (pipeline)

Filtration of a simplicial complex:

\[ K_1 \subseteq K_2 \subseteq \ldots \subseteq K_n \]

(w.l.o.g. assume \( K_{i+1} = K_i + \sigma_i \)).

so, really, an ordering of simplices

\[ H(K_1) \to H(K_2) \to \ldots \to H(K_n) \]
Persistent Homology (pipeline)

```python
p = StaticPersistence(f)
p.pair_simplices()
dgms = init_diagrams(p, f)
for i, dgm in enumerate(dgms):
    print "Dimension:" , i
    print dgm
```

\[ H_0 : \]

\[ H_1 : \]

\[ H(K_1) \rightarrow H(K_2) \rightarrow \ldots \rightarrow H(K_n) \]
Filtrations: $\alpha$-shapes

$K_r = \text{Nrv}\{B_r(u) \cap \text{Vor } u\}$

$K_r \simeq \bigcup_{p \in P} B_r(p)$

$K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots$
Filtrations: $\alpha$-shapes

$K_r = Nrv\{B_r(u) \cap \text{Vor } u\}$

$K_r \simeq \bigcup_{p \in P} B_r(p)$

$K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots$
Filtrations: $\alpha$-shapes

$K_r = \text{Nrv}\{B_r(u) \cap \text{Vor } u\}$

$K_r \simeq \bigcup_{p \in P} B_r(p)$

$K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots$
Filtrations: $\alpha$-shapes

$K_r = \text{Nrv}\{B_r(u) \cap \text{Vor } u\}$

$K_r \simeq \bigcup_{p \in P} B_r(p)$

$K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots$
Filtrations: $\alpha$-shapes

$K_r = \operatorname{Nrv}\{B_r(u) \cap \operatorname{Vor} u\}$

$K_r \simeq \bigcup_{p \in P} B_r(p)$

$K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots$
Filtrations: α-shapes

\[ K_r = \text{Nrv}\{B_r(u) \cap \text{Vor } u\} \]

\[ K_r \simeq \bigcup_{p \in P} B_r(p) \]

\[ K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots \]
Filtrations: $\alpha$-shapes

$K_r = Nrv\{B_r(u) \cap Vor \ u\}$

$K_r \simeq \bigcup_{p \in P} B_r(p)$

$K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots$
Filtrations: $\alpha$-shapes

$K_r = \text{Nrv}\{B_r(u) \cap \text{Vor } u\}$

$K_r \simeq \bigcup_{p \in P} B_r(p)$

$K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots$

$r_\sigma = \min_{x \in \text{Vor } \sigma} d_P(x)$
Filtrations: $\alpha$-shapes

\[ K_r = \text{Nrv}\{B_r(u) \cap \text{Vor } u\} \]
\[ K_r \simeq \bigcup_{p \in P} B_r(p) \]
\[ K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots \]
\[ r_\sigma = \min_{x \in \text{Vor } \sigma} d_P(x) \]

from math import sqrt
points = read_points('data/trefoil.pts')
f = Filtration()
fill_alpha_complex(points, f)
show_complex(points, [s for s in f if sqrt(s.data[0]) < 1])

Fills $f$ with all the simplices of the Delaunay triangulation (thanks to CGAL's Delaunay package).

The data field of each simplex is set to a pair $(r_\sigma^2, \sigma \cap \text{Vor } \sigma \neq \emptyset)$. 
Filtrations: $\alpha$-shapes

\[ K_r = \text{Nrv}\{B_r(u) \cap \text{Vor} u\} \]
\[ K_r \simeq \bigcup_{p \in P} B_r(p) \]
\[ K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots \]
\[ r_\sigma = \min_{x \in \text{Vor} \sigma} d_P(x) \]

```python
from math import sqrt
points = read_points('data/trefoil.pts')
f = Filtration()
fill_alpha_complex(points, f)
show_complex(points, [s for s in f if sqrt(s.data[0]) < 1])
```

Fills $f$ with all the simplices of the Delaunay triangulation (thanks to CGAL’s Delaunay package).

