Geometric Sciences in Action
May, 27th-31st 2024
Centre International de Rencontres Mathématiques

Topological Data Analysis with the Gudhi library

Mathieu Carrière
DataShape team
Centre Inria d'Université Côte d'Azur
mathieu.carriere@inria.fr

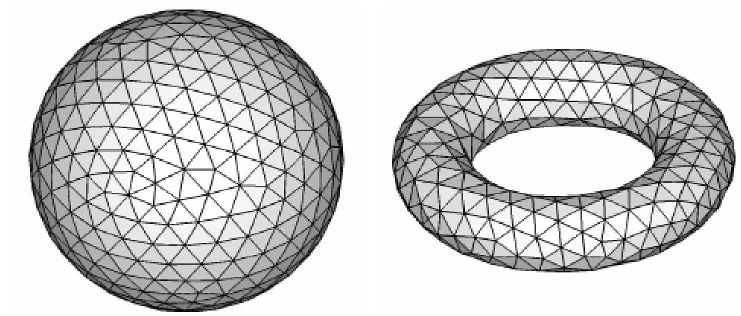
Inria informatics mathematics

What is Topological Data Analysis?

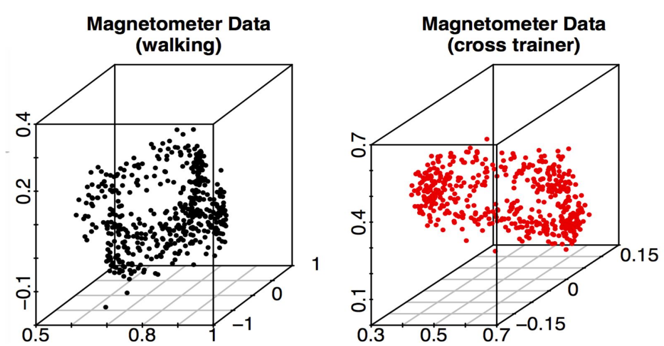
Topological Data Analysis is:

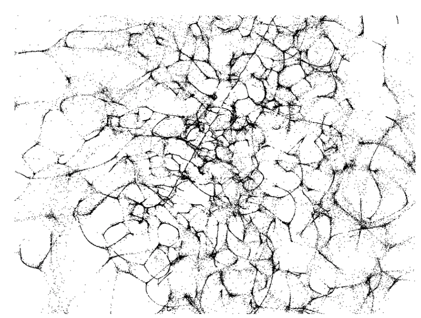
a mathematically grounded framework...

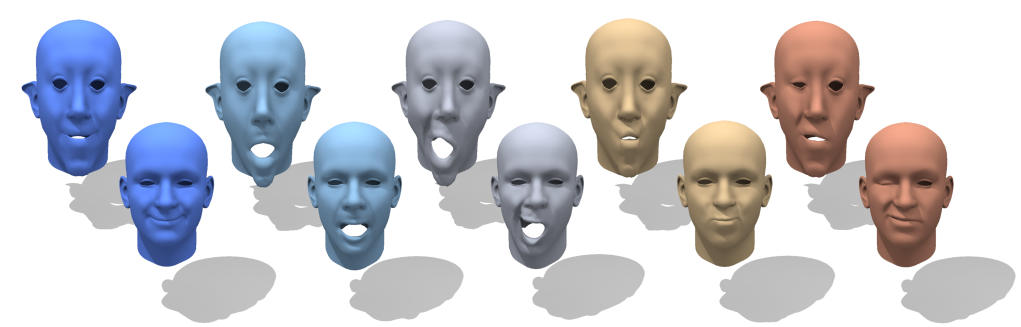
$$H_k = Z_k / B_k$$



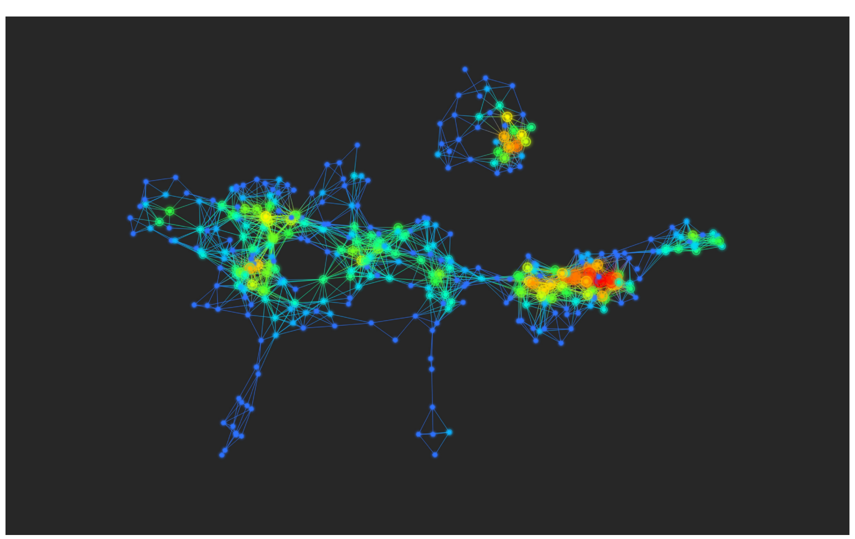
...that applies to a wide variety of data sets...



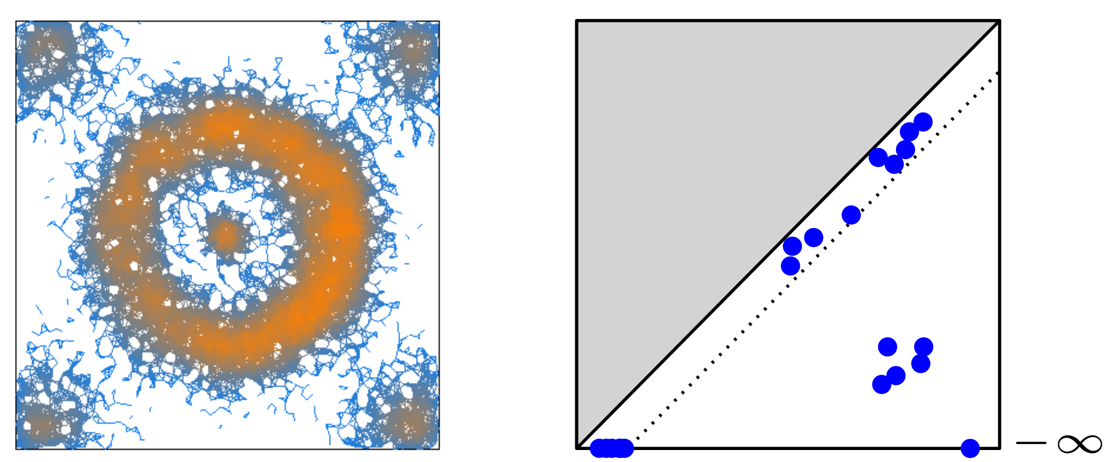




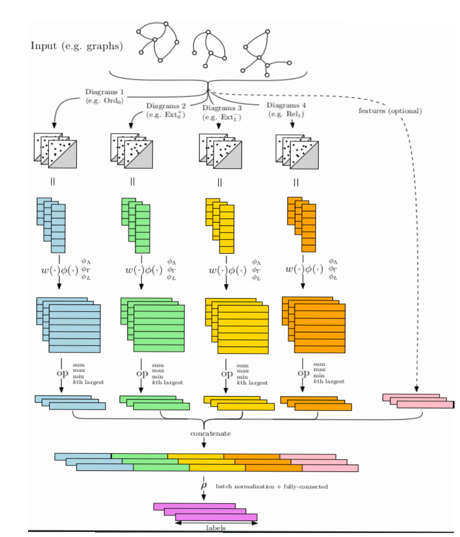
...for a wide variety of tasks.



Mapper: exploratory data analysis



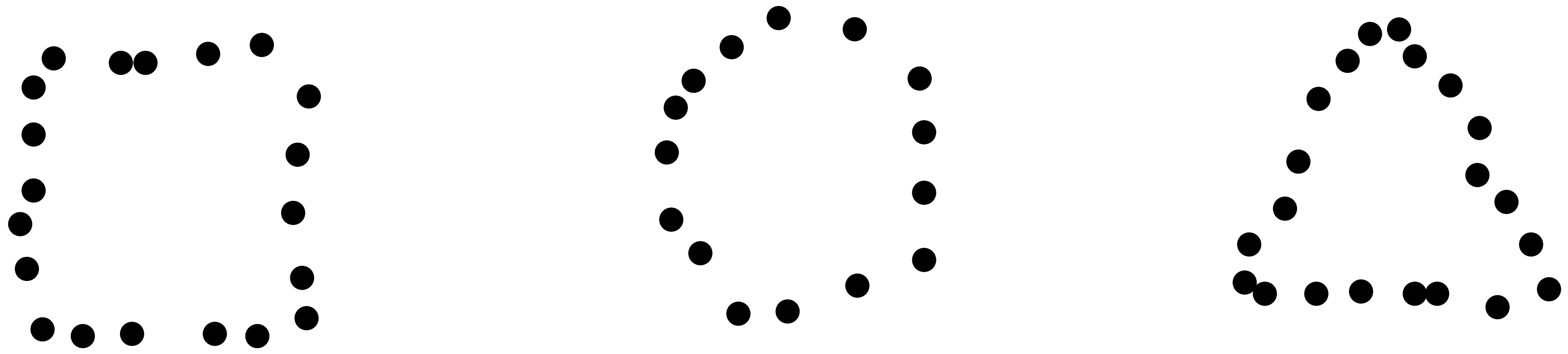
ToMATo: clustering



Persistence diagrams: machine learning

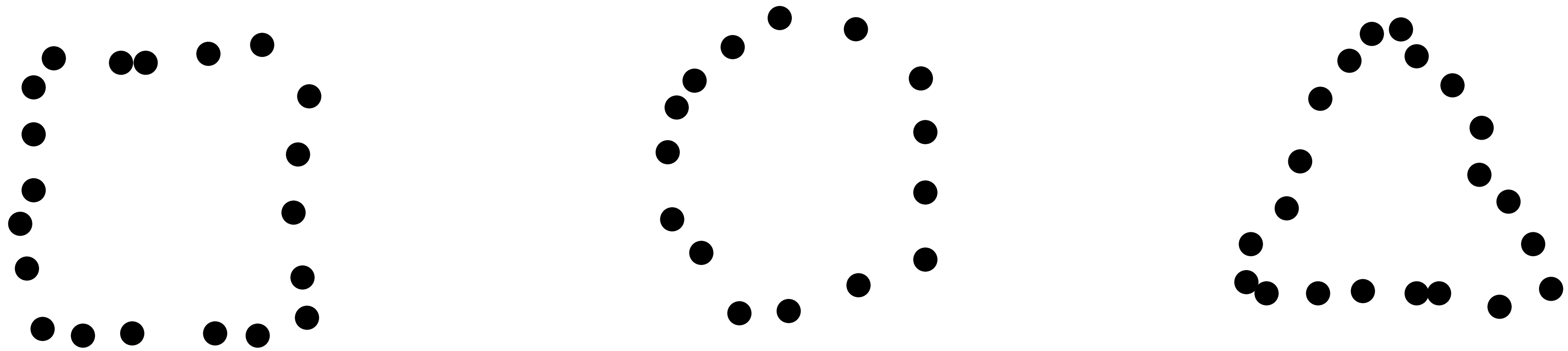
What is Topological Data Analysis?

Its main goal is to compute the topology of spaces (number and sizes of connected components, loops, cavities, etc) from samplings...



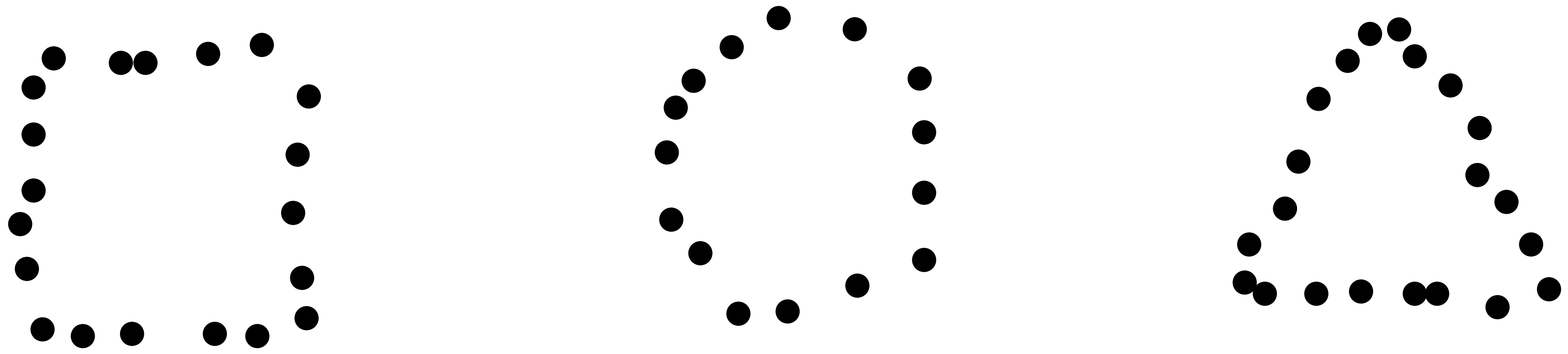
What is Topological Data Analysis?

Its main goal is to compute the topology of spaces (number and sizes of connected components, loops, cavities, etc) from samplings...



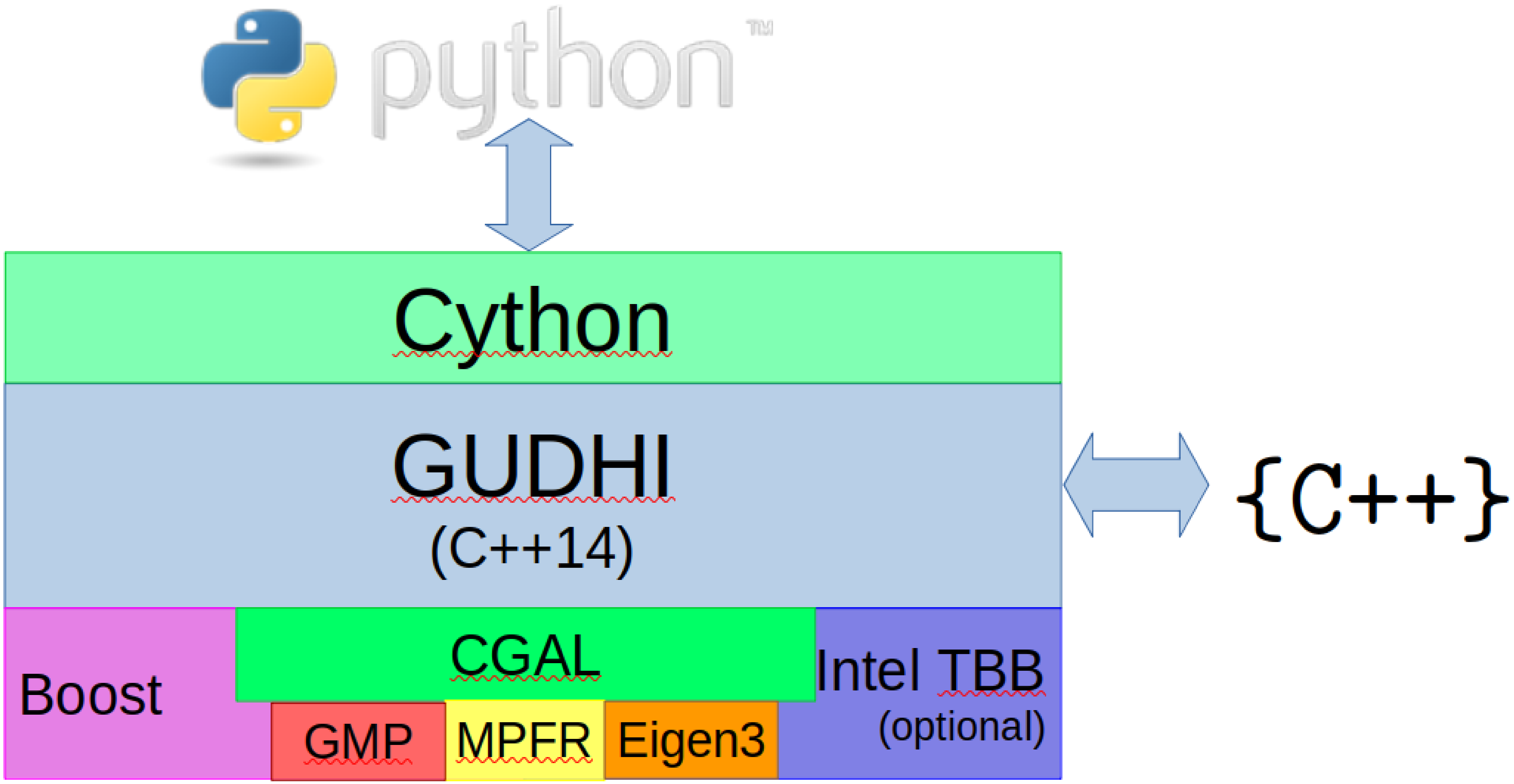
What is Topological Data Analysis?

Its main goal is to compute the topology of spaces (number and sizes of connected components, loops, cavities, etc) from samplings...