The data field of each simplex is set to a pair \((r_\sigma^2, \sigma \cap \text{Vor} \sigma \neq \emptyset)\).
Filtrations: $\alpha$-shapes

\[ K_r = \text{Nrv}\{B_r(u) \cap \text{Vor } u\} \]

\[ K_{r_1} \subseteq K_{r_2} \subseteq \ldots \subseteq K_{r_\sigma} \subseteq \ldots \]

\[ r_\sigma = \min_{x \in \text{Vor } \sigma} d_P(x) \]

```python
from math import sqrt
points = read_points('data/trefoil.pts')
f = Filtration()
fill_alpha_complex(points, f)
show_complex(points, [s for s in f if sqrt(s.data[0]) < 1])

f.sort(dim_data_cmp)
p = StaticPersistence(f)
p.pair_simplices()

dgms = init_diagrams(p, f, lambda s: sqrt(s.data[0]))
show_diagram(dgms)
```
Filtrations: Vietoris-Rips

\[ \text{VR}(r) = \{ \sigma \subseteq P \mid |u - v| < r \ \forall \ u, v \in \sigma \} \]

(clique complex of r-nearest neighbor graph)

NB: only pairwise distances matter
Filtrations: Vietoris-Rips

\[ VR(r) = \{ \sigma \subseteq P \mid |u - v| < r \ \forall \ u, v \in \sigma \} \]

(clique complex of r-nearest neighbor graph)

NB: only pairwise distances matter

```python
points = read_points('data/trefoil.pts')
distances = PairwiseDistances(points)
distances = ExplicitDistances(distances)
rips = Rips(distances)
f = Filtration()
rips.generate(2, 1.7, f.append)
print "Number of simplices: ", len(f)

show_complex(points, f)
show_complex(points, [s for s in f if rips.eval(s) < 1.6])
```
Filtrations: Vietoris-Rips

\[ VR(r) = \{ \sigma \subseteq P \mid |u - v| < r \quad \forall \ u, v \in \sigma \} \]

(clique complex of r-nearest neighbor graph)

NB: only pairwise distances matter

```
points = read_points('data/trefoil.pts')
distances = PairwiseDistances(points)
distances = ExplicitDistances(distances)
rips = Rips(distances)
f = Filtration()
rips.generate(2, 1.7, f.append)
print "Number of simplices:", len(f)

show_complex(points, f)
show_complex(points, [s for s in f if rips.eval(s) < 1.6])
```
Filtrations: Vietoris-Rips

\[ VR(r) = \{ \sigma \subseteq P \mid |u - v| < r \ \forall \ u, v \in \sigma \} \]

(clique complex of r-nearest neighbor graph)

NB: only pairwise distances matter

```python
points = read_points('data/trefoil.pts')
distances = PairwiseDistances(points)
distances = ExplicitDistances(distances)
rips = Rips(distances)
f = Filtration()
rips.generate(2, 1.7, f.append)
print "Number of simplices: ", len(f)

show_complex(points, f)
show_complex(points, [s for s in f if rips.eval(s) < 1.6])

f.sort(rips.cmp)
p = StaticPersistence(f)
p.pair_simplices()

dgms = init_diagrams(p, f, rips.eval)
show_diagram(dgms[:2])
```

06-rips.py
Filtrations: Lower-Star

\[ \hat{f} : \text{Vrt } K \rightarrow \mathbb{R} \]

\[ f : |K| \rightarrow \mathbb{R} \quad \text{linearly interpolated} \]

\[ |K|_a = f^{-1}(-\infty, a] \]

Interested in the filtration:

\[ |K|_{a_1} \subseteq |K|_{a_2} \subseteq \ldots \subseteq |K|_{a_n} \]
Filtrations: Lower-Star

\[ \hat{f} : \text{Vrt } K \rightarrow \mathbb{R} \]
\[ f : |K| \rightarrow \mathbb{R} \quad \text{linearly interpolated} \]

\[ |K|_a = f^{-1}(-\infty, a] \]

Interested in the filtration:

\[ |K|_{a_1} \subseteq |K|_{a_2} \subseteq \ldots \subseteq |K|_{a_n} \]

\[ K_a = \{ \sigma \in K \mid \max_{v \in \sigma} \hat{f}(v) \leq a \} \]

(changes only as \( a \) passes vertex values)

\[ |K|_a \simeq K_a \]

So, instead, we can compute:

\[ K_{a_1} \subseteq K_{a_2} \subseteq \ldots \subseteq K_{a_n} \]
Filtrations: Lower-Star

\[ \hat{f} : \text{Vrt } K \rightarrow \mathbb{R} \]
\[ f : |K| \rightarrow \mathbb{R} \quad \text{linearly interpolated} \]
\[ |K|_a = f^{-1}(-\infty, a] \]

Interested in the filtration:

\[ |K|_{a_1} \subseteq |K|_{a_2} \subseteq \ldots \subseteq |K|_{a_n} \]

\[ K_a = \{ \sigma \in K \mid \max_{v \in \sigma} \hat{f}(v) \leq a \} \]

(changes only as \( a \) passes vertex values)

\[ |K|_a \simeq K_a \]

So, instead, we can compute:

\[ K_{a_1} \subseteq K_{a_2} \subseteq \ldots \subseteq K_{a_n} \]
Filtrations: Lower-Star

\[ \hat{f} : \text{Vrt } K \rightarrow \mathbb{R} \]
\[ f : |K| \rightarrow \mathbb{R} \quad \text{linearly interpolated} \]
\[ |K|_a = f^{-1}(-\infty, a] \]

Interested in the filtration:

\[ |K|_{a_1} \subseteq |K|_{a_2} \subseteq \ldots \subseteq |K|_{a_n} \]

\[ K_a = \{ \sigma \in K \mid \max_{v \in \sigma} \hat{f}(v) \leq a \} \]

(changes only as \( a \) passes vertex values)

\[ |K|_a \simeq K_a \]

So, instead, we can compute:

\[ K_{a_1} \subseteq K_{a_2} \subseteq \ldots \subseteq K_{a_n} \]
Filtrations: Lower-Star

elephant_points, elephant_complex = read_off('data/cgal/elephant.off')
elephant_complex = closure(elephant_complex, 2)
show_complex(elephant_points, elephant_complex)

def pojection(points, axis = 1):  # projection onto a coordinate axis
    def value(v):
        return points[v][axis]
    return value

value = projection(elephant_points, 1)
Filtrations: Lower-Star

elephant_points, elephant_complex = read_off('data/cgal/elephant.off')
elephant_complex = closure(elephant_complex, 2)
show_complex(elephant_points, elephant_complex)

def pojection(points, axis = 1): # projection onto a coordinate axis
def value(v):
    return points[v][axis]
return value
value = projection(elephant_points, 1)

def max_vertex_compare(value):
def max_vertex(s):
    return max(value(v) for v in s.vertices)
def compare(s1, s2):
    return cmp(s1.dimension(), s2.dimension()) or \
    cmp(max_vertex(s1), max_vertex(s2))
return compare

f = Filtration(elephant_complex, max_vertex_compare(value))
Filtrations: Lower-Star

elephant_points, elephant_complex = read_off('data/cgal/elephant.off')
elephant_complex = closure(elephant_complex, 2)
show_complex(elephant_points, elephant_complex)

def pojection(points, axis = 1):  # projection onto a coordinate axis
def value(v):
    return points[v][axis]
return value
value = projection(elephant_points, 1)

def max_vertex_compare(value):
def max_vertex(s):
    return max(value(v) for v in s.vertices)
def compare(s1, s2):
    return cmp(s1.dimension(), s2.dimension()) or \
        cmp(max_vertex(s1), max_vertex(s2))
return compare

f = Filtration(elephant_complex, max_vertex_compare(value))
p = DynamicPersistenceChains(f)
p.pair_simplices()
dgms = init_diagrams(p, f, lambda s: max(value(v) for v in s.vertices))
show_diagrams(dgms)
Extended Persistence