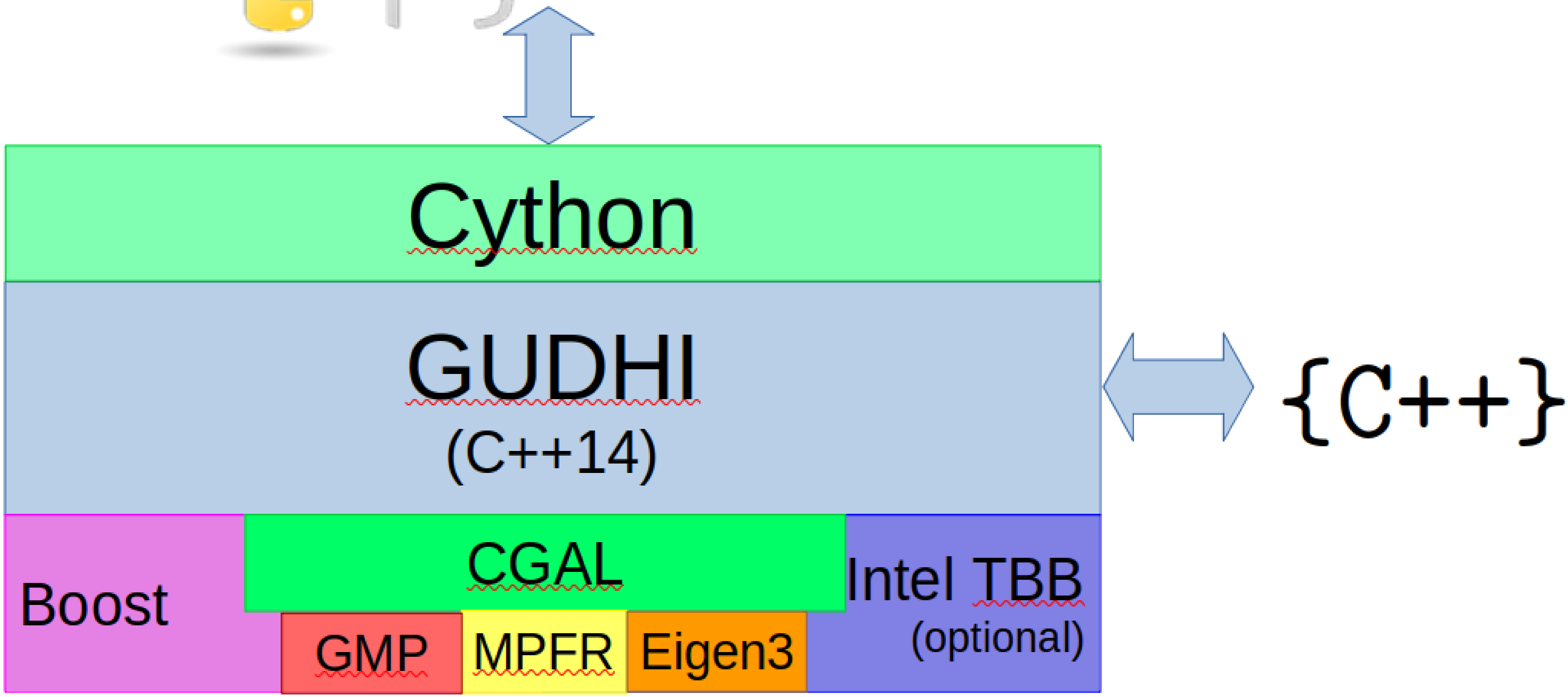


...and to summarize this info into descriptors suitable for data science

The Gudhi library

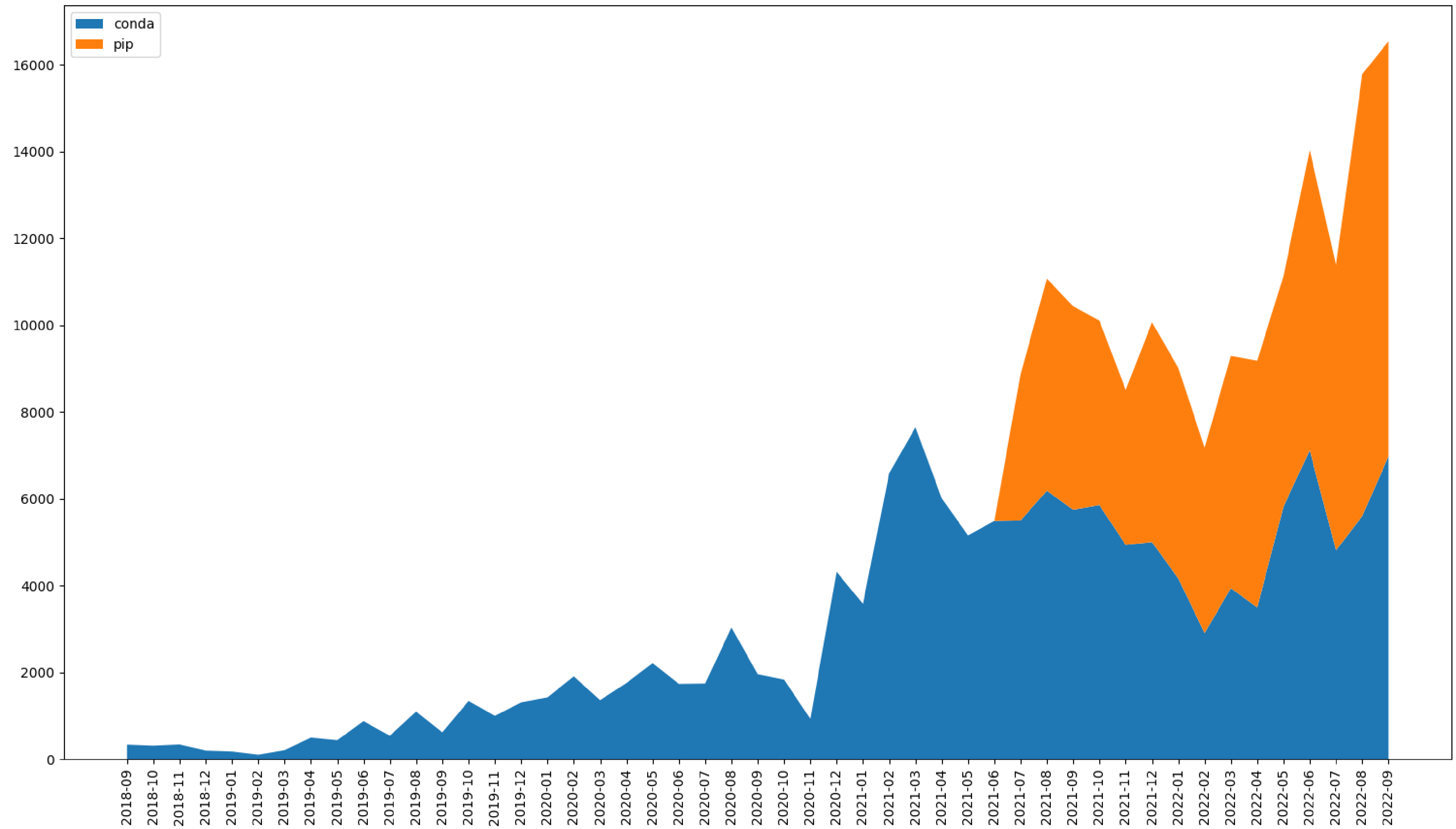


The Gudhi library



The Gudhi library

conda: ∼250 000 total downloads
pip: ∼34 000 downloads over the last 6 months



I. Turn datasets into simplicial complexes

II. Compute and compare persistence diagrams

III. Feed / regularize ML models w/ topology

I. Turn datasets into simplicial complexes

II. Compute and compare persistence diagrams

III. Feed / regularize ML models w/ topology

Čech/Alpha complexes

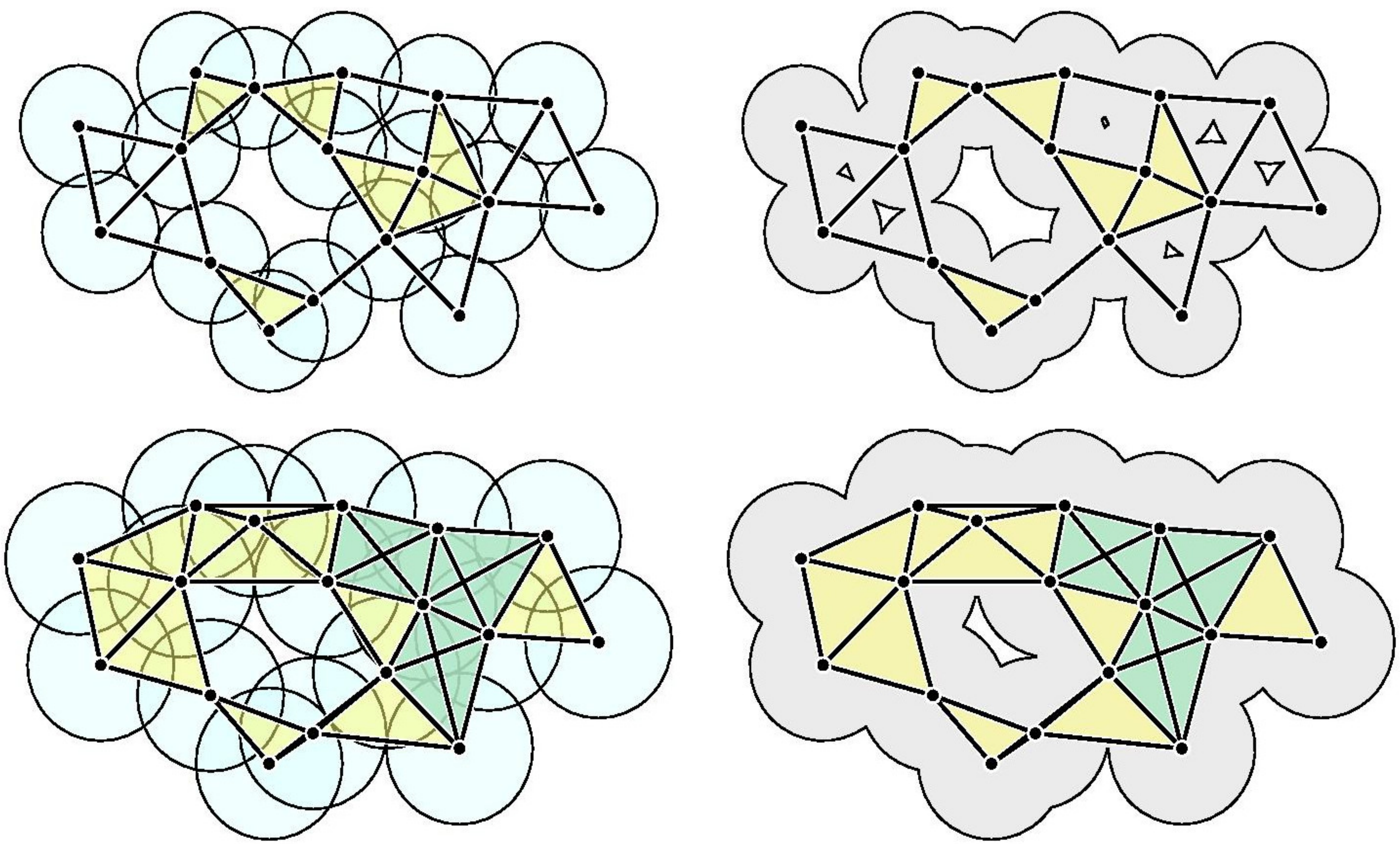
Def: Given a point cloud $P = \{P_1, \ldots, P_n\} \subset \mathbb{R}^d$, its Čech complex of radius $r > 0$ is the abstract simplicial complex $C(P, r)$ s.t. $\mathrm{vert}(C(P, r)) = P$ and

$$\sigma = [P_{i_0}, P_{i_1}, \ldots, P_{i_k}] \in C(P, r) \quad \text{iif} \quad \cap_{j=0}^{k} B(P_{i_j}, r) \neq \emptyset.$$

Čech/Alpha complexes

Def: Given a point cloud $P = \{P_1, \ldots, P_n\} \subset \mathbb{R}^d$, its Čech complex of radius $r > 0$ is the abstract simplicial complex $C(P, r)$ s.t. $\mathrm{vert}(C(P, r)) = P$ and

$$\sigma = [P_{i_0}, P_{i_1}, \ldots, P_{i_k}] \in C(P, r) \quad \text{iif} \quad \cap_{j=0}^{k} B(P_{i_j}, r) \neq \emptyset.$$



Čech/Alpha complexes

Def: Given a point cloud $P = \{P_1, \ldots, P_n\} \subset \mathbb{R}^d$, its Čech complex of radius $r > 0$ is the abstract simplicial complex $C(P, r)$ s.t. $\mathrm{vert}(C(P, r)) = P$ and

$$\sigma = [P_{i_0}, P_{i_1}, \ldots, P_{i_k}] \in C(P, r) \quad \text{iif} \quad \cap_{j=0}^{k} B(P_{i_j}, r) \neq \emptyset.$$

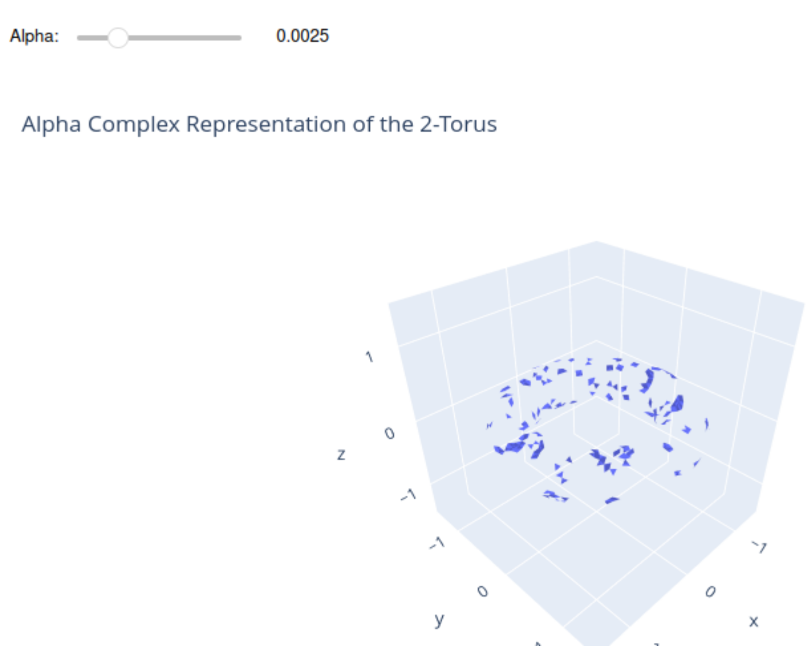
In [ ]:
```
torus = gudhi.read_points_from_off_file(off_file='datasets/tore3D_1307.off')
ac = gudhi.AlphaComplex(points=torus)
st = ac.create_simplex_tree()
```

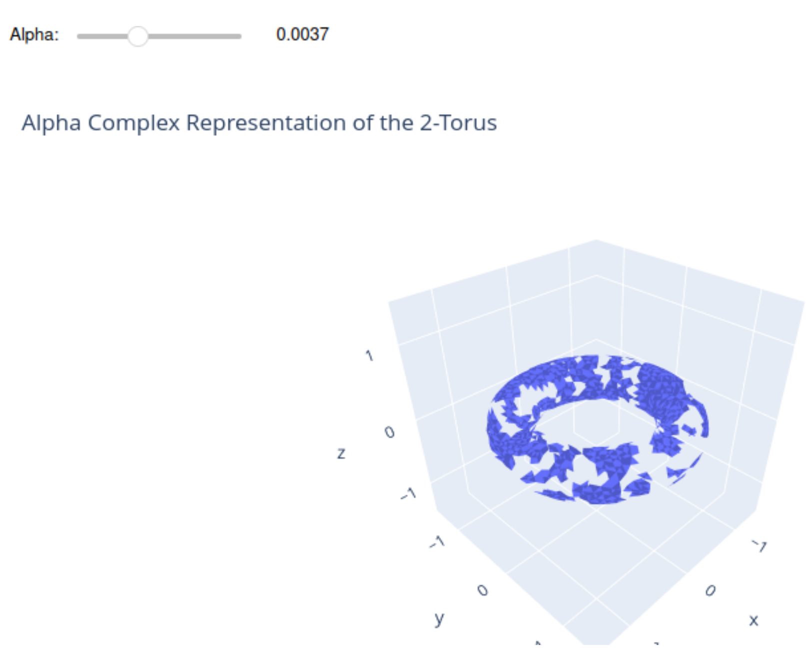
Čech/Alpha complexes

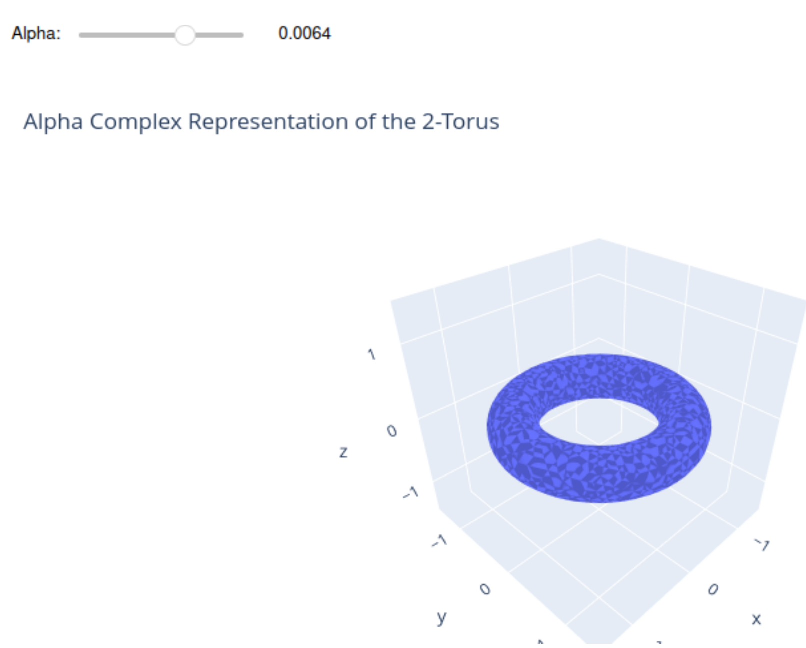
Def: Given a point cloud $P = \{P_1, \ldots, P_n\} \subset \mathbb{R}^d$, its Čech complex of radius $r > 0$ is the abstract simplicial complex $C(P, r)$ s.t. $\mathrm{vert}(C(P, r)) = P$ and

$$\sigma = [P_{i_0}, P_{i_1}, \ldots, P_{i_k}] \in C(P, r) \quad \text{iif} \quad \cap_{j=0}^{k} B(P_{i_j}, r) \neq \emptyset.$$

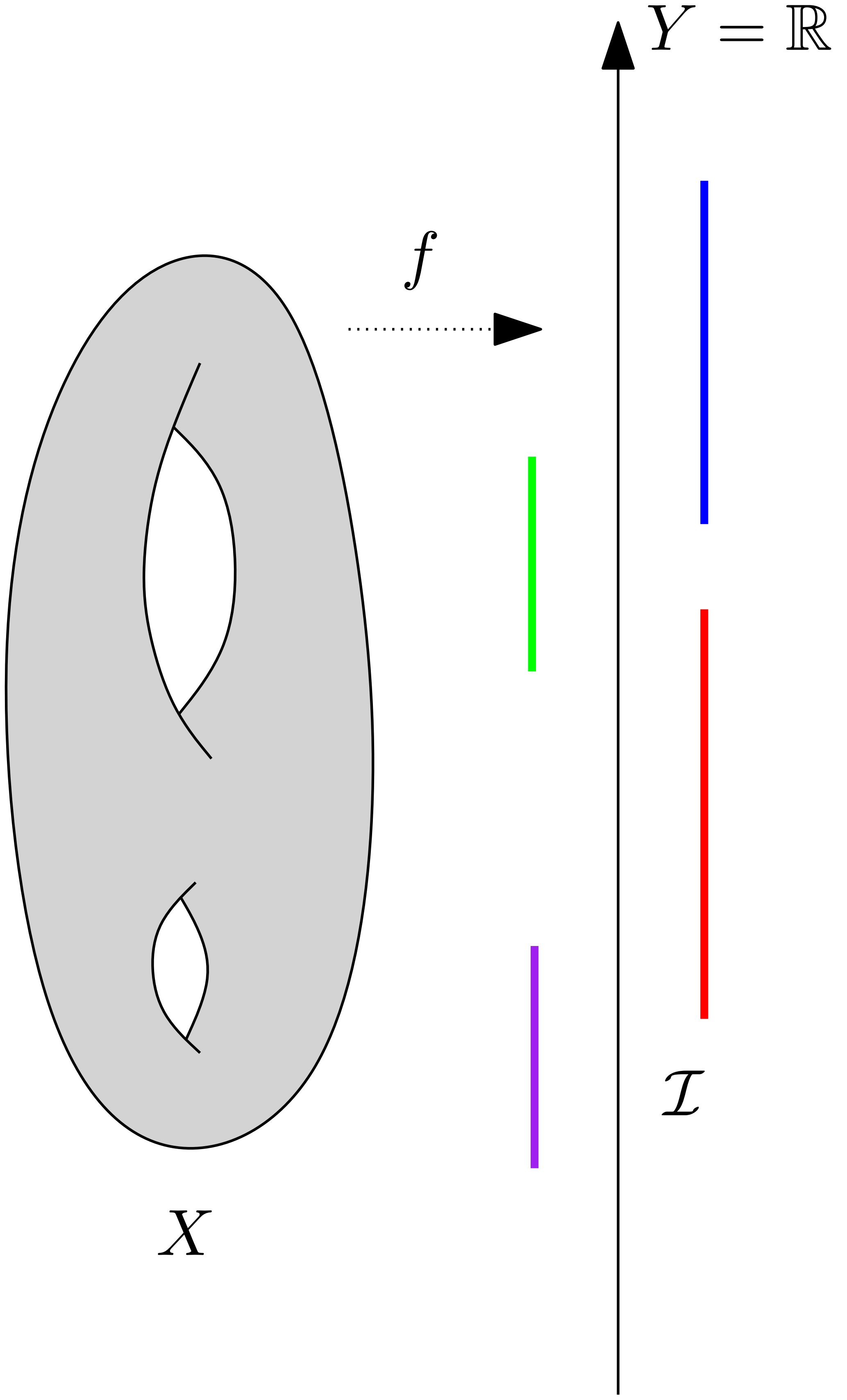
```
In [ ]: torus = gudhi.read_points_from_off_file(off_file='datasets/tore3D_1307.off')
        ac = gudhi.AlphaComplex(points=torus)
        st = ac.create_simplex_tree()
```



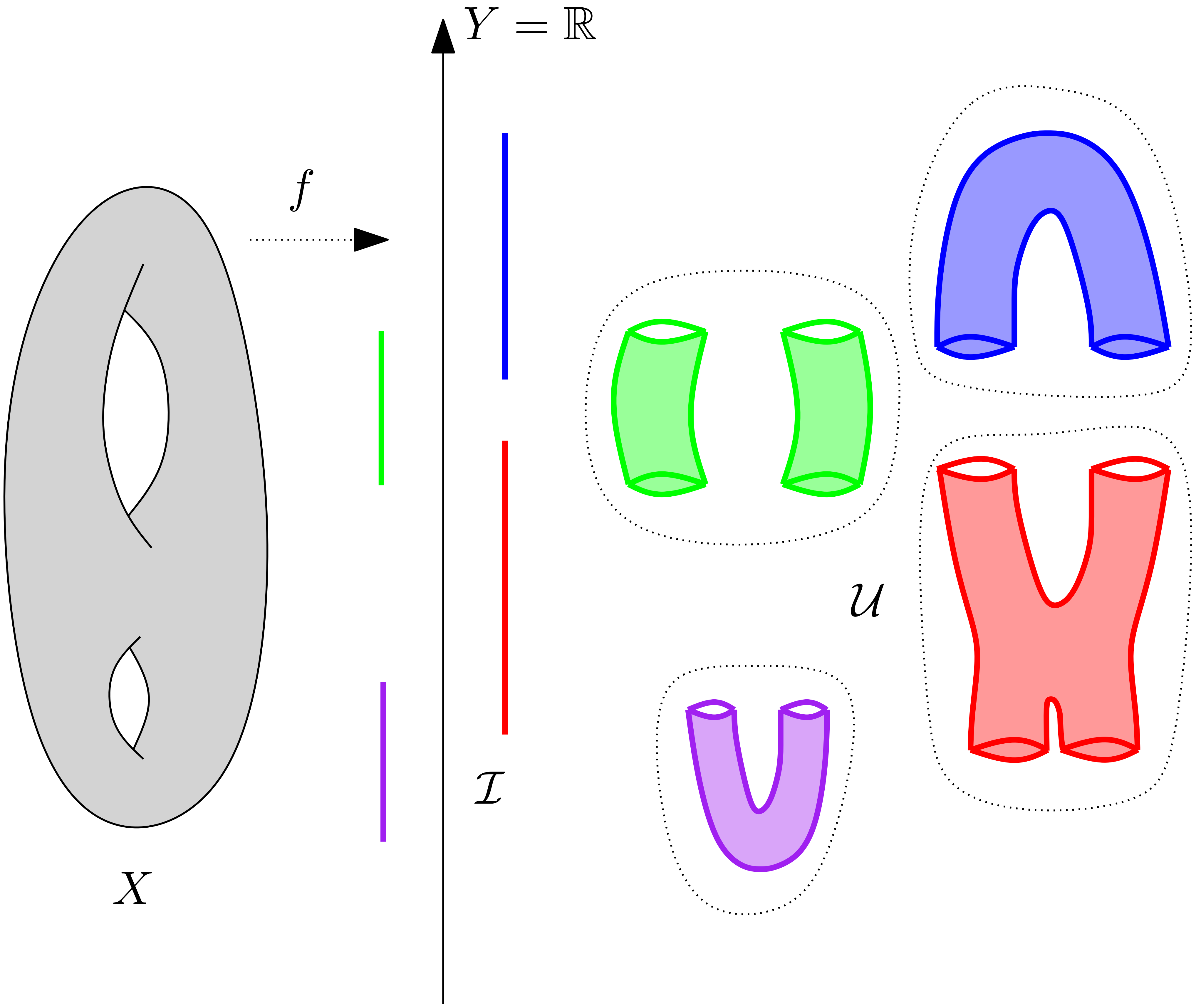




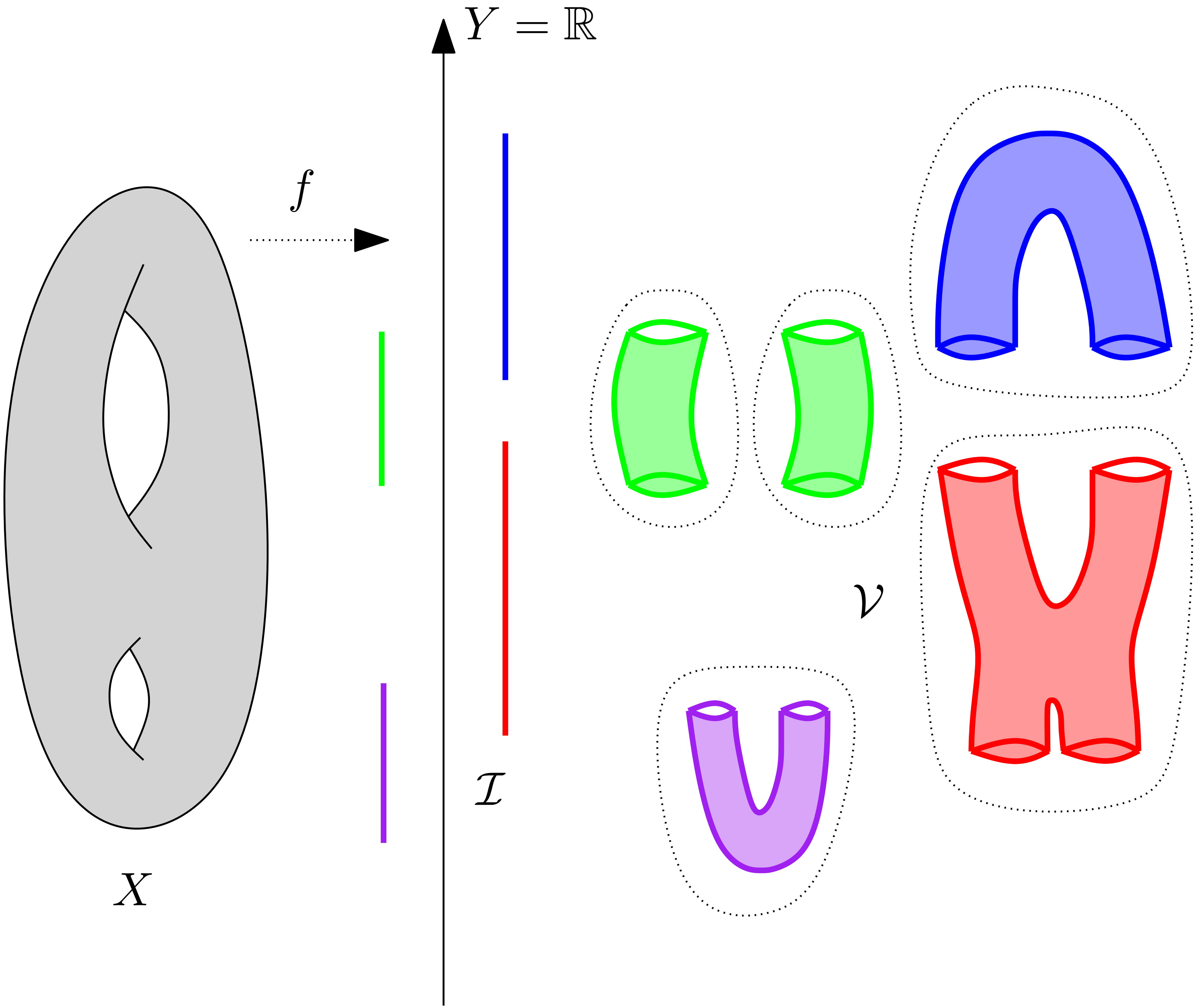
Mapper complexes



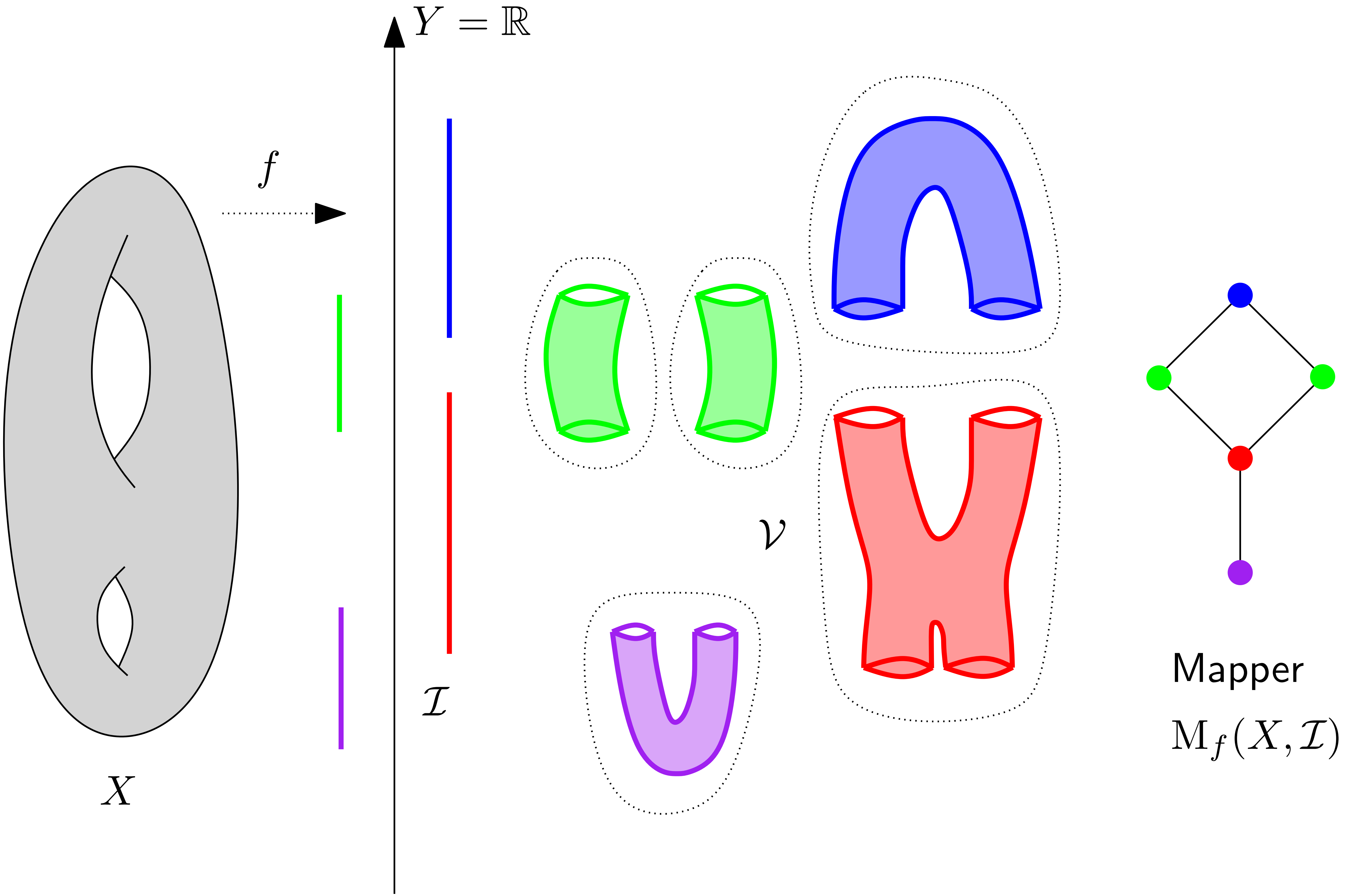
Mapper complexes



Mapper complexes



Mapper complexes



Mapper complexes

Input:

- topological space X
- continuous function $f : X \to Y$ (99% of the time $Y = \mathbb{R}^D$)
- cover $\mathcal{I}$ of $\mathrm{im}(f)$ by open intervals: $\mathrm{im}(f) \subseteq \bigcup_{I \in \mathcal{I}} I$

Method:

- Compute *pullback cover* $\mathcal{U}$ of X: $\mathcal{U} = \{f^{-1}(I)\}_{I \in \mathcal{I}}$
- Refine $\mathcal{U}$ by separating each of its elements into its various connected components in X $\to$ connected cover $\mathcal{V}$
- The Mapper is the *nerve* of $\mathcal{V}$:
  - 1 vertex per element $V \in \mathcal{V}$
  - 1 edge per intersection $V \cap V' \neq \emptyset$, $V, V' \in \mathcal{V}$
  - 1 k-simplex per $(k+1)$-fold intersection $\bigcap_{i=0}^{k} V_i \neq \emptyset$, $V_0, \cdots, V_k \in \mathcal{V}$

Mapper complexes

Input:

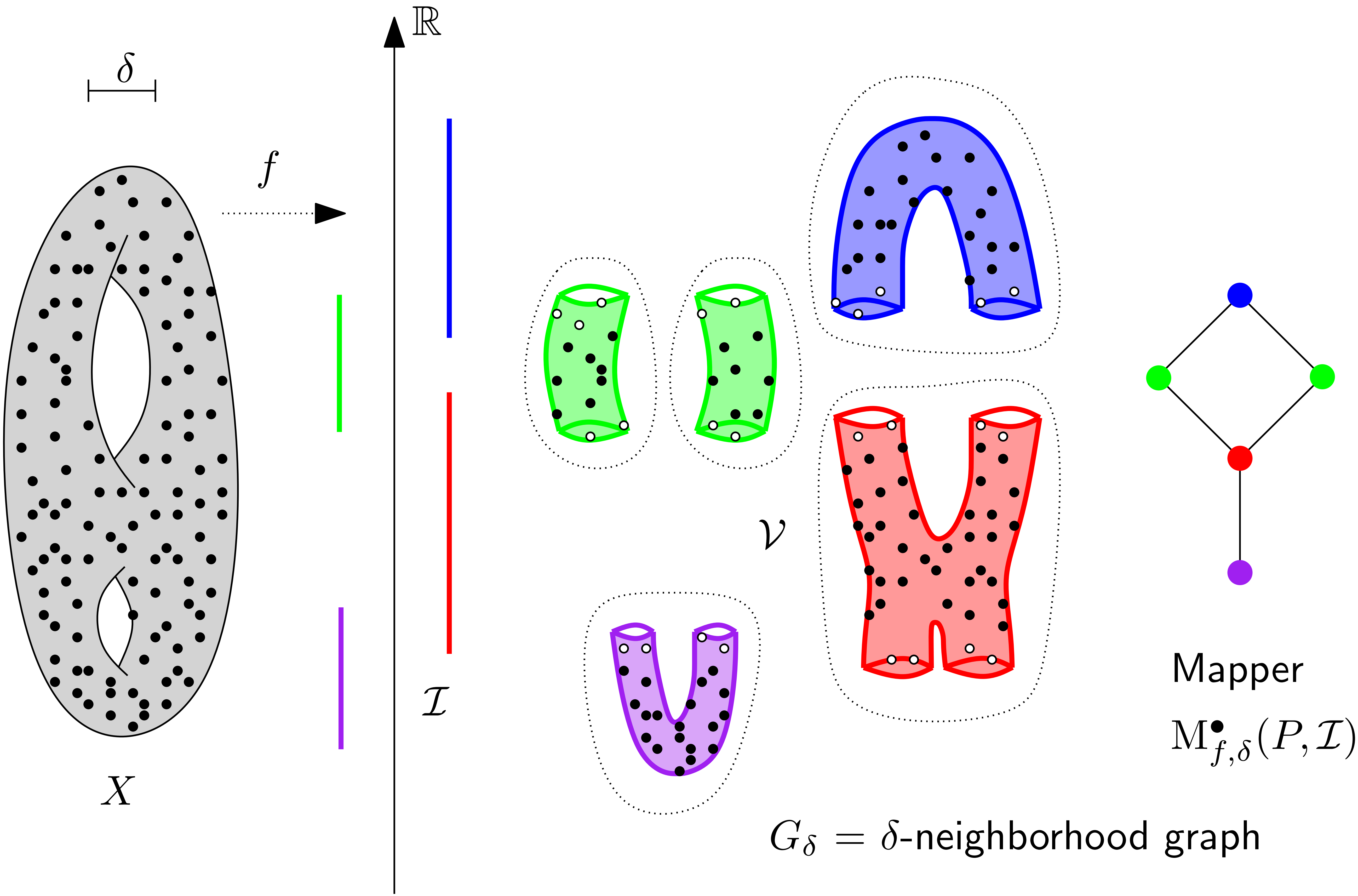
- point cloud $P \subseteq X$ with metric d_P
- continuous function $f : P \to \mathbb{R}$
- cover $\mathcal{I}$ of $\mathrm{im}(f)$ by open intervals: $\mathrm{im} f \subseteq \bigcup_{I \in \mathcal{I}} I$

Method:

- Compute *pullback cover* $\mathcal{U}$ of P: $\mathcal{U} = \{f^{-1}(I)\}_{I \in \mathcal{I}}$
- Refine $\mathcal{U}$ by separating each of its elements into its various clusters, as identified by a clustering algorithm $\to$ connected cover $\mathcal{V}$
- The Mapper is the *nerve* of $\mathcal{V}$:
  - 1 vertex per element $V \in \mathcal{V}$
  - 1 edge per intersection $V \cap V' \neq \emptyset$, $V, V' \in \mathcal{V}$
  - 1 k-simplex per $(k+1)$-fold intersection $\bigcap_{i=0}^{k} V_i \neq \emptyset$, $V_0, \cdots, V_k \in \mathcal{V}$

intersections are assessed by the presence of common data points

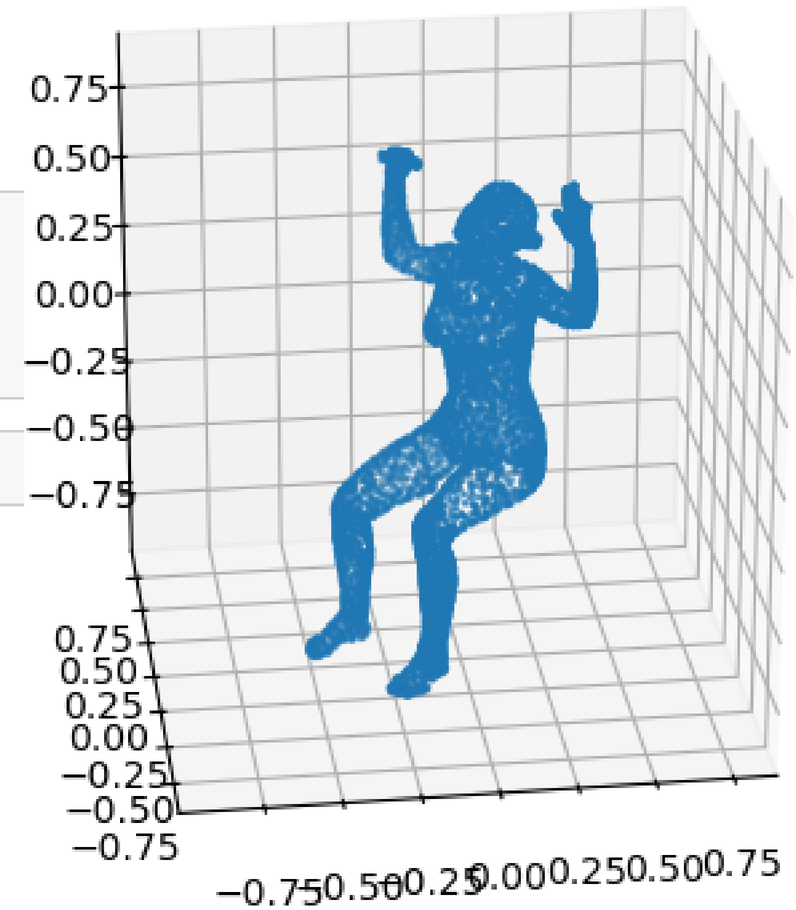
Mapper complexes



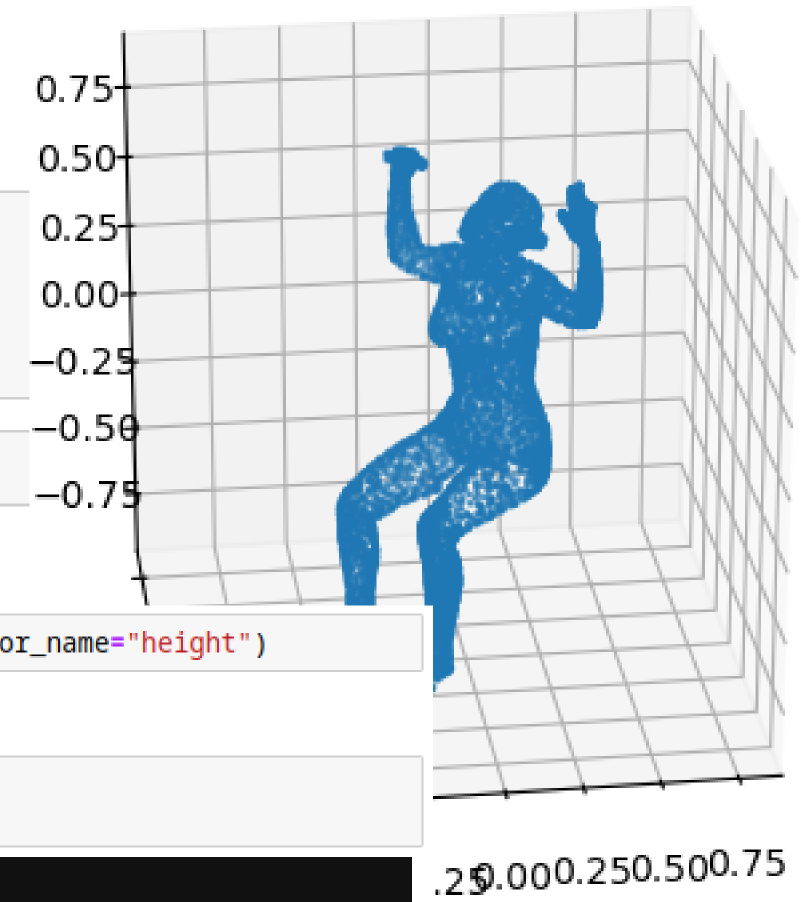
Mapper complexes

In [22]:
```
cover_complex = MapperComplex(
    input_type='point cloud', min_points_per_node=0,
    clustering=None, N=100, beta=0., C=10,
    filter_bnds=None, resolutions=[20,2], gains=None, verbose=verbose)
```

In [23]:
```
_ = cover_complex.fit(X, filters=filt2d, colors=filt2d)
```



Mapper complexes



In [22]:
```
cover_complex = MapperComplex(
    input_type='point cloud', min_points_per_node=0,
    clustering=None, N=100, beta=0., C=10,
    filter_bnds=None, resolutions=[20,2], gains=None, verbose=verbose)
```

In [23]:
```
_ = cover_complex.fit(X, filters=filt2d, colors=filt2d)
```

In [24]:
```
cover_complex.save_to_html(file_name="human", data_name="human", cover_name="uniform", color_name="height")
```

'human.html' is generated. You can now use your favorite web browser to visualize it.

In [25]:
```
from IPython.display import IFrame
IFrame(src="human.html", width='100%', height='500px')
```

Out[25]:

I. Turn datasets into simplicial complexes

II. Compute and compare persistence diagrams

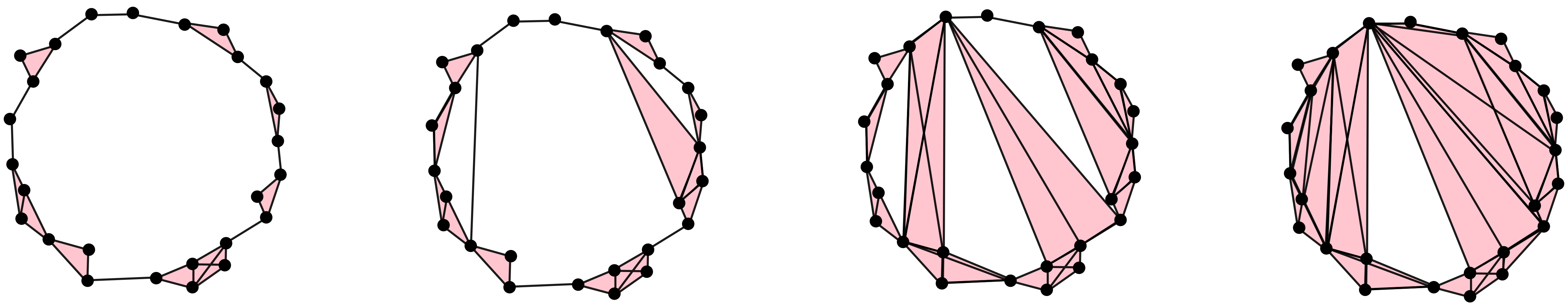
III. Feed / regularize ML models w/ topology

Computation with filtrations and matrix reduction

Algorithms for computing the homology groups of a simplicial complex work by *decomposing* it with a so-called *filtration*.

Computation with filtrations and matrix reduction

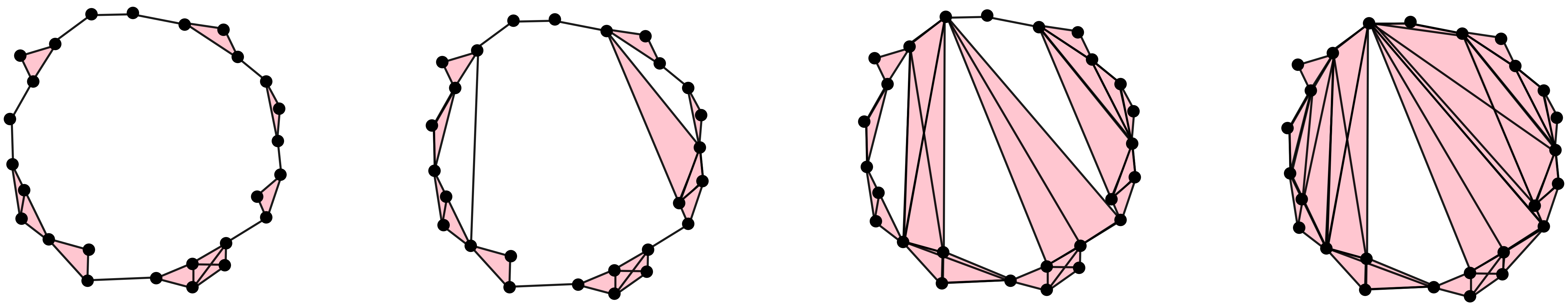
Algorithms for computing the homology groups of a simplicial complex work by *decomposing* it with a so-called *filtration*.



Def: A filtered simplicial complex S is a family $\{S_u\}_{u \in \mathbb{R}}$ of subcomplexes of some fixed simplicial complex S s.t. $S_a \subseteq S_b$ for any $a \leq b$.

Computation with filtrations and matrix reduction

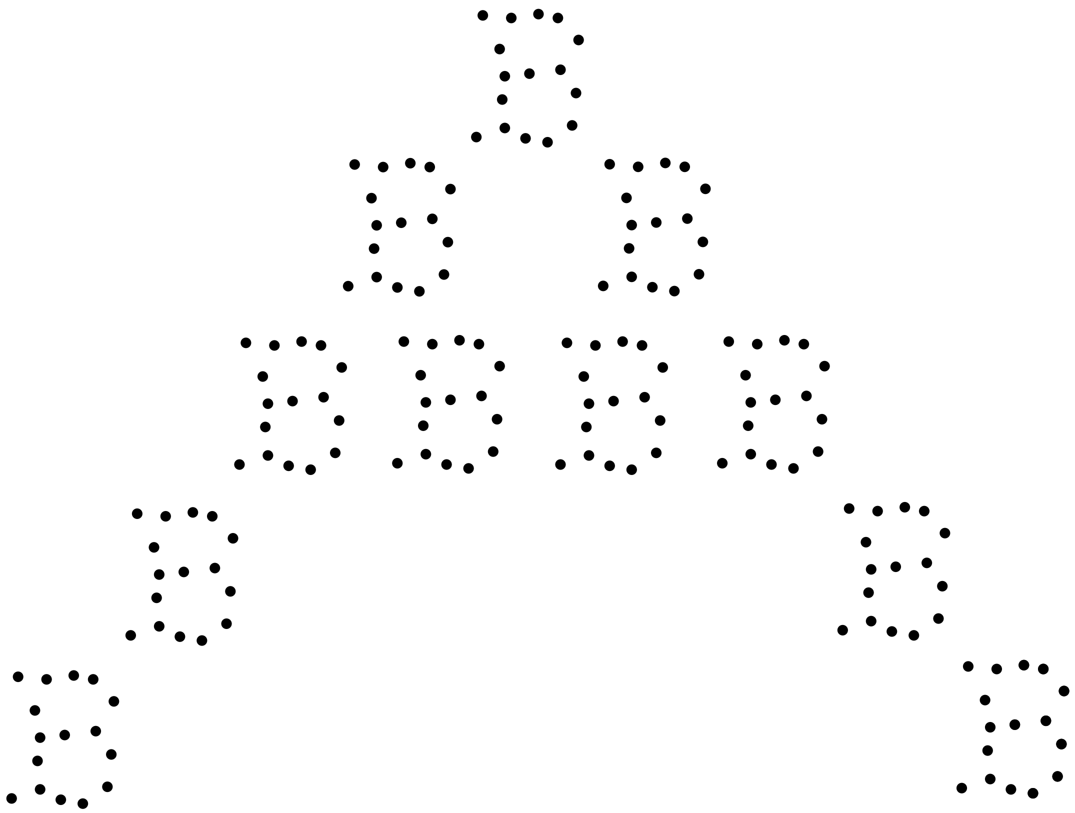
Algorithms for computing the homology groups of a simplicial complex work by *decomposing* it with a so-called *filtration*.

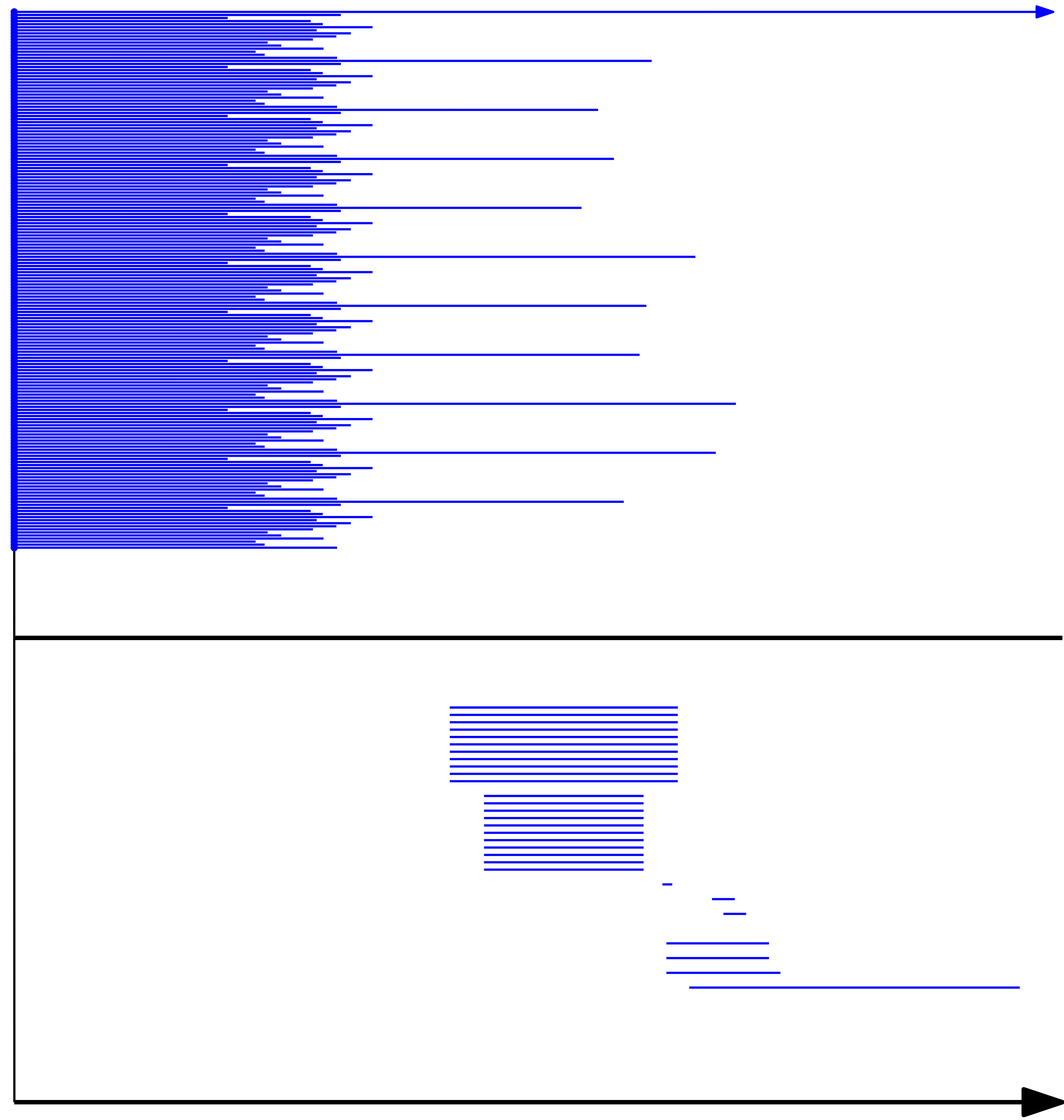


Def: A filtered simplicial complex S is a family $\{S_u\}_{u\in\mathbb{R}}$ of subcomplexes of some fixed simplicial complex S s.t. $S_a \subseteq S_b$ for any $a \leq b$.

Def: The persistence barcode (resp. diagram) D is a set of points in the plane (resp. intervals) encoding the topological features that appeared and disappeared in the filtration.

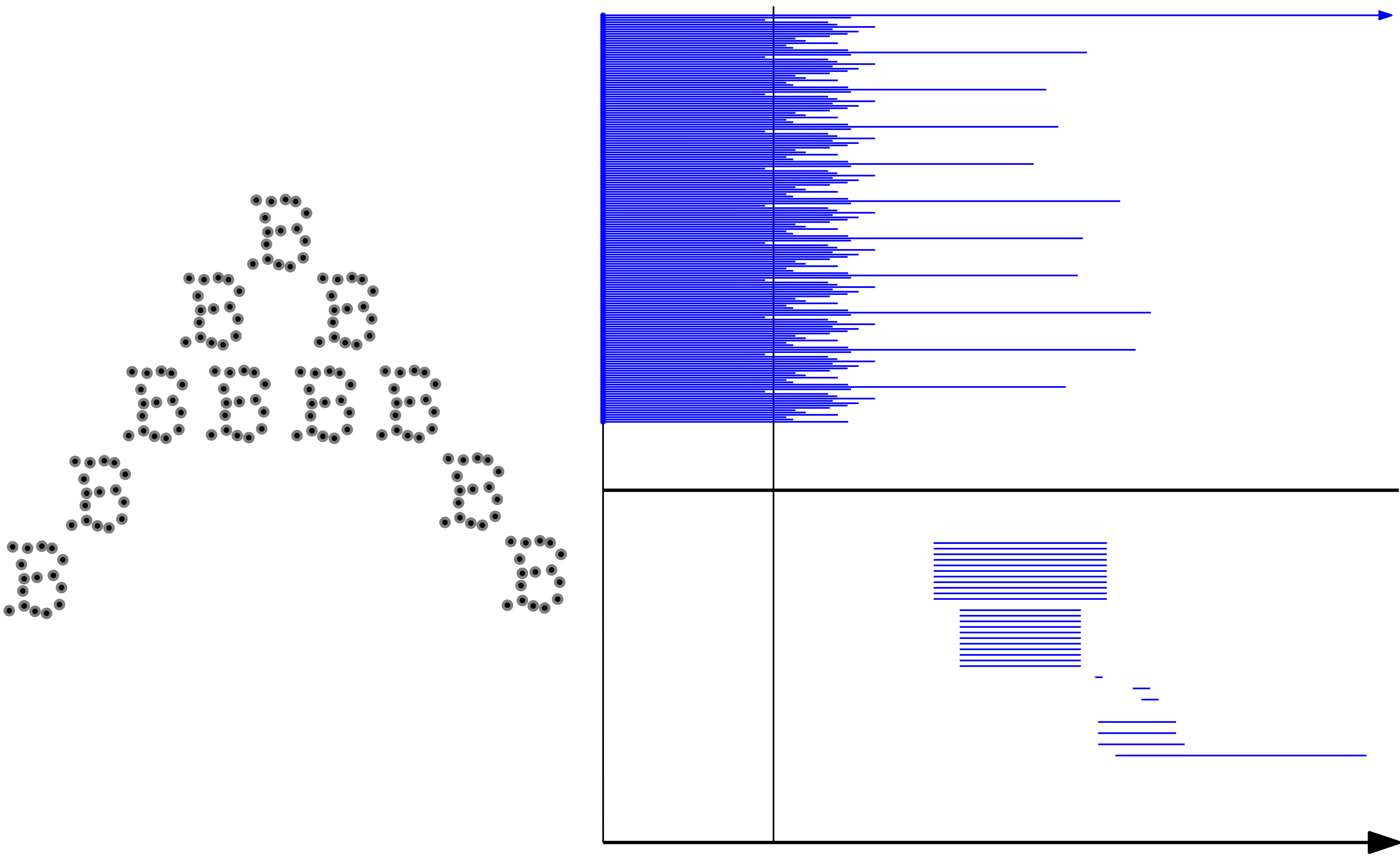
Persistence of Čech complexes





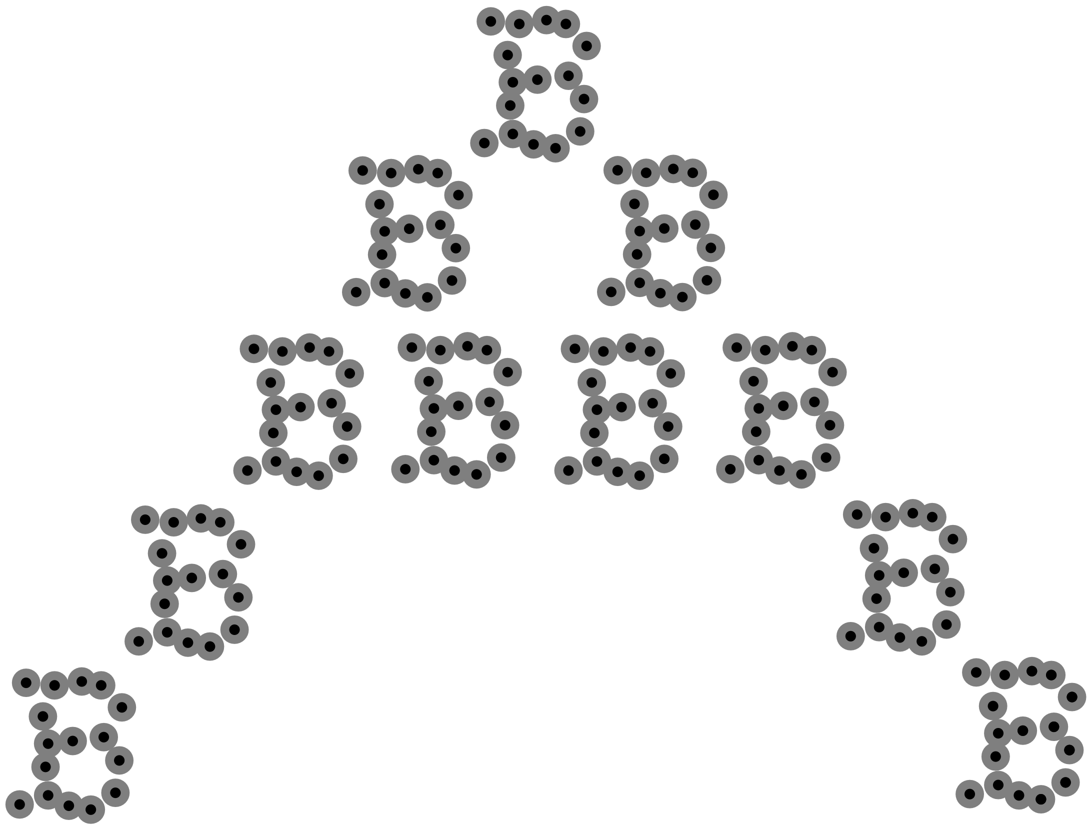
H_0 (connected components) + H_1 (loops)

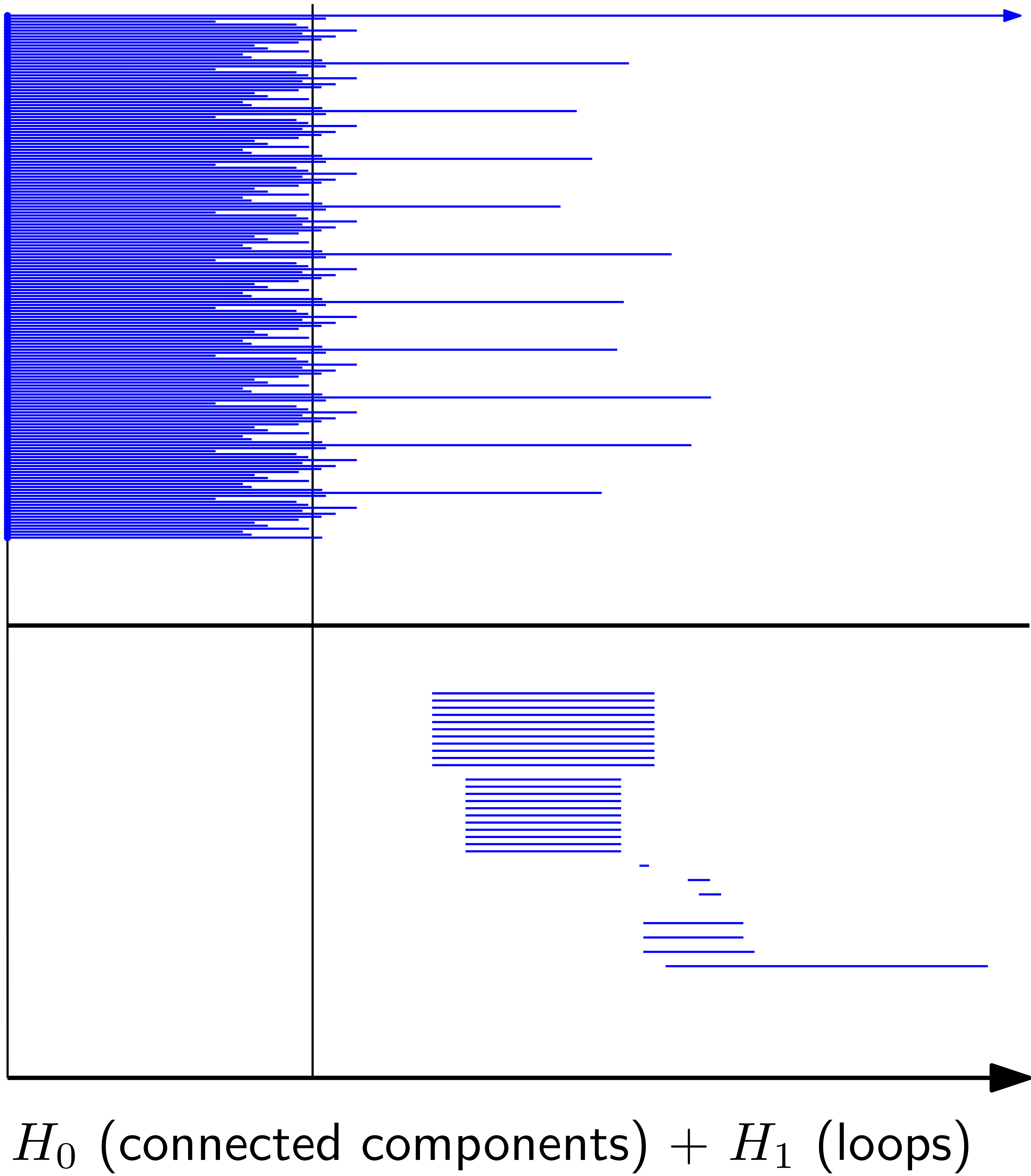
Persistence of Čech complexes



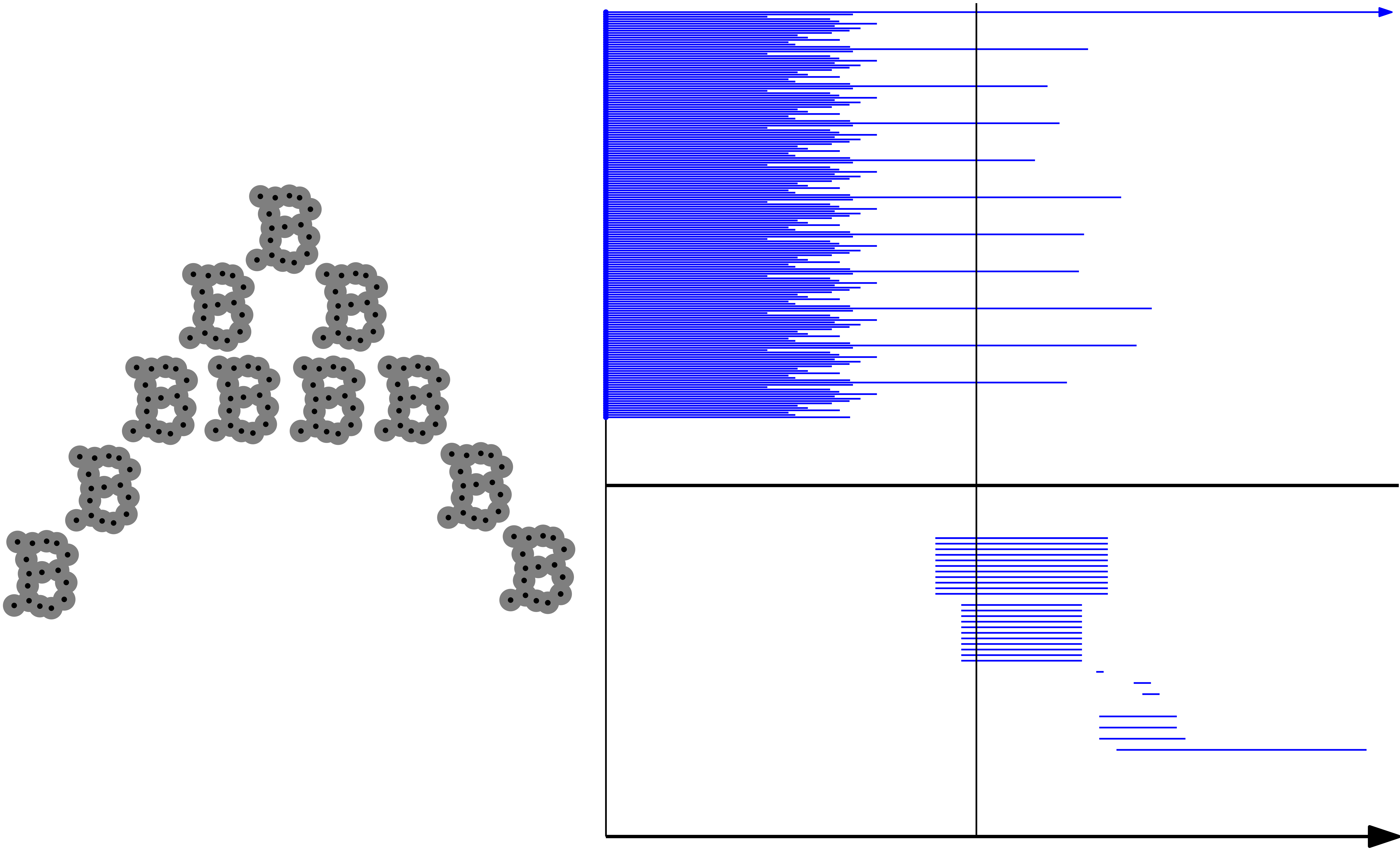
H_0 (connected components) + H_1 (loops)

Persistence of Čech complexes



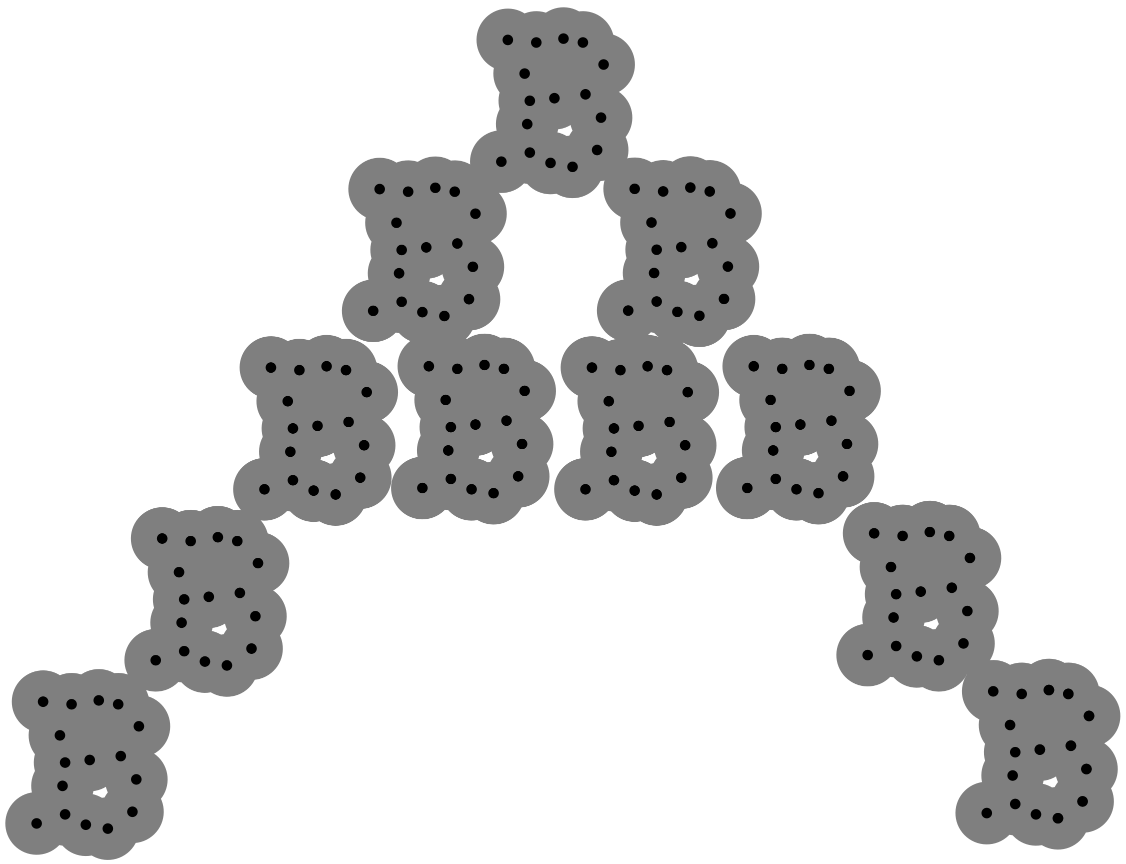


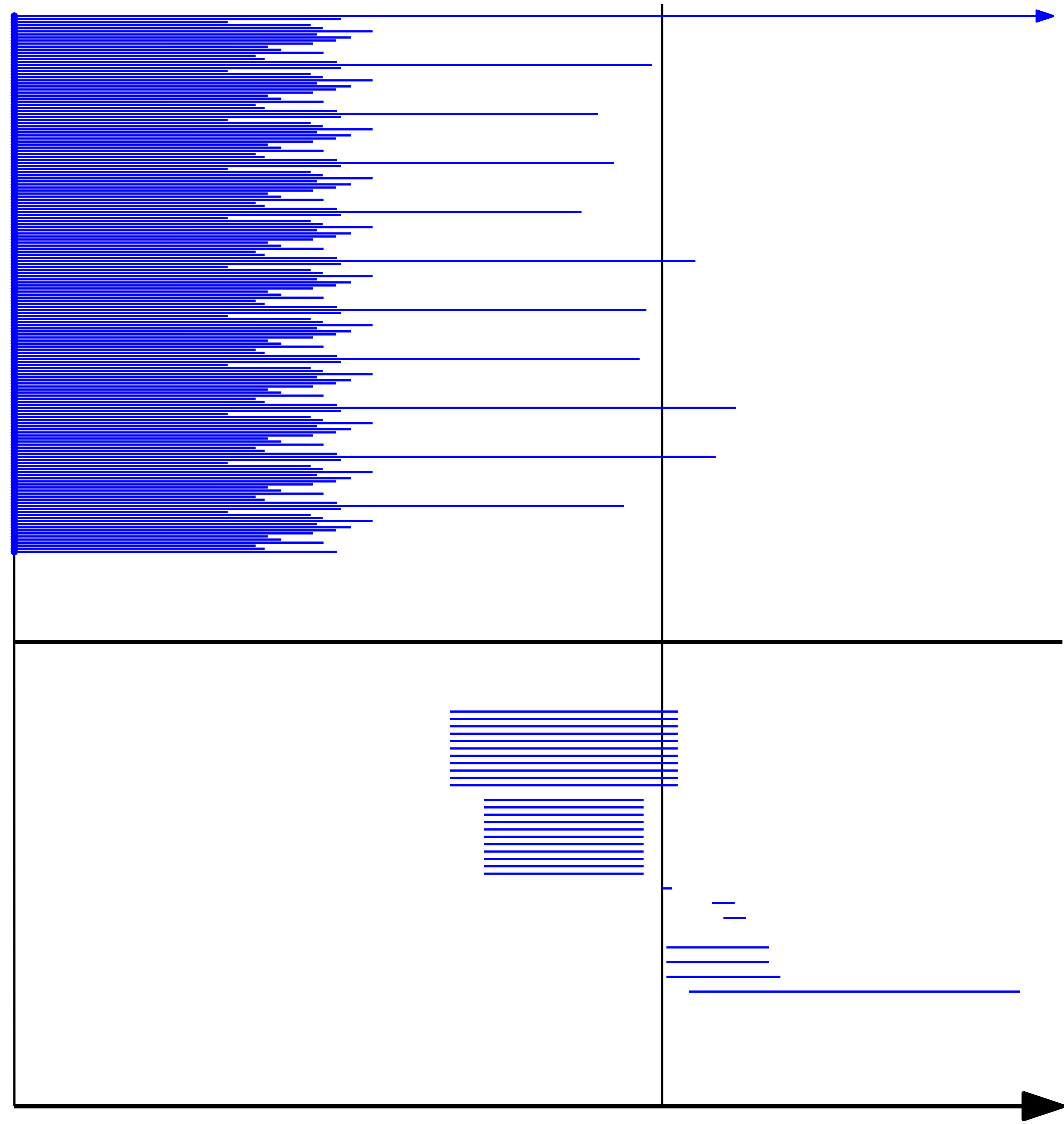
Persistence of Čech complexes



H_0 (connected components) + H_1 (loops)