Extended persistence was introduced as a way to measure the essential persistence classes:

\[
\begin{align*}
H(X_{a_1}) & \rightarrow H(X_{a_2}) & \rightarrow & \ldots & \rightarrow & H(X_{a_n}) & \rightarrow & H(X) \\
H(X, X_{a_1}) & \leftarrow H(X, X_{a_2}) & \leftarrow & \ldots & \leftarrow & H(X, X_{a_n}) & \leftarrow & H(X, \emptyset)
\end{align*}
\]
Extended Persistence

Extended persistence was introduced as a way to measure the essential persistence classes:

\[ \begin{align*}
H(X_{a_1}) & \to H(X_{a_2}) \to \ldots \to H(X_{a_n}) \to H(X) \\
H(X, X^{a_1}) & \leftarrow H(X, X^{a_2}) \leftarrow \ldots \leftarrow H(X, X^{a_n}) \leftarrow H(X, \emptyset)
\end{align*} \]

\[ H(X, Y) \simeq H(X \cup w \ast Y, w) \]
Persistent Homology

Filtration $\rightarrow$ $D$, ordered boundary matrix (indexed by simplices)
$D[i, j] = \text{index of } \sigma_i \text{ in boundary of } \sigma_j$

Persistence $\rightarrow$ Decomposition $R = DV$, where $R$ is reduced, meaning lowest ones are in unique rows, and $V$ is upper-triangular.

$$R = D \cdot V$$
Persistent Homology

Filtration $\rightarrow$ $D$, ordered boundary matrix (indexed by simplices)

$D[i, j] = \text{index of } \sigma_i \text{ in boundary of } \sigma_j$

Persistence $\rightarrow$ Decomposition $R = DV$, where $R$ is reduced, meaning lowest ones are in unique rows, and $V$ is upper-triangular.

\[
\begin{array}{c|c|c}
\sigma & \begin{array}{c}
R
\end{array} & \begin{array}{c}
\sigma
\end{array} \\
- & 0 & - \\
\end{array}
\begin{array}{c|c|c}
\begin{array}{c}
D
\end{array} & \begin{array}{c}
= \\
\uparrow
\end{array} & \begin{array}{c}
V
\end{array}
\end{array}
\]
**Persistent Homology**

**Filtration** → $D$, ordered boundary matrix (indexed by simplices)

$D[i, j] = \text{index of } \sigma_i \text{ in boundary of } \sigma_j$

**Persistence** → Decomposition $R = DV$, where $R$ is reduced, meaning lowest ones are in unique rows, and $V$ is upper-triangular.

\[
\begin{array}{ccc}
\sigma & \tau \\
\sigma & & \\
0 & & \\
\end{array}
\]

\[
\begin{array}{ccc}
\sigma & \tau \\
\sigma & & \\
0 & & \\
\end{array}
\]

\[
R = D V
\]

\[
\sigma & \tau \\
\sigma & & \\
0 & & \\
\end{array}
\]

\[
\begin{array}{ccc}
\sigma & \tau \\
\sigma & & \\
0 & & \\
\end{array}
\]

\[
D V
\]

\[
\sigma & \tau \\
\sigma & & \\
0 & & \\
\end{array}
\]

\[
\begin{array}{ccc}
\sigma & \tau \\
\sigma & & \\
0 & & \\
\end{array}
\]

\[
D V
\]
**Persistent Homology**

Filtration  $\rightarrow$  $D$, ordered boundary matrix (indexed by simplices)

$D[i, j] =$ index of $\sigma_i$ in boundary of $\sigma_j$

Persistence  $\rightarrow$  Decomposition $R = DV$, where $R$ is reduced, meaning lowest ones are in unique rows, and $V$ is upper-triangular.

StaticPersistence computes just $R$, enough for the pairing. Iterating over StaticPersistence, we can access columns of $R$, through cycle attribute. (Also pair(), sign(), unpaired().)

```
smap = p.make_simplex_map(f)
for i in p:
    if not i.sign():
        print [smap[j] for j in i.cycle]
```
Persistent Homology

Filtration $\rightarrow$ $D$, ordered boundary matrix (indexed by simplices)

$D[i, j] =$ index of $\sigma_i$ in boundary of $\sigma_j$

Persistence $\rightarrow$ Decomposition $R = DV$, where $R$ is reduced, meaning lowest ones are in unique rows, and $V$ is upper-triangular.

StaticPersistence computes just $R$, enough for the pairing.
Iterating over StaticPersistence, we can access columns of $R$, through cycle attribute. (Also pair(), sign(), unpaired().)

DynamicPersistenceChains computes matrices $R$ and $V$.
Access columns of $V$ through chain. (E.g., gives access to the infinitely persistent classes.)

```
smap = p.make_simplex_map(f)
for i in p:
    if not i.sign():
        print [smap[j] for j in i.cycle]
```
Persistent Homology

Filtration $\rightarrow D$, ordered boundary matrix (indexed by simplices)
$D[i, j] = \text{index of } \sigma_i \text{ in boundary of } \sigma_j$

Persistence $\rightarrow$ Decomposition $R = DV$, where $R$ is reduced, meaning lowest ones are in unique rows, and $V$ is upper-triangular.

```python
while True:
    pt = show_diagram(dgms)
    if not pt: break
    print pt
    i = pt[2]
    smap = persistence.make_simplex_map(f)
    chain = [smap[ii] for ii in i.chain]
    pair_cycle = [smap[ii] for ii in i.pair().cycle]
    pair_chain = [smap[ii] for ii in i.pair().chain]
    show_complex(elephant_points, subcomplex = chain)
    show_complex(elephant_points, subcomplex = pair_cycle + pair_chain)
execlfile('08-cycle-chain.py')
```
Diagrams, Stability, and Distances

$Dgm(f)$
Diagrams, Stability, and Distances

Bottleneck distance:

\[ W_\infty(Dgm(f), Dgm(g)) = \inf_{\gamma} \|x - \gamma(x)\|_\infty \]
Diagrams, Stability, and Distances

Bottleneck distance:

\[ W_\infty(Dgm(f), Dgm(g)) = \inf_\gamma \| x - \gamma(x) \|_\infty \]
Bottleneck distance:

\[ W_\infty(Dgm(f), Dgm(g)) = \inf_{\gamma} \| x - \gamma(x) \|_\infty \]

\[
\text{bottleneck_distance}(\text{dgm1, dgm2})
\]
Bottleneck distance:

\[ W_\infty(Dgm(f), Dgm(g)) = \inf_\gamma \| x - \gamma(x) \|_\infty \]

Stability Theorem:

\[ W_\infty(Dgm(f), Dgm(g)) \leq \| f - g \|_\infty \]
Diagrams, Stability, and Distances

Bottleneck distance:

\[
W_\infty(Dgm(f), Dgm(g)) = \inf_{\gamma} \|x - \gamma(x)\|_\infty
\]

Stability Theorem:

\[
W_\infty(Dgm(f), Dgm(g)) \leq \|f - g\|_\infty
\]

Wasserstein distance:

\[
W^q_\infty(Dgm(f), Dgm(g)) = \inf_{\gamma} \sum \|x - \gamma(x)\|_\infty^q
\]

Wasserstein Stability Theorem: For Lipschitz functions \(f\) and \(g\), under some technical conditions on the domain,

\[
W^q_\infty(Dgm(f), Dgm(g)) \leq C \cdot \|f - g\|_\infty^k
\]
Circle-Valued Coordinates

- How to get a tangible feel for the topological features that we find?
Circle-Valued Coordinates

- How to get a tangible feel for the topological features that we find?

\[ H^1(X; \mathbb{Z}) \cong [X, S^1] \]

- Maps into circles, natural for:
  - Phase coordinates for waves
  - Angle coordinates for directions
  - Periodic data
Circle-Valued Coordinates

- How to get a tangible feel for the topological features that we find?

\[ H^1(X; \mathbb{Z}) \cong [X, S^1] \]

Start with the canonical isomorphism between 1-dimensional cohomology classes and homotopy classes of maps into a circle.