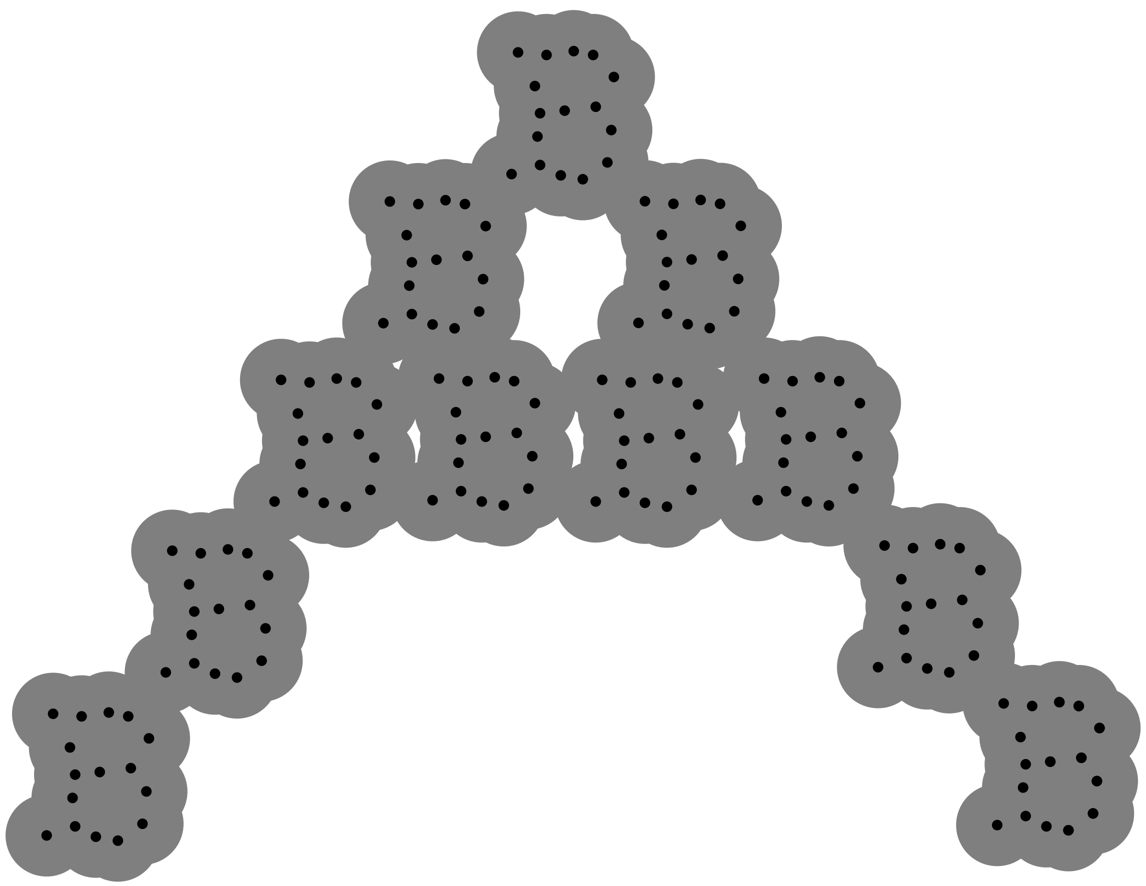
Persistence of Čech complexes

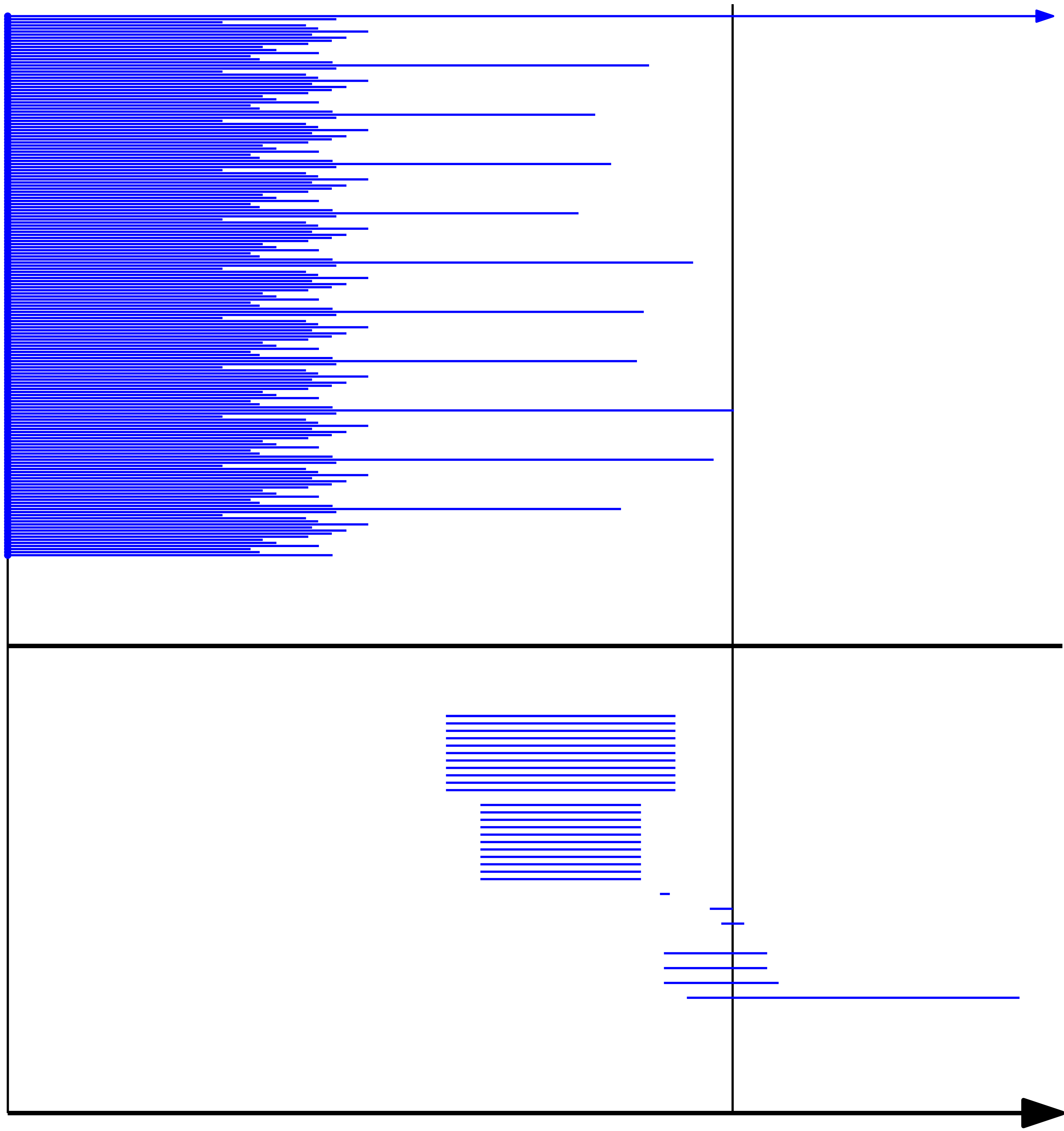




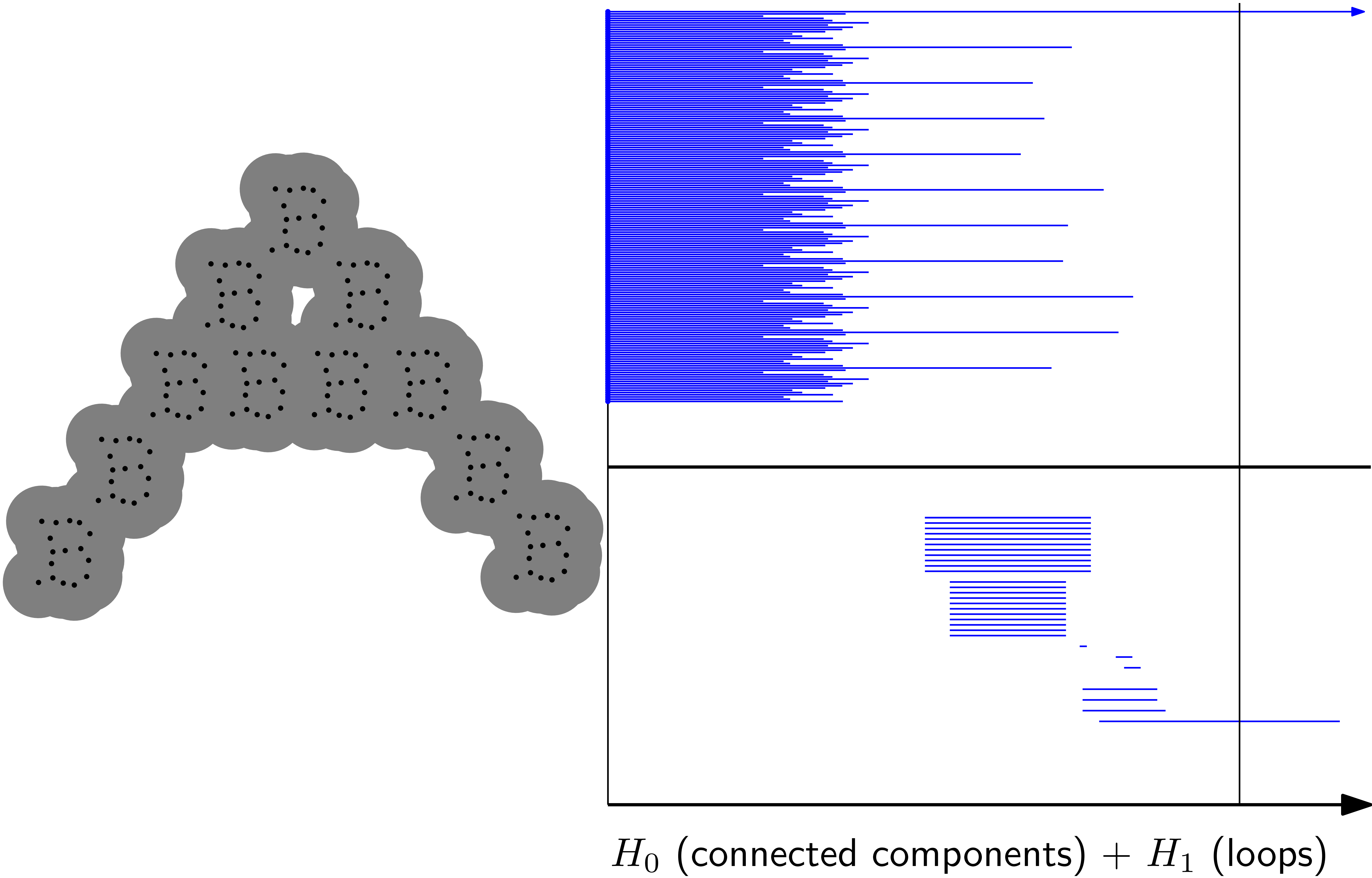
H_0 (connected components) + H_1 (loops)

Persistence of Čech complexes

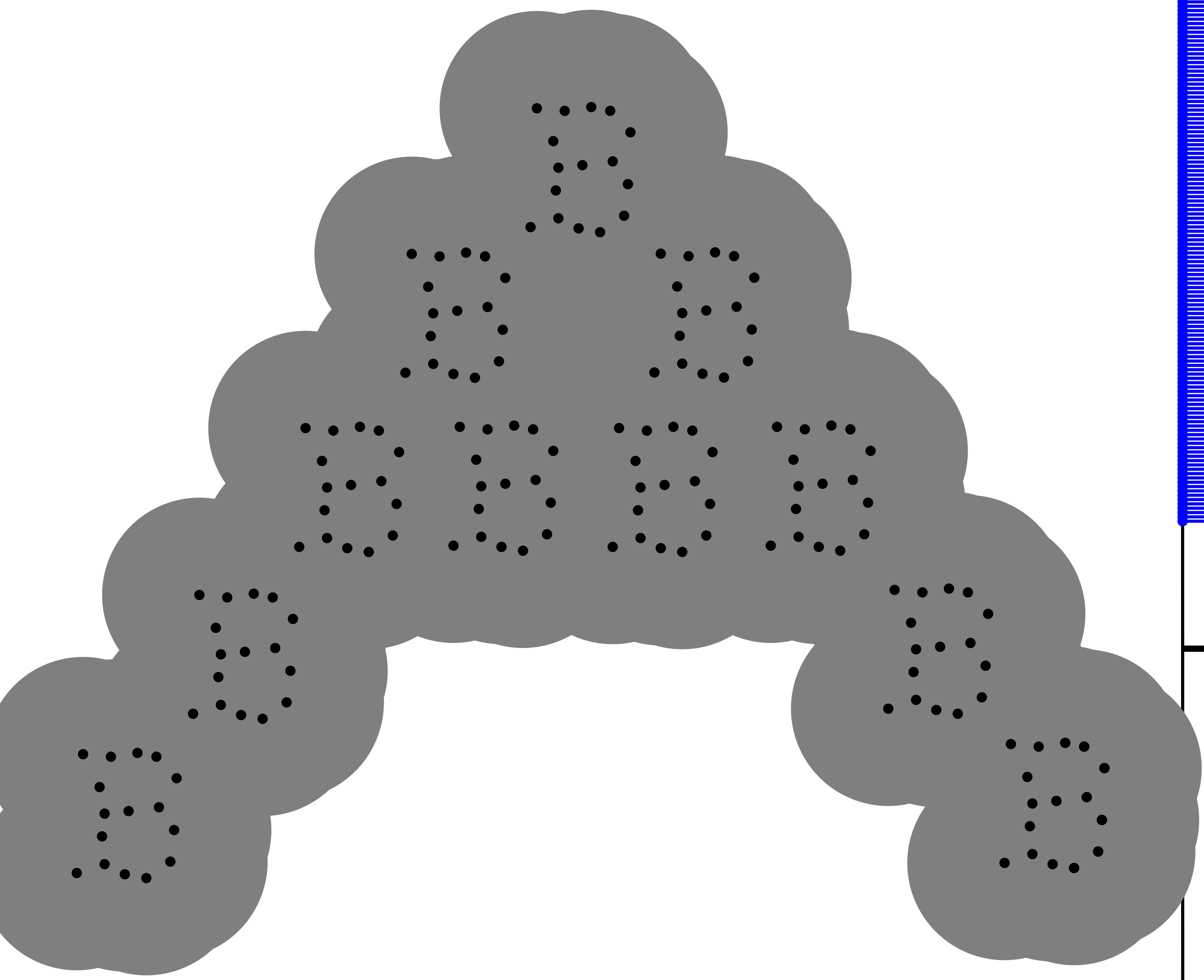


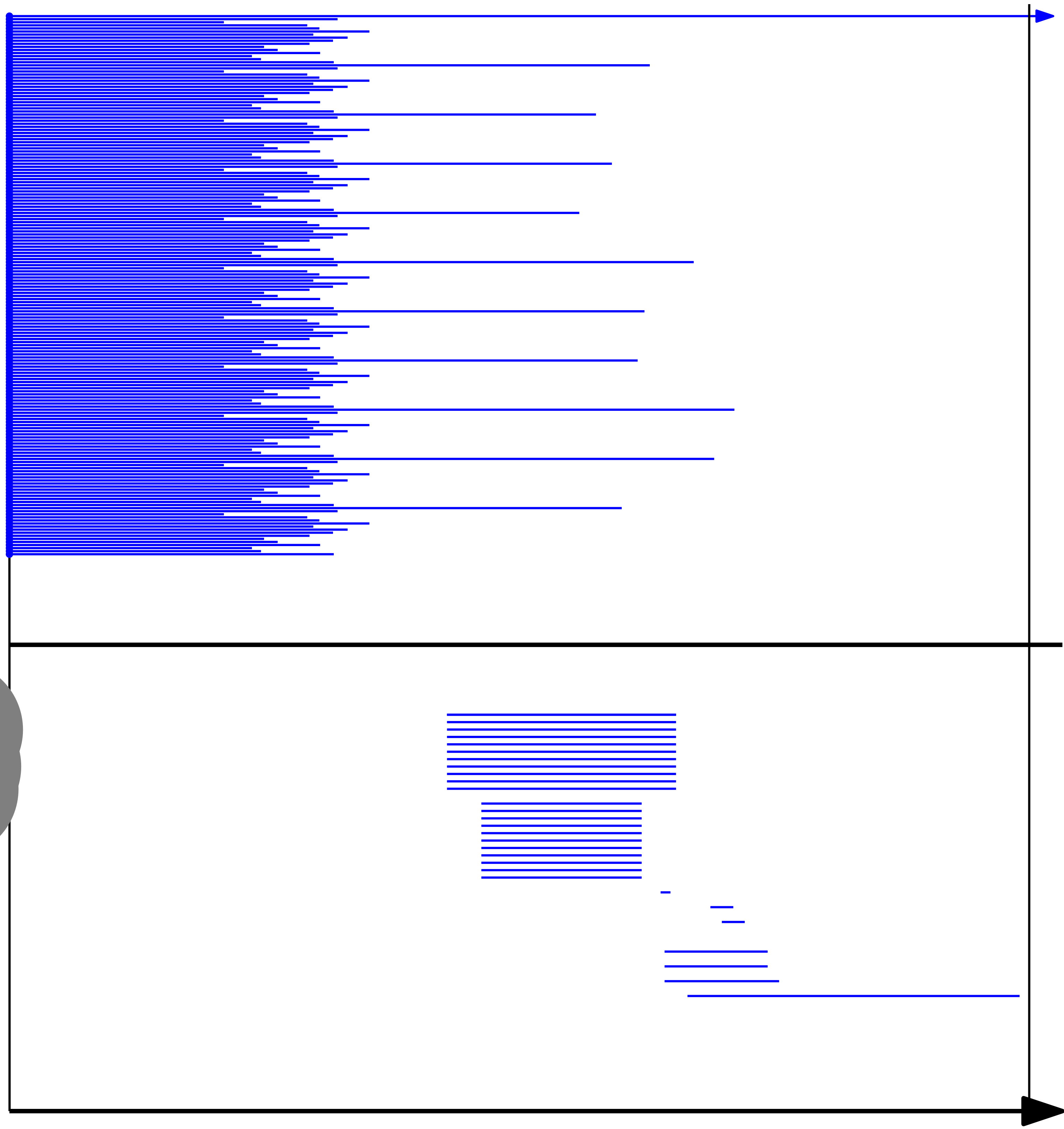


Persistence of Čech complexes



Persistence of Čech complexes

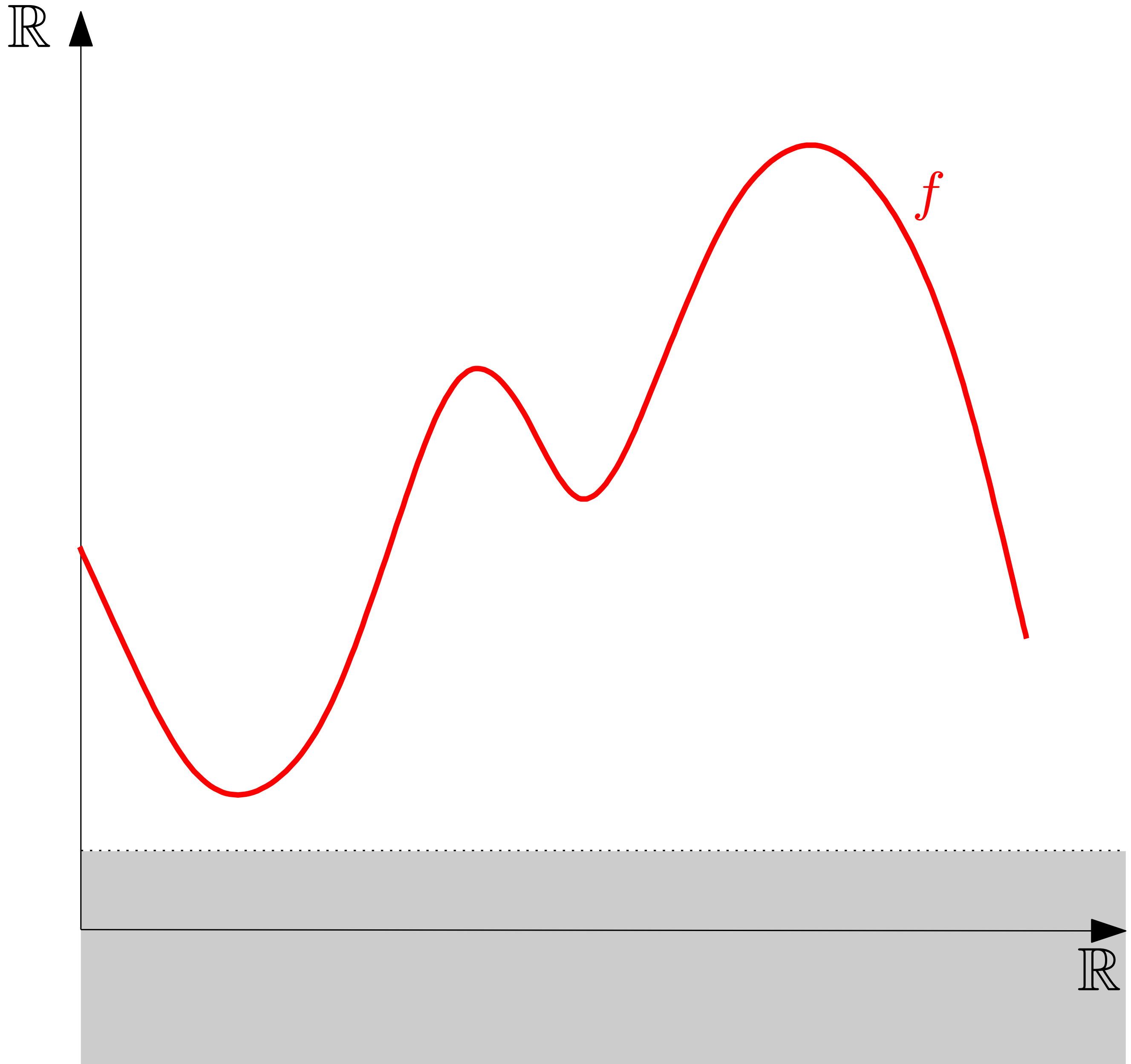




H_0 (connected components) + H_1 (loops)

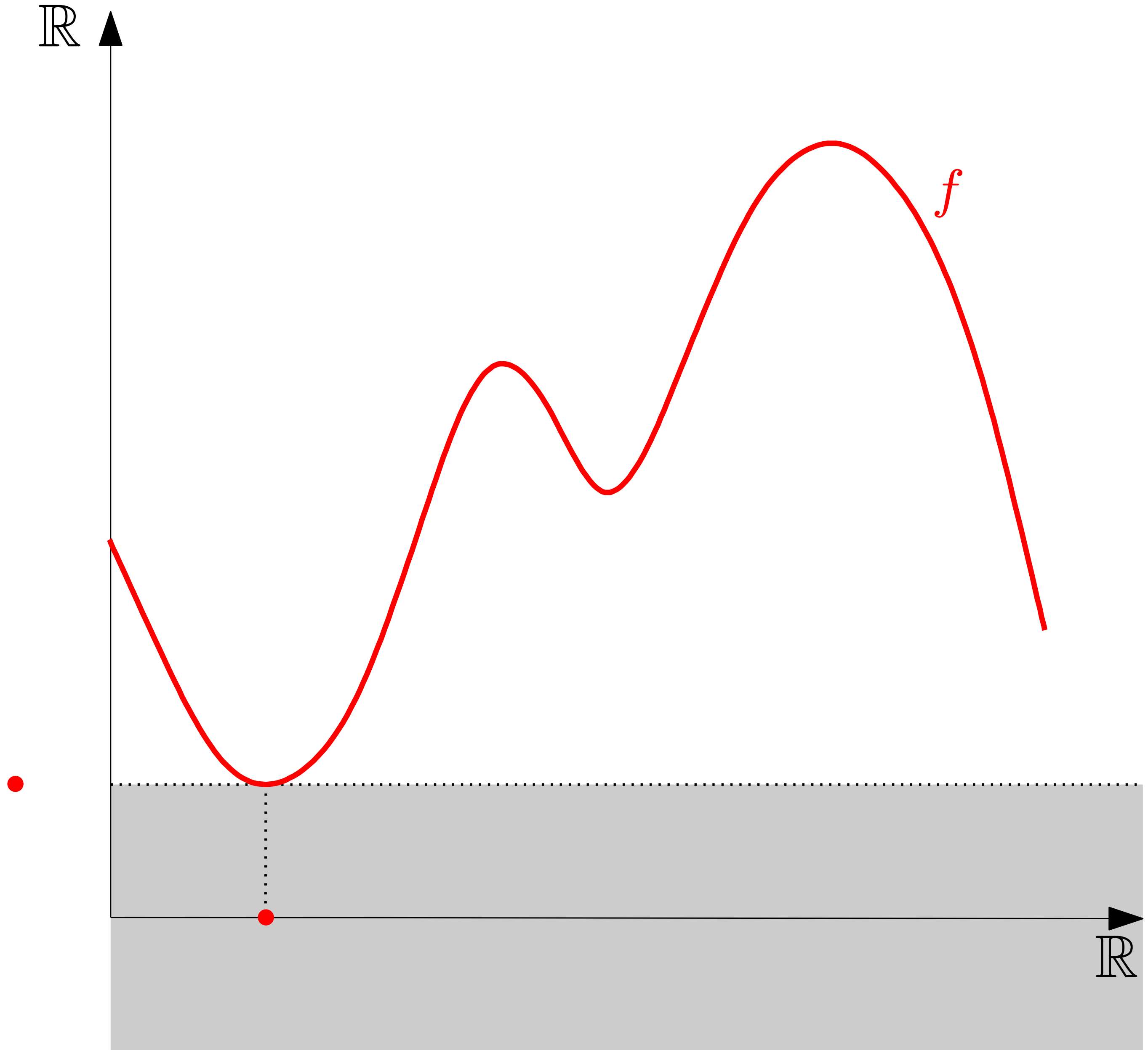
Persistence of sublevel sets of function

H_0 (connected components)



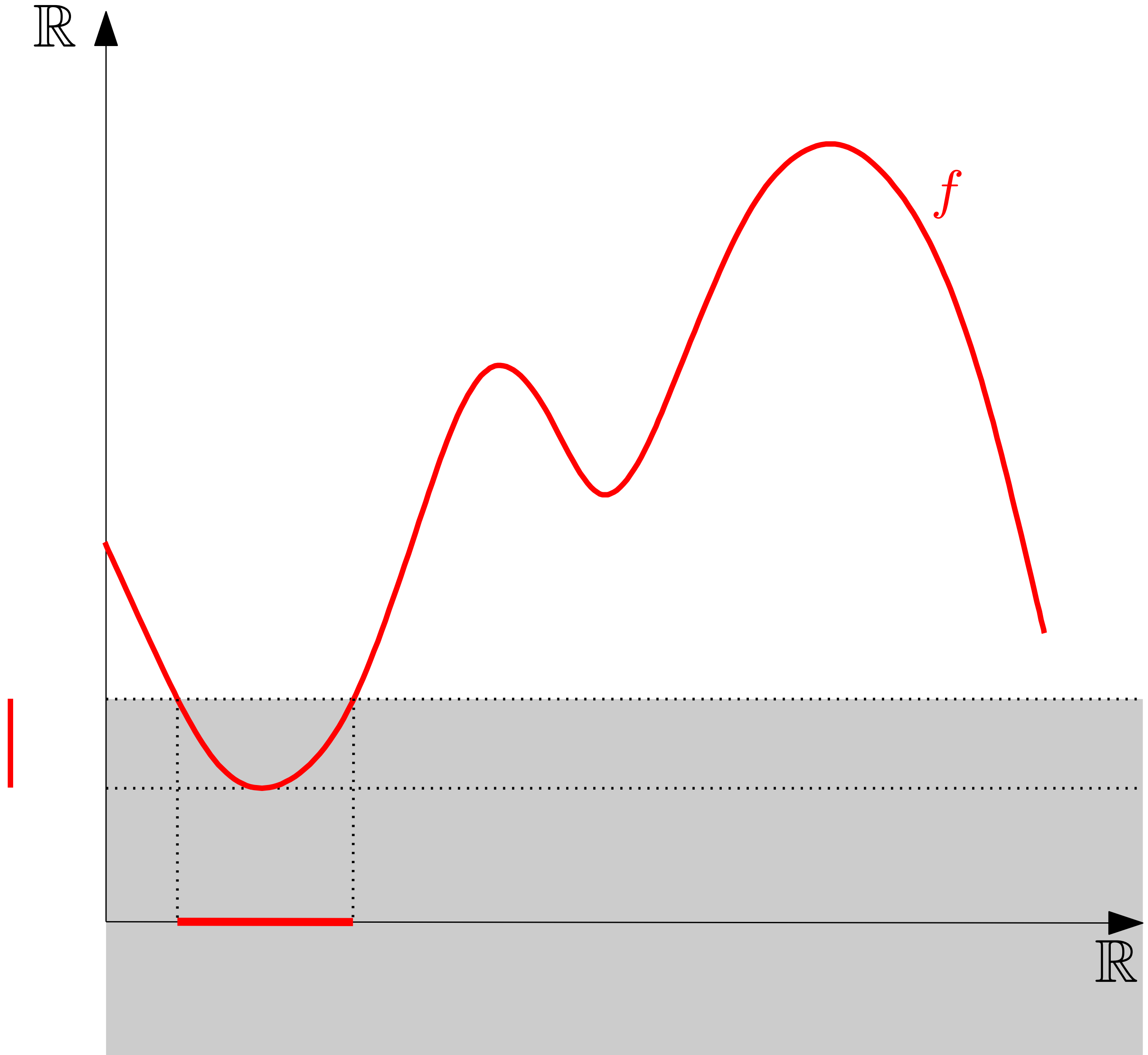
Persistence of sublevel sets of function