- Maps into circles, natural for:
  - Phase coordinates for waves
  - Angle coordinates for directions
  - Periodic data

Algorithm:

1. Compute **persistent cohomology** classes
2. Turn each representative cocycle \( z^* \) into a map, \( X \rightarrow S^1 \)
3. Smooth that map (minimize variation across edges), staying within the same cohomology/homotopy class (equivalently, find the harmonic cocycle)
Circle-Valued Coordinates

- How to get a tangible feel for the topological features that we find?

\[ H^1(X; \mathbb{Z}) \cong [X, S^1] \]

Start with the canonical isomorphism between 1-dimensional cohomology classes and homotopy classes of maps into a circle.

- Maps into circles, natural for:
  - Phase coordinates for waves
  - Angle coordinates for directions
  - Periodic data

Algorithm:

1. Compute **persistent cohomology** classes
2. Turn each representative cocycle \( z^* \) into a map, \( X \to S^1 \)
3. Smooth that map (minimize variation across edges), staying within the same cohomology/homotopy class (equivalently, find the harmonic cocycle)
Circle-Valued Coordinates

- How to get a tangible feel for the topological features that we find?

\[ H^1(X; \mathbb{Z}) \cong [X, S^1] \]

Start with the canonical isomorphism between 1-dimensional cohomology classes and homotopy classes of maps into a circle.

- Maps into circles, natural for:
  - Phase coordinates for waves
  - Angle coordinates for directions
  - Periodic data

Algorithm:

1. Compute **persistent cohomology** classes

2. Turn each representative cocycle \( z^* \) into a map, \( X \to S^1 \)

3. Smooth that map (minimize variation across edges), staying within the same cohomology/homotopy class (equivalently, find the harmonic cocycle)

Vertices map to 0; edges wind with the degree given by \( z^*(e) \).
Circle-Valued Coordinates

- How to get a tangible feel for the topological features that we find?

\[ H^1(X; \mathbb{Z}) \cong [X, S^1] \]

Start with the canonical isomorphism between 1-dimensional cohomology classes and homotopy classes of maps into a circle.

- Maps into circles, natural for:
  - Phase coordinates for waves
  - Angle coordinates for directions
  - Periodic data

Algorithm:

1. Compute **persistent cohomology** classes
2. Turn each representative cocycle \( z^* \) into a map, \( X \to S^1 \)
3. Smooth that map (minimize variation across edges), staying within the same cohomology/homotopy class (equivalently, find the harmonic cocycle)
points = read_points('data/annulus.pts')
execfile('10-circular.py')

from math import sqrt

f = Filtration()
fill_alpha_complex(points, f)
f.sort(dim_data_cmp)

p = StaticCohomologyPersistence(f, prime = 11)
p.pair_simplices()
dgms = init_diagrams(p,f, lambda s: sqrt(s.data[0]), lambda n: n.cocycle)

while True:
    pt = show_diagram(dgms)
    if not pt: break
    rf = Filtration((s for s in f if sqrt(s.data[0]) <= (pt[0] + pt[1])/2))
    values = circular.smooth(rf, pt[2])
    cocycle = [rf[i] for (c,i) in pt[2] if i < len(rf)]
    show_complex(points, subcomplex = cocycle)
    show_complex(points, values = values)
points = read_points('data/annulus.pts')
eexecfile('10-circular.py')

from math import sqrt

f = Filtration()
fill_alpha_complex(points, f)
f.sort(dim_data_cmp)

p = StaticCohomologyPersistence(f, prime = 11)
p.pair_simplices()
dgms = init_diagrams(p,f, lambda s: sqrt(s.data[0]), lambda n: n.cocycle)

while True:
    pt = show_diagram(dgms)
    if not pt: break
    rf = Filtration((s for s in f if sqrt(s.data[0]) <= (pt[0] + pt[1])/2))
    values = circular.smooth(rf, pt[2])
    cocycle = [rf[i] for (c,i) in pt[2] if i < len(rf)]
    show_complex(points, subcomplex = cocycle)
    show_complex(points, values = values)
Image Persistence

Noisy domains: instead of $f : \mathbb{X} \rightarrow \mathbb{R}$, we have a function $\tilde{f} : P \rightarrow \mathbb{R}$