H_0 (connected components)



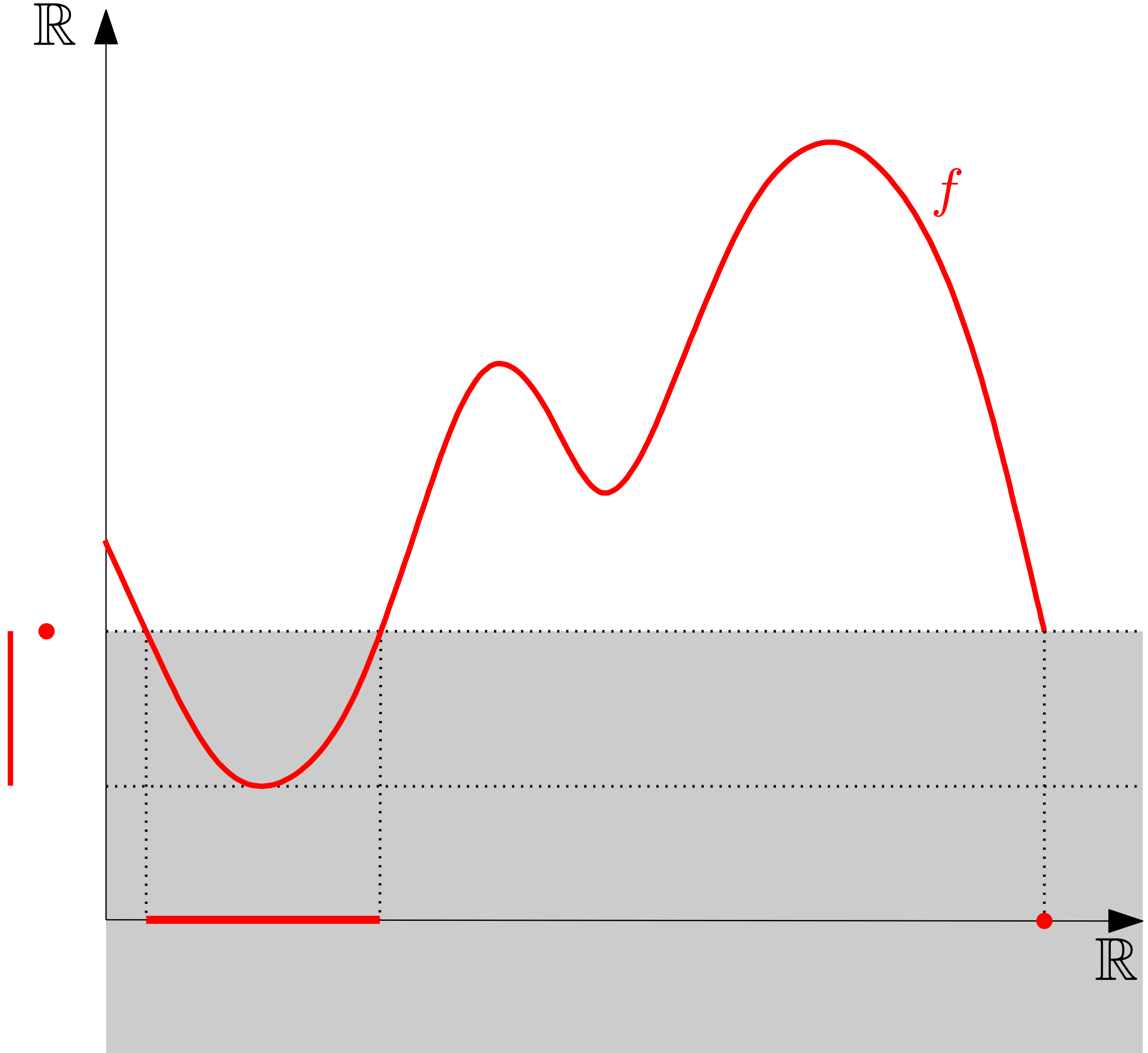
Persistence of sublevel sets of function

H_0 (connected components)



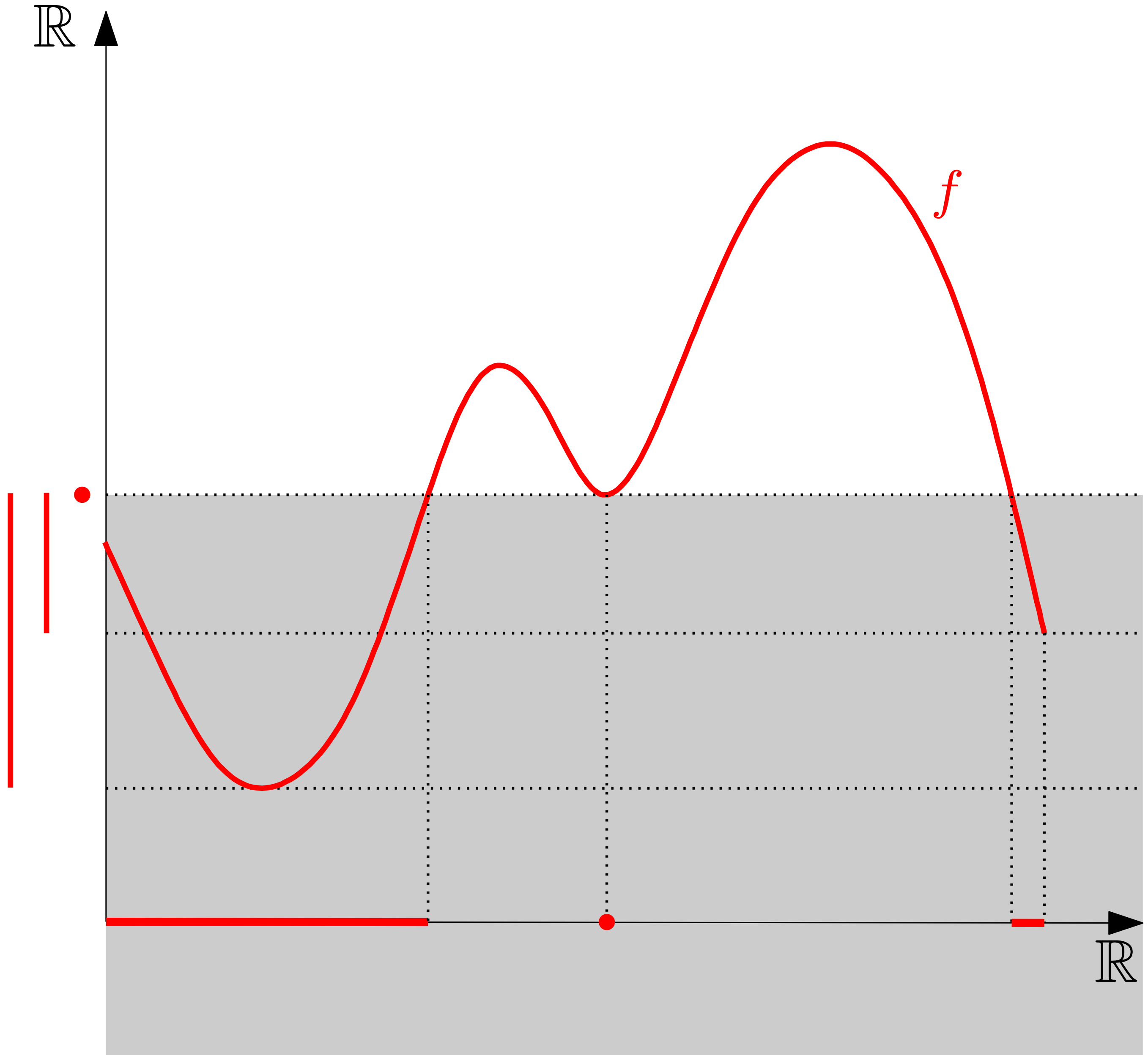
Persistence of sublevel sets of function

H_0 (connected components)



Persistence of sublevel sets of function

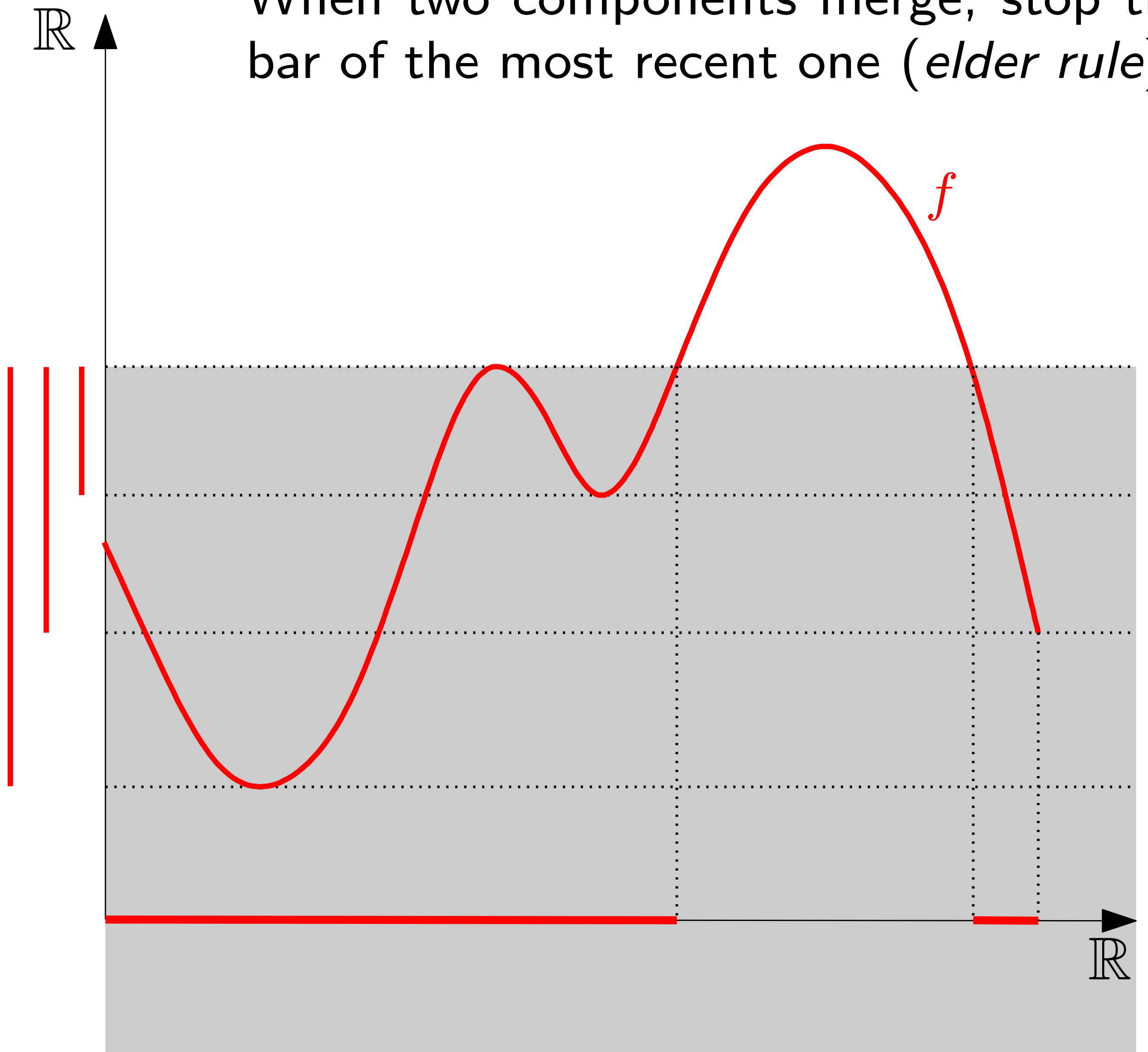
H_0 (connected components)



Persistence of sublevel sets of function

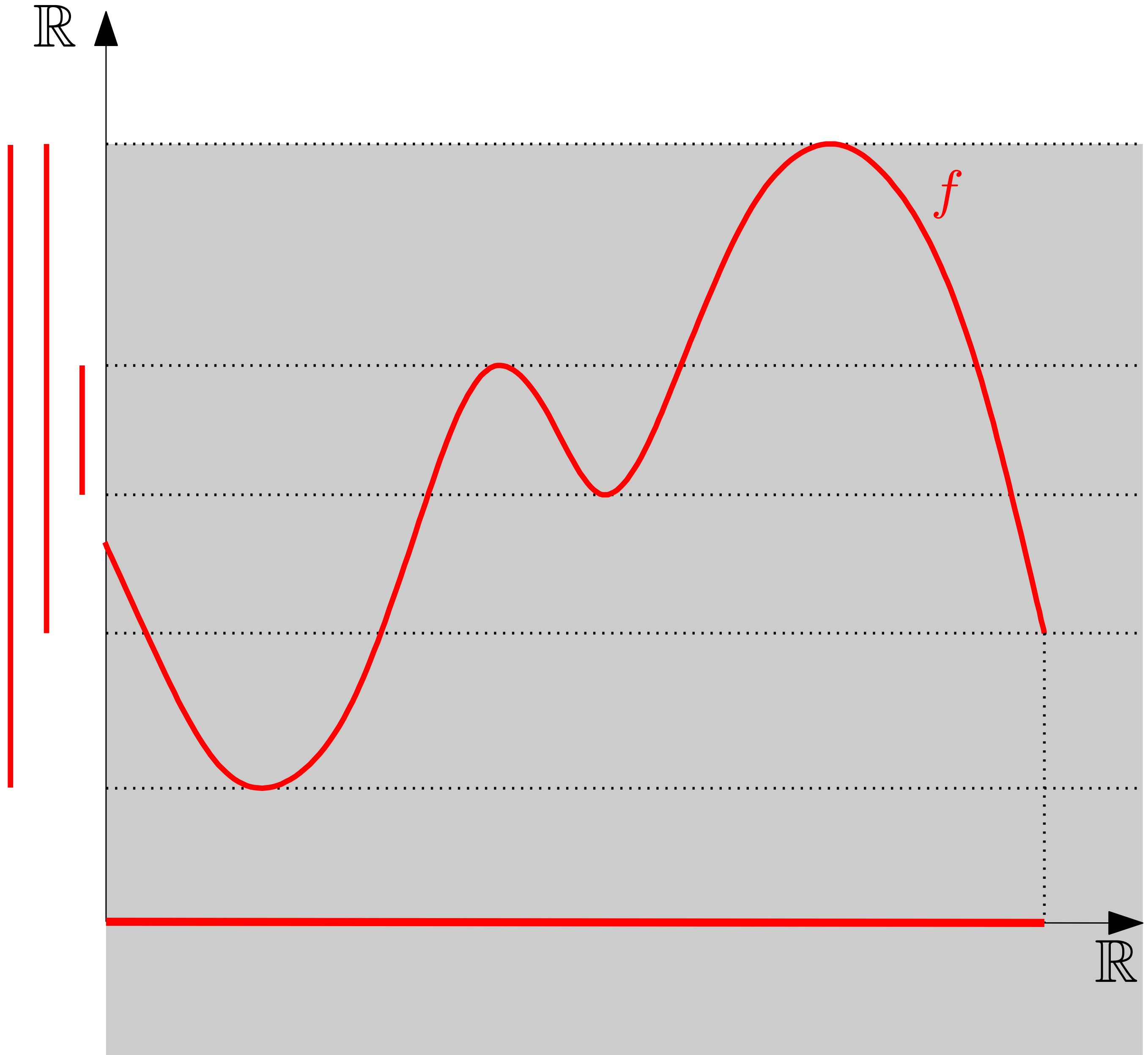
H_0 (connected components)

When two components merge, stop the bar of the most recent one (*elder rule*).



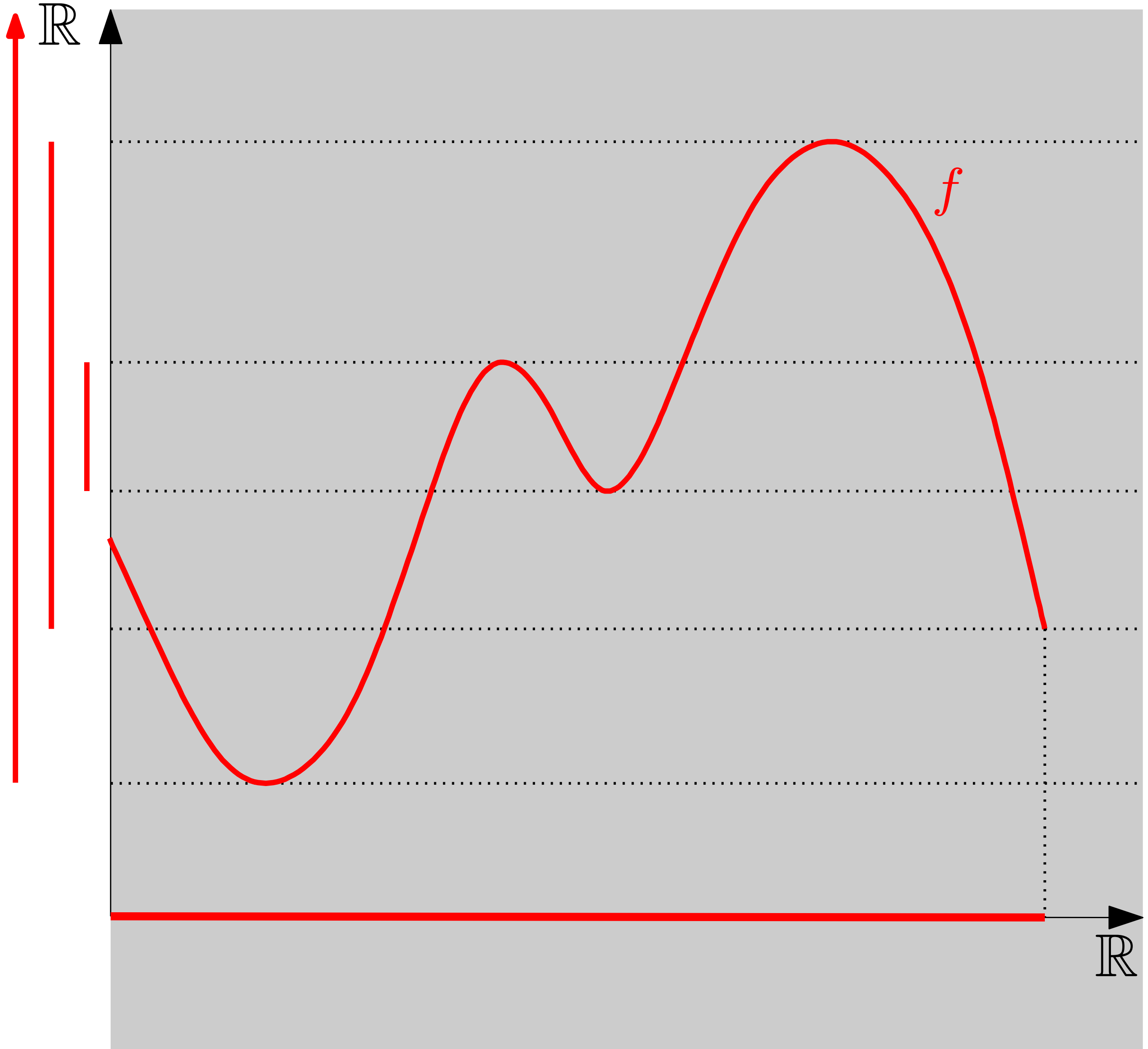
Persistence of sublevel sets of function

H_0 (connected components)



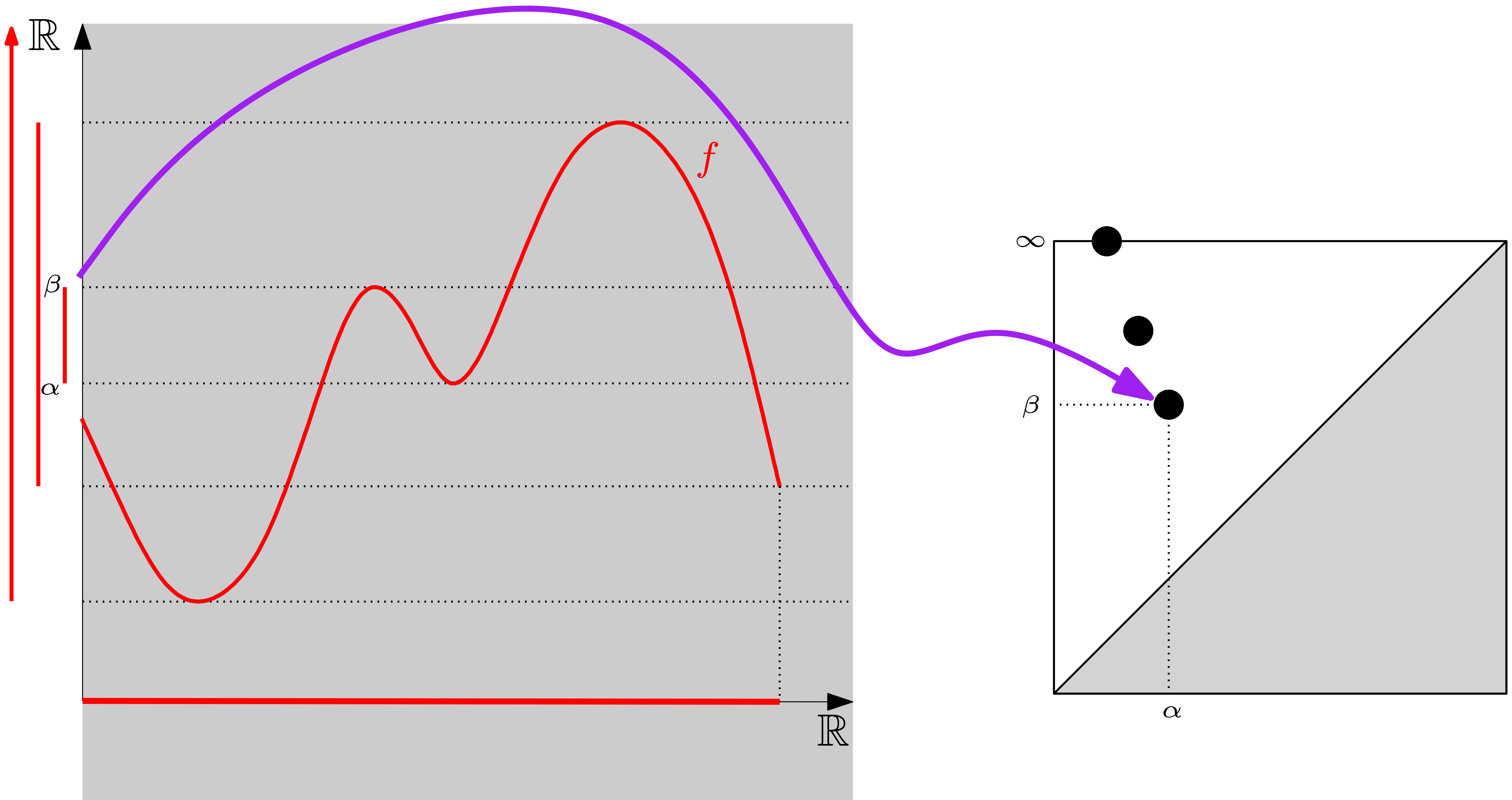
Persistence of sublevel sets of function

H_0 (connected components)

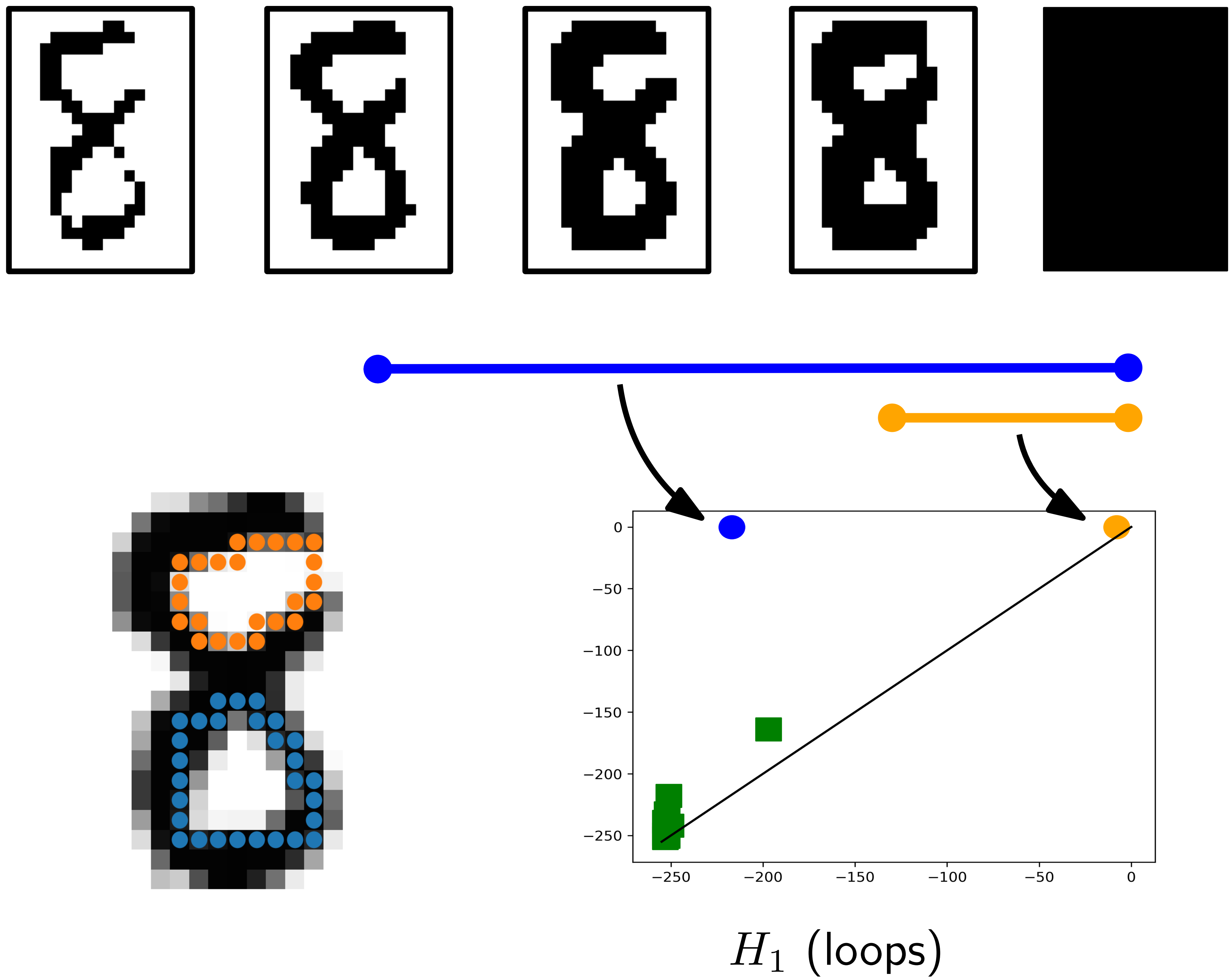


Persistence of sublevel sets of function

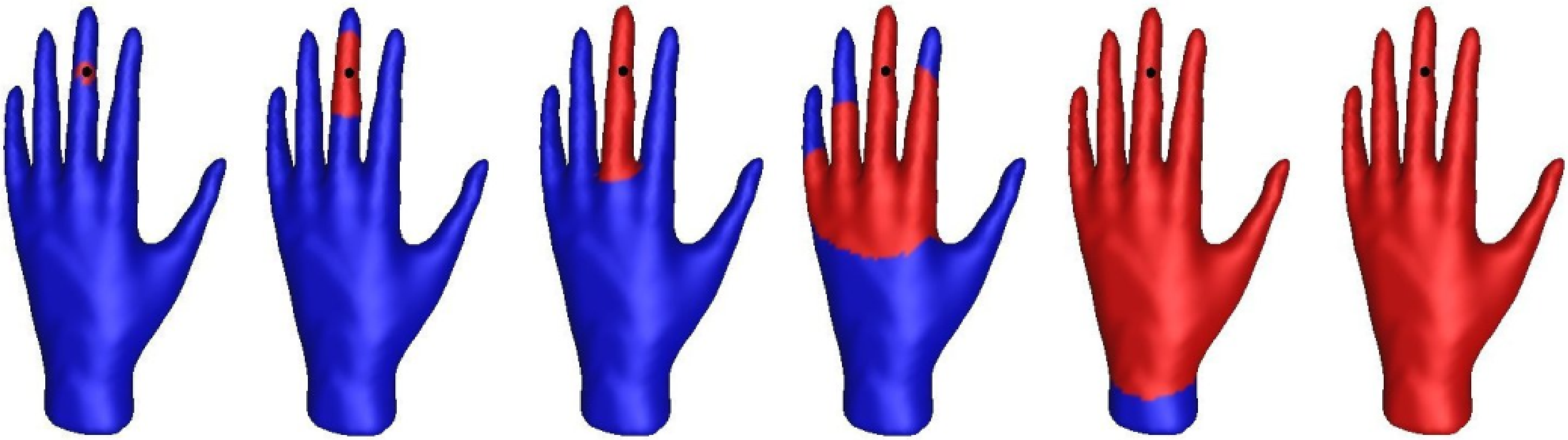
H_0 (connected components)



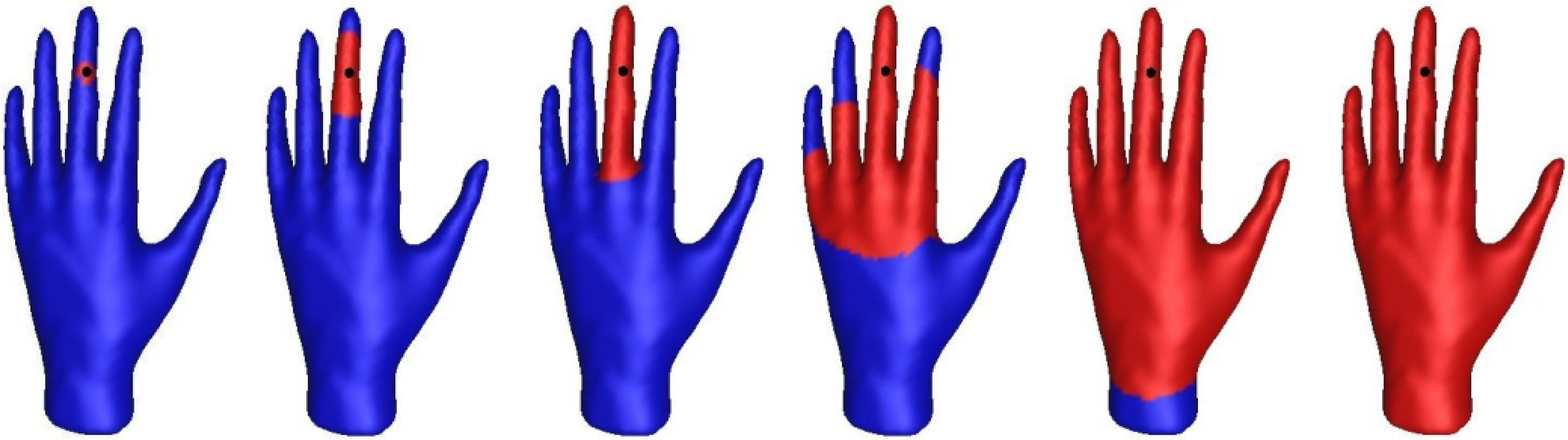
Persistence of images



Persistence of meshes

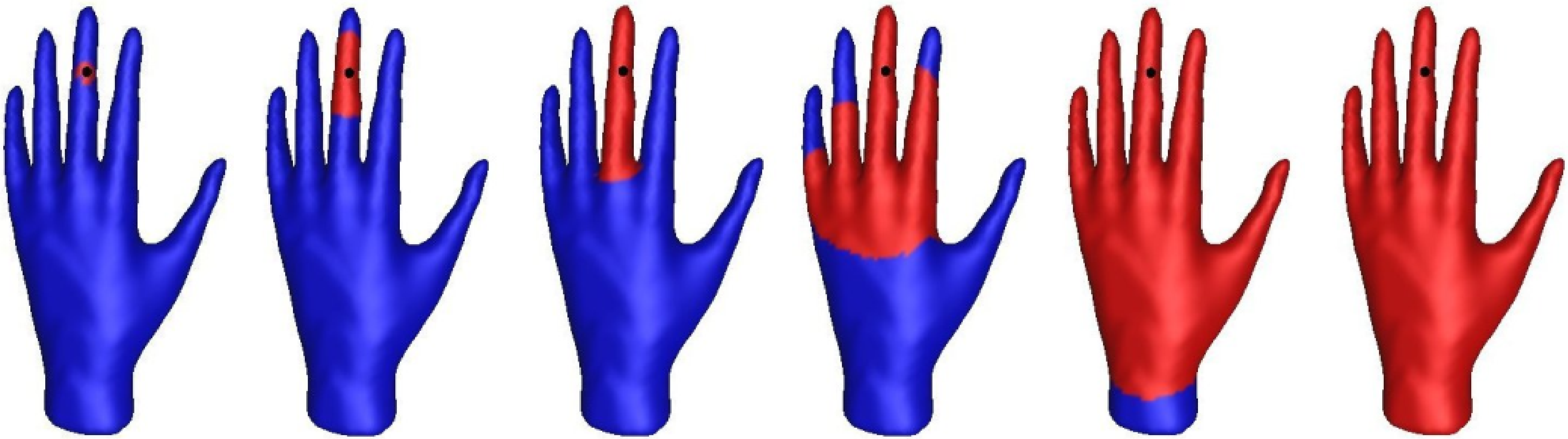


Persistence of meshes

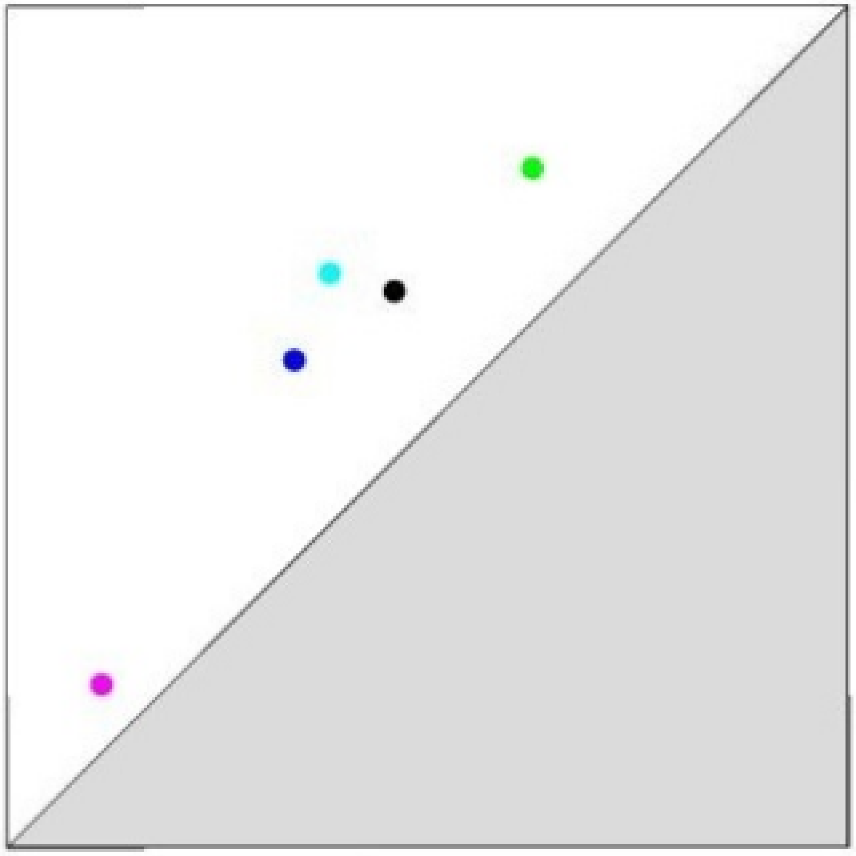


```
In [ ]: st = get_simplex_tree_from_faces(faces)
        filtration = geodesic_distances(base_vertex)
        for v in range(len(vertices)):
            st.assign_filtration([v], filtration[v])
        st.make_filtration_non_decreasing()
        st.persistence()
        dgm = st.persistence_intervals_in_dimension(1)
```

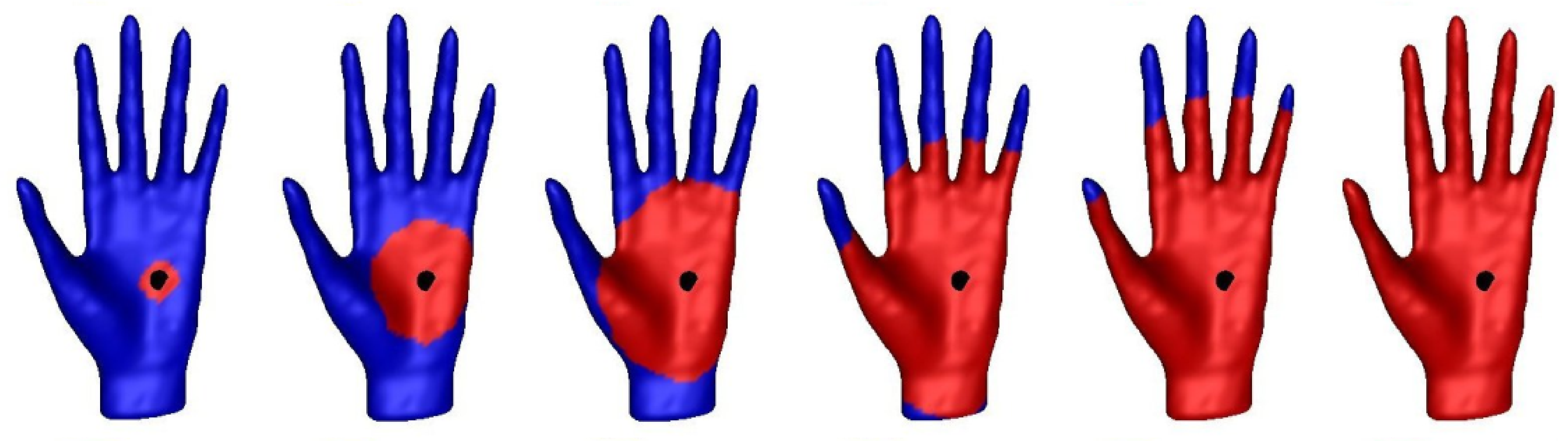
Persistence of meshes



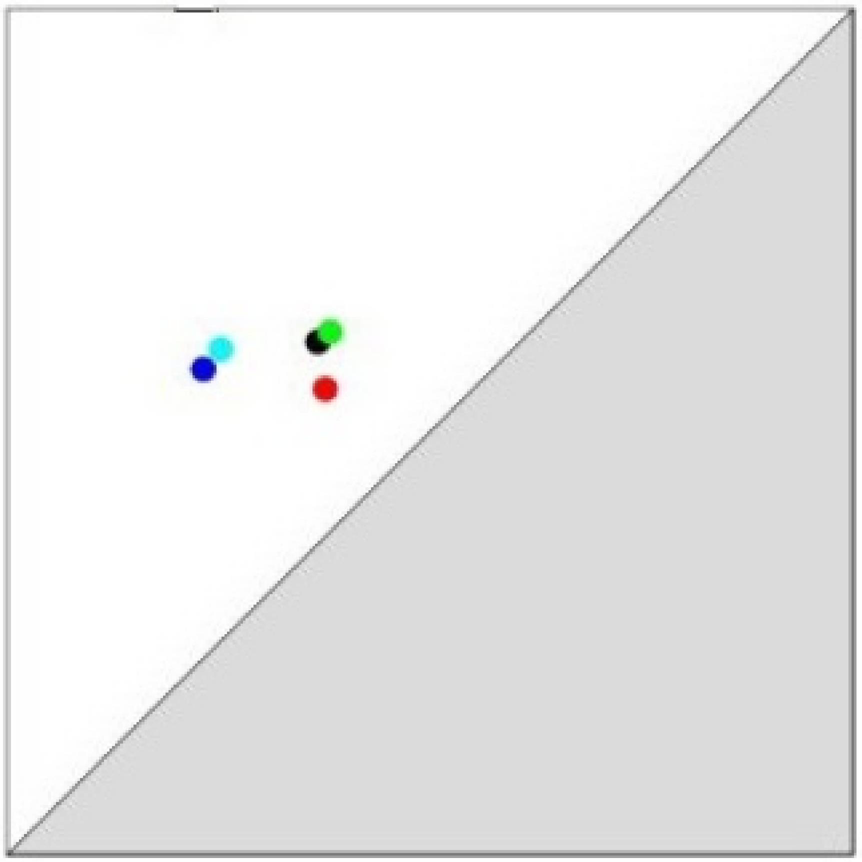
In [ ]:
```
st = get_simplex_tree_from_faces(faces)
filtration = geodesic_distances(base_vertex)
for v in range(len(vertices)):
    st.assign_filtration([v], filtration[v])
st.make_filtration_non_decreasing()
st.persistence()
dgm = st.persistence_intervals_in_dimension(1)
```



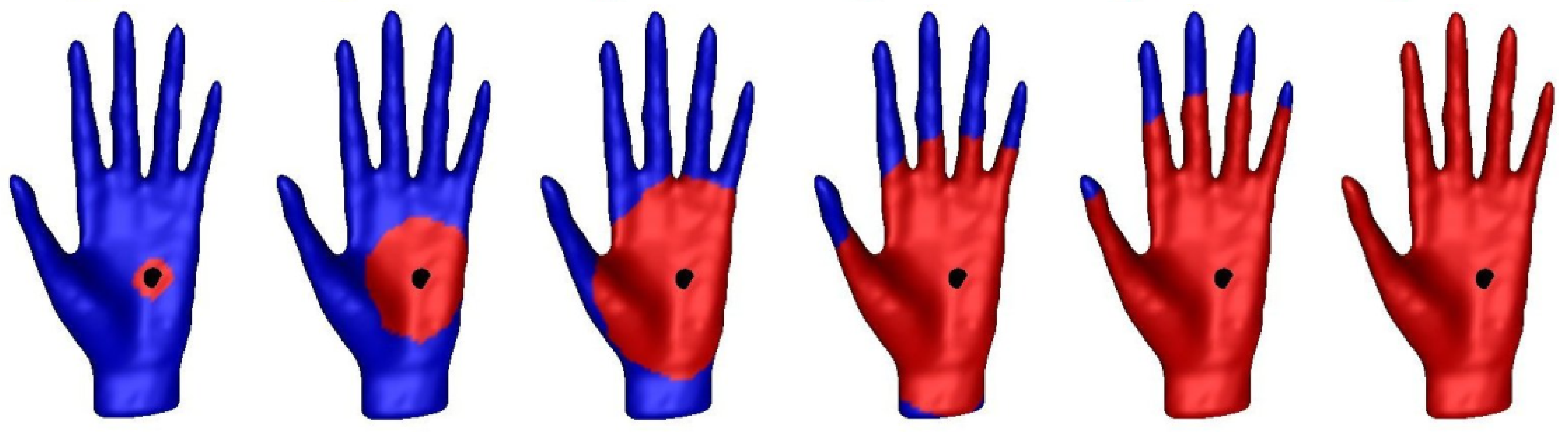
Persistence of meshes



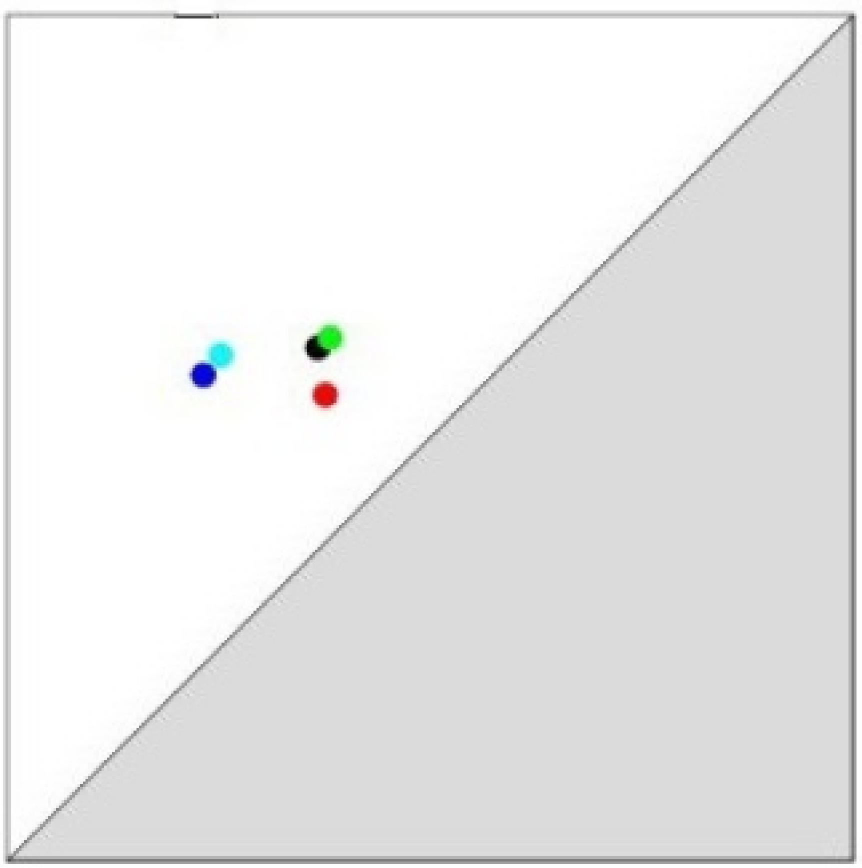
In [ ]:
```
st = get_simplex_tree_from_faces(faces)
filtration = geodesic_distances(base_vertex)
for v in range(len(vertices)):
    st.assign_filtration([v], filtration[v])
st.make_filtration_non_decreasing()
st.persistence()
dgm = st.persistence_intervals_in_dimension(1)
```