$P$ a sample of $\mathbb{X}$

For suitably-chosen parameters $\alpha$ and $\beta$:

\[
\begin{align*}
H(K_{\beta}^{a_1}) & \rightarrow H(K_{\beta}^{a_2}) \rightarrow \ldots \rightarrow H(K_{\beta}^{a_n}) \\
\uparrow & \uparrow \uparrow \\
H(K_{\alpha}^{a_1}) & \rightarrow H(K_{\alpha}^{a_2}) \rightarrow \ldots \rightarrow H(K_{\alpha}^{a_n})
\end{align*}
\]

$K_{\alpha}^{a} = \alpha$ shape or Vietoris-Rips complex with parameter $\alpha$ built on $\tilde{f}^{-1}(-\infty, a]$
Image Persistence

Noisy domains: instead of \( f : X \to \mathbb{R} \), we have a function \( \tilde{f} : P \to \mathbb{R} \)

\( P \) a sample of \( X \)

For suitably-chosen parameters \( \alpha \) and \( \beta \):

\[
\begin{align*}
H(K_{\beta}^{a_1}) & \to H(K_{\beta}^{a_2}) \to \ldots \to H(K_{\beta}^{a_n}) \\
\uparrow & \quad \uparrow \quad \uparrow \\
H(K_{\alpha}^{a_1}) & \to H(K_{\alpha}^{a_2}) \to \ldots \to H(K_{\alpha}^{a_n}) \\
\end{align*}
\]

\( K_{\alpha}^{a} = \) alpha shape or Vietoris-Rips complex with parameter \( \alpha \) built on \( \tilde{f}^{-1}(-\infty, a] \)

# assume parallel lists points and values
f = Filtration()
f = fill_alpha_complex(points, f)
# use persistence of f to choose alpha and beta chosen
f = Filtration([s for s in f if sqrt(s.data[0]) <= beta])
f.sort(max_vertex_compare(values))
p = ImagePersistence(f, lambda s: sqrt(s.data[0]) <= alpha)
p.pair_simplices()

dgms = init_diagrams(p, f, lambda s: max(values(v) for v in s.vertices))
show_diagrams(dgms)
Conclusions

- Persistence is easy to use. Dionysus can help you try out new ideas.
Conclusions

- Persistence is easy to use. Dionysus can help you try out new ideas.
- Practice reinforces theory. For example, persistent cohomology algorithm, in practice, is the fastest way I know to compute persistence diagrams. (This realization is a pure accident of experimental work with circular coordinates.) Studying why this is the case has lead to “Dualities in Persistent (Co)Homology.”
Conclusions

• Persistence is easy to use. Dionysus can help you try out new ideas.

• Practice reinforces theory. For example, persistent cohomology algorithm, in practice, is the fastest way I know to compute persistence diagrams. (This realization is a pure accident of experimental work with circular coordinates.) Studying why this is the case has lead to “Dualities in Persistent (Co)Homology.”

• Python bindings were one of the best decisions. (Hint, hint, CGAL.) However, sometimes much slower than the C++ counter-parts. A lot of the common functionality is available as examples in C++; don’t overlook them.
Conclusions

- Persistence is easy to use. Dionysus can help you try out new ideas.

- Practice reinforces theory. For example, persistent cohomology algorithm, in practice, is the fastest way I know to compute persistence diagrams. (This realization is a pure accident of experimental work with circular coordinates.) Studying why this is the case has lead to “Dualities in Persistent (Co)Homology.”

- Python bindings were one of the best decisions. (Hint, hint, CGAL.) However, sometimes much slower than the C++ counter-parts. A lot of the common functionality is available as examples in C++; don’t overlook them.

- Dionysus includes significant chunks of open-source code by the following people (many thanks to them):
  - Jeffrey Kline (LSQR port to Python)
  - Bernd Gaertner (implementation of Miniball algorithm used for Čech complexes)
  - John Weaver (Hungarian algorithm used for Wasserstein distances)
  - Arne Schmitz (PyGLWidget.py)
Thank you for your time and attention!