Persistence of meshes

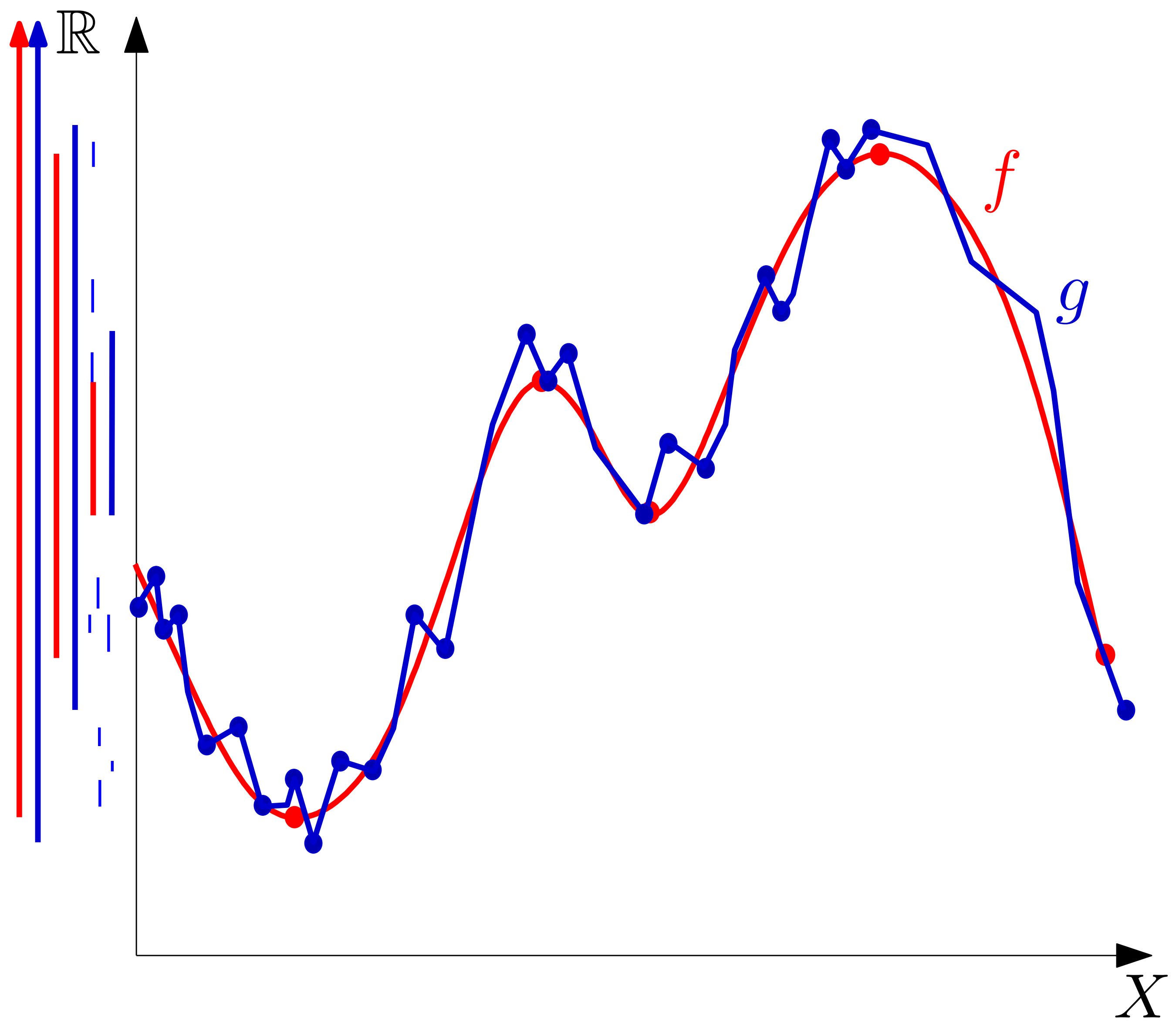


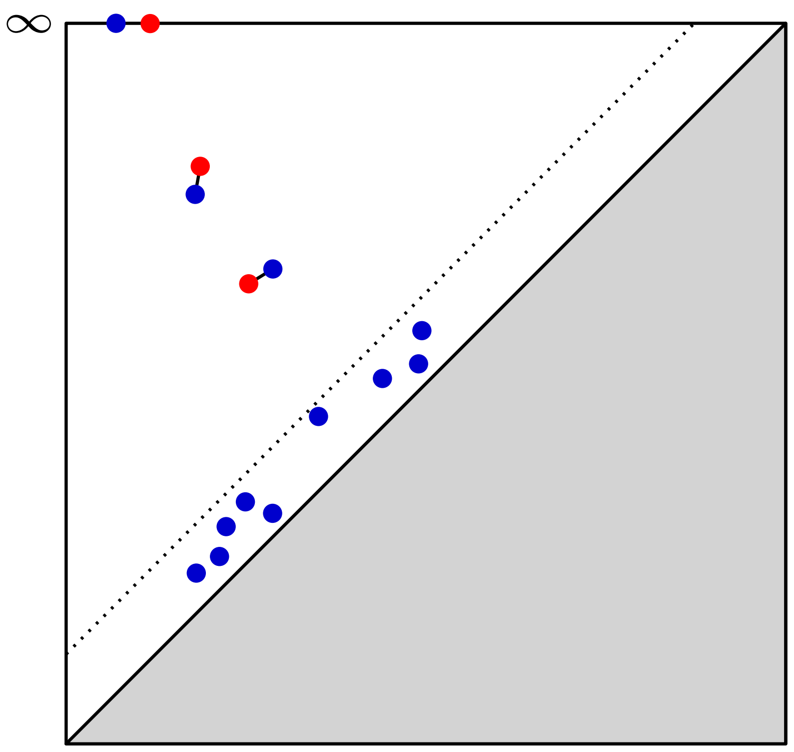
```
In [ ]: st = get_simplex_tree_from_faces(faces)
        filtration = geodesic_distances(base_vertex)
        for v in range(len(vertices)):
            st.assign_filtration([v], filtration[v])
        st.make_filtration_non_decreasing()
        st.persistence()
        dgm = st.persistence_intervals_in_dimension(1)
```



Thm: $d_b(D(f), D(g)) \leq \|f - g\|_\infty$

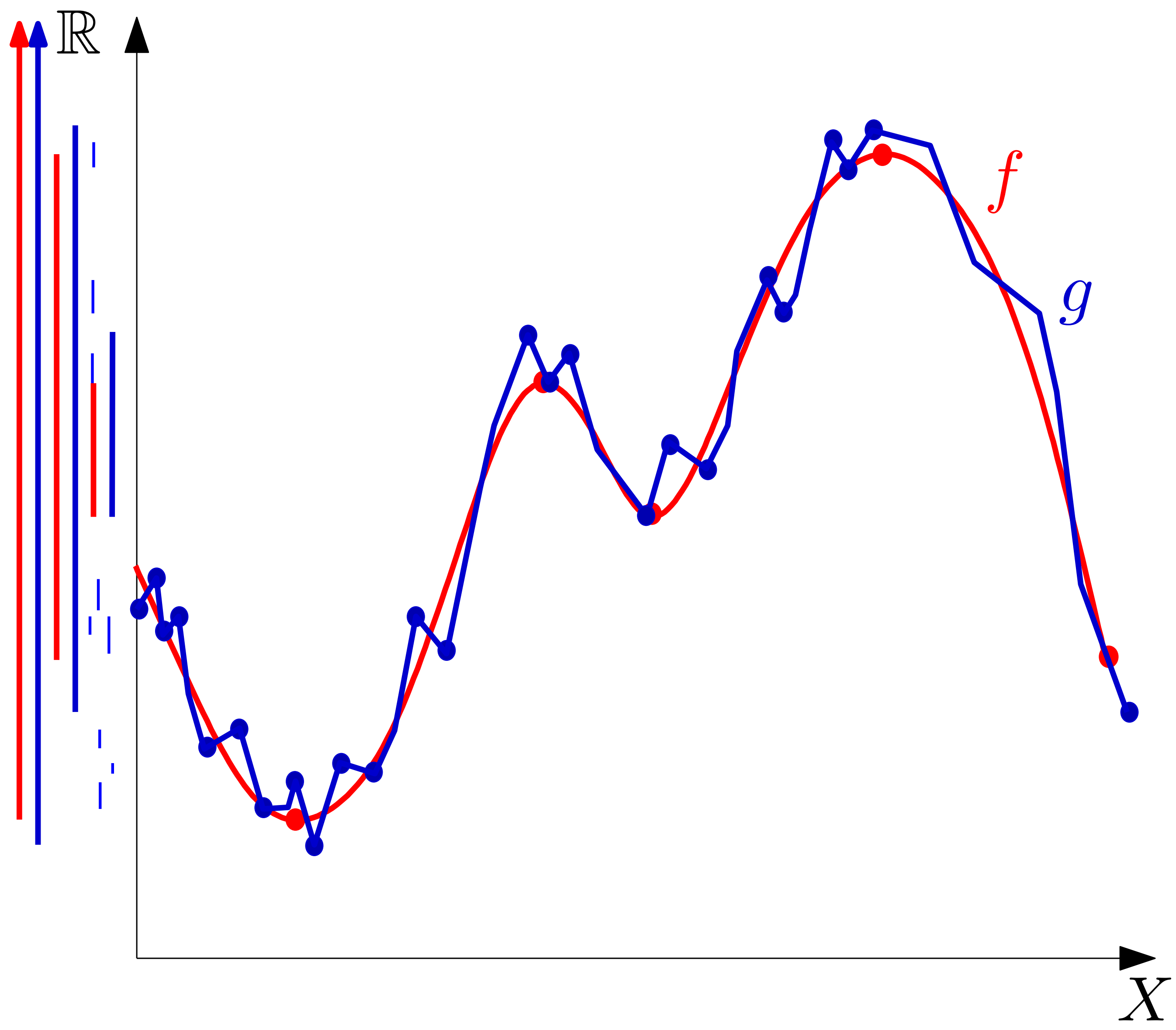
Stability and distance between PDs

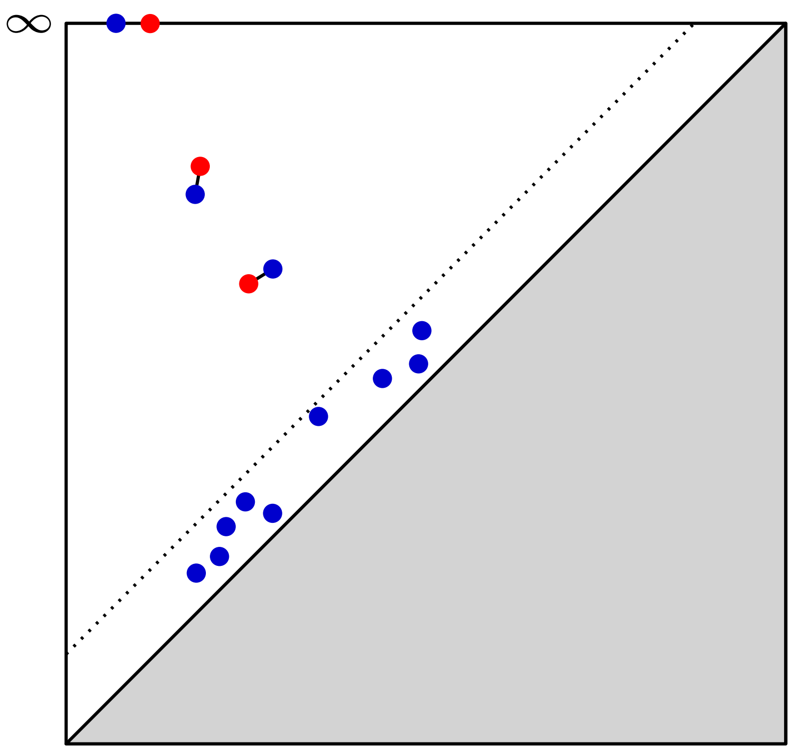




Thm: $d_b(D(f), D(g)) \leq \|f - g\|_\infty$

Stability and distance between PDs





Thm: $d_b(D(f), D(g)) \leq \|f - g\|_\infty$

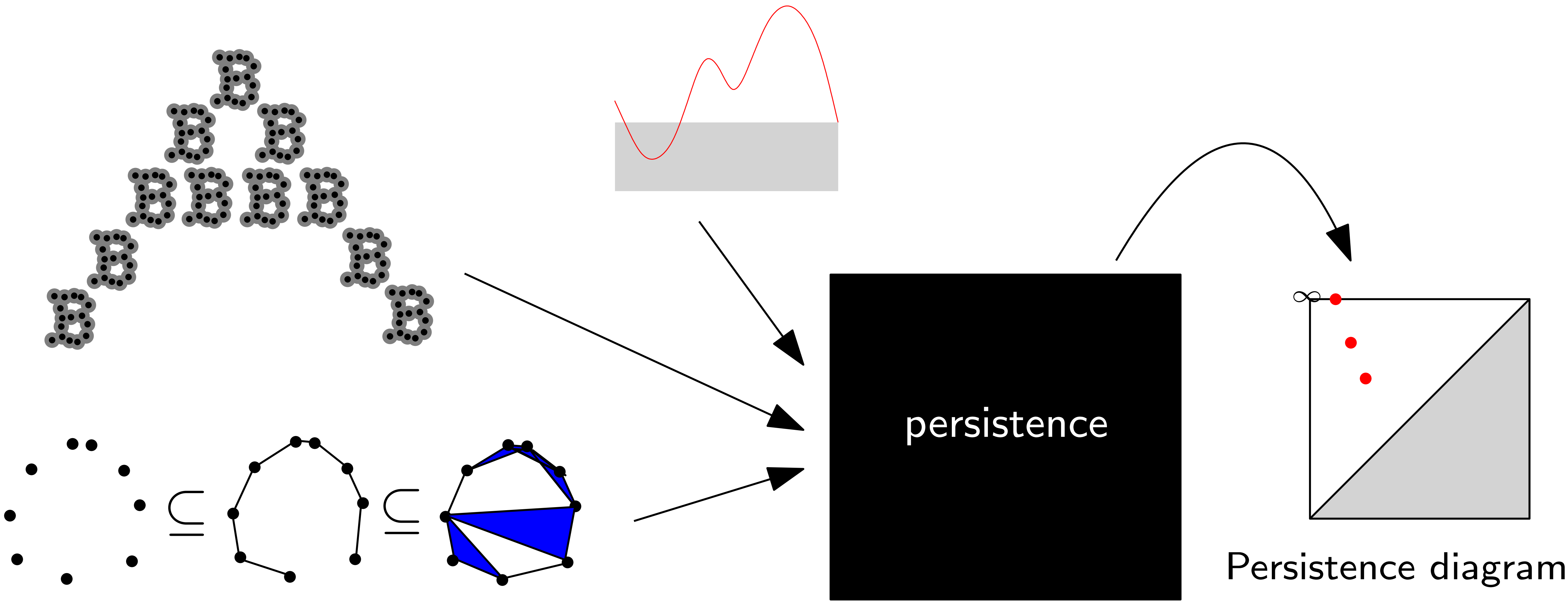
In [ ]:
```
BD = gudhi.representations.BottleneckDistance(epsilon=.001)
BD.fit([st1.persistence_intervals_in_dimension(1)])
bd = BD.transform([st2.persistence_intervals_in_dimension(1)])

WD = gudhi.representations.WassersteinDistance(internal_p=2, order=2)
WD.fit([st1.persistence_intervals_in_dimension(1)])
wd = WD.transform([st2.persistence_intervals_in_dimension(1)])
```

I. Turn datasets into simplicial complexes

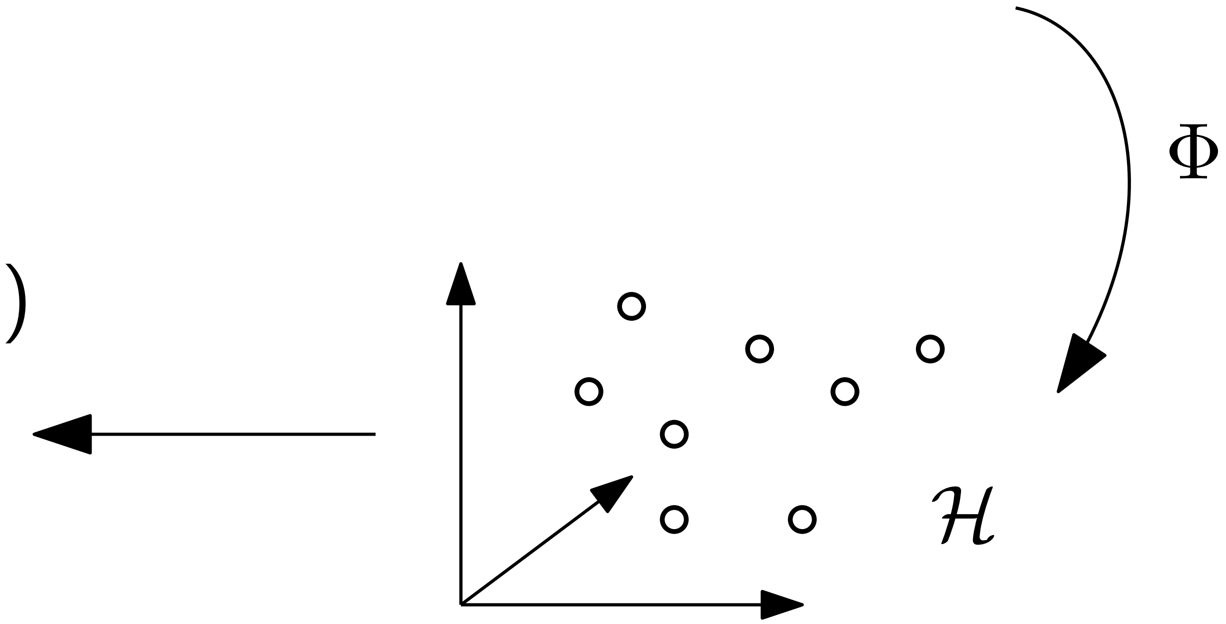
II. Compute and compare persistence diagrams

III. Feed / regularize ML models w/ topology

Persistence diagrams and ML



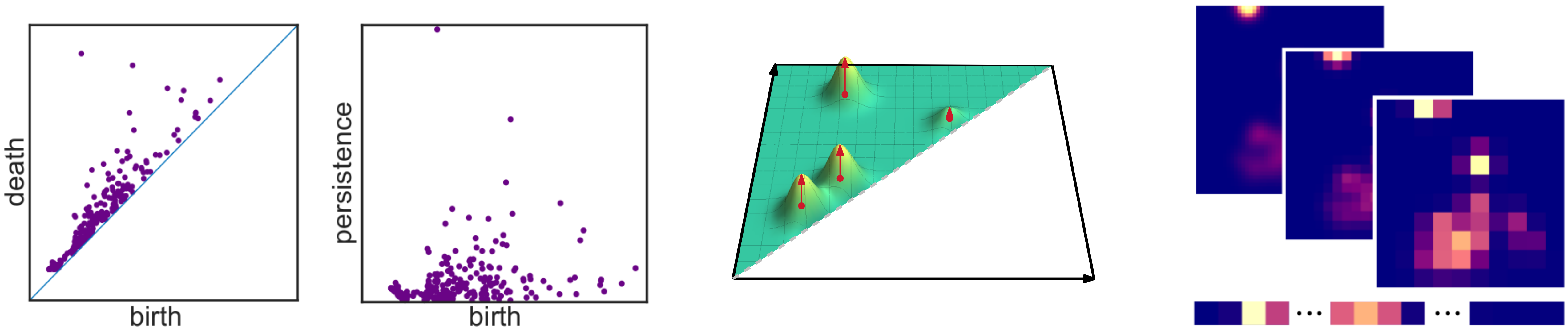
- Classifier (RF, SVM, NN etc.)
- Dim. red. (PCA, MDS, UMAP, t-SNE)
- Clustering (DBSCAN, K-means, etc.)



Etc.

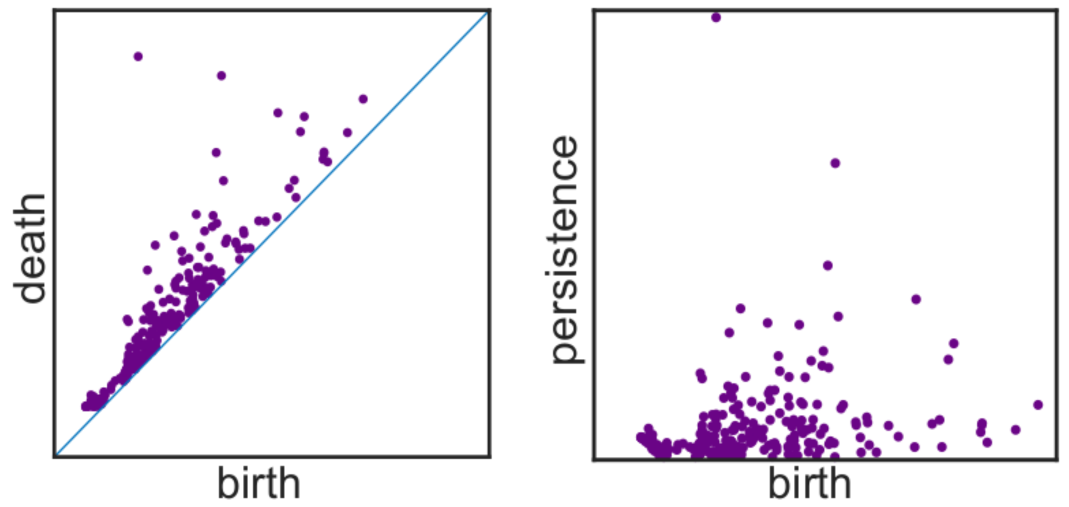
Persistence representations

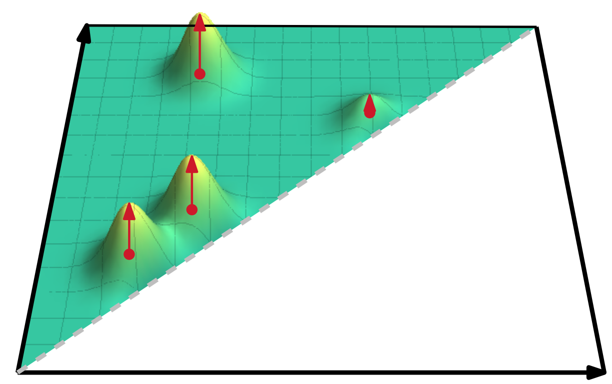
[*Persistence Images: A Stable Vector Representation of Persistent Homology*, Adams et al., JMLR, 2017]

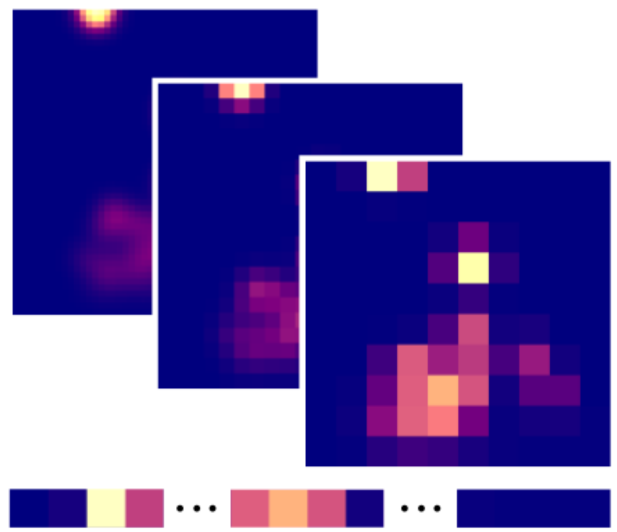


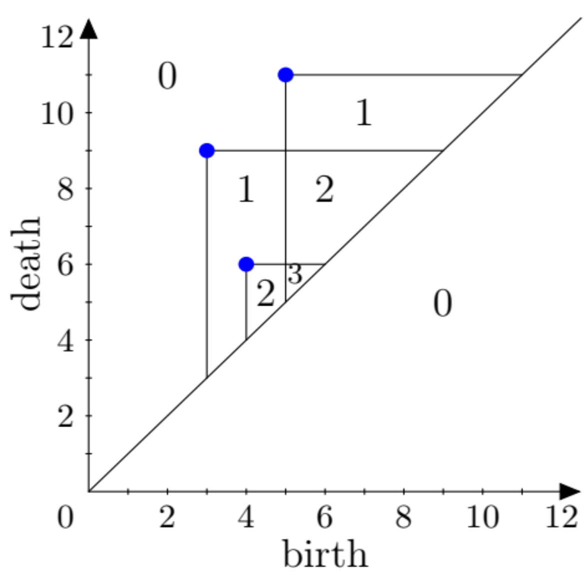
Persistence representations

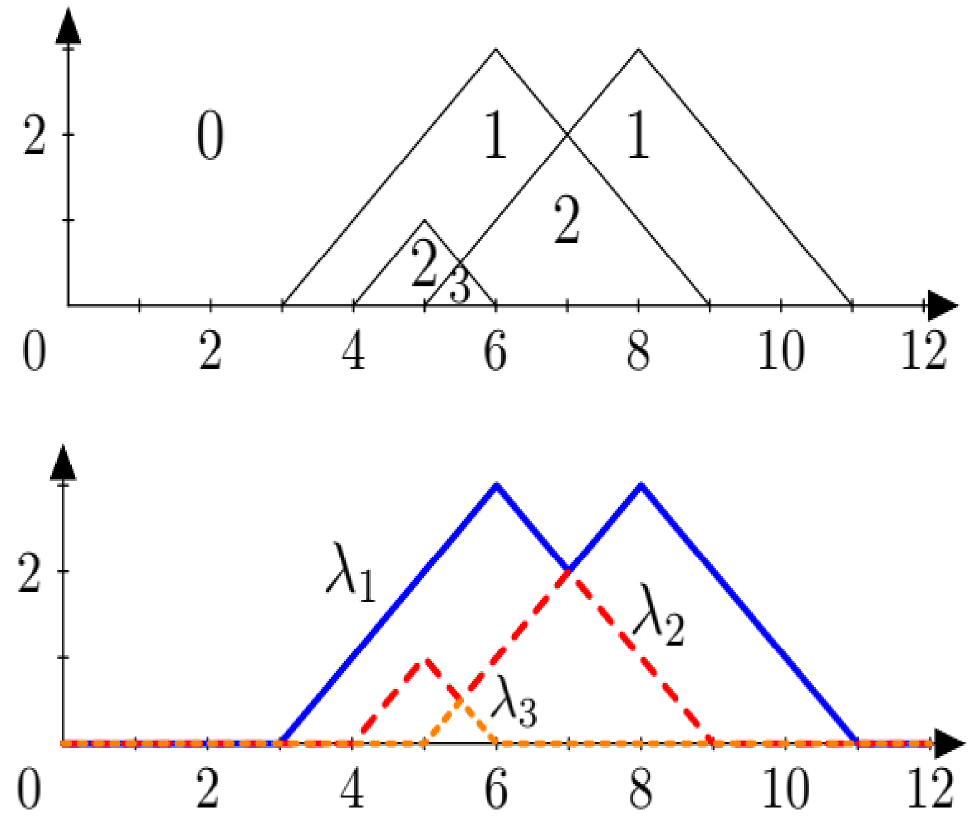
[*Persistence Images: A Stable Vector Representation of Persistent Homology*, Adams et al., JMLR, 2017]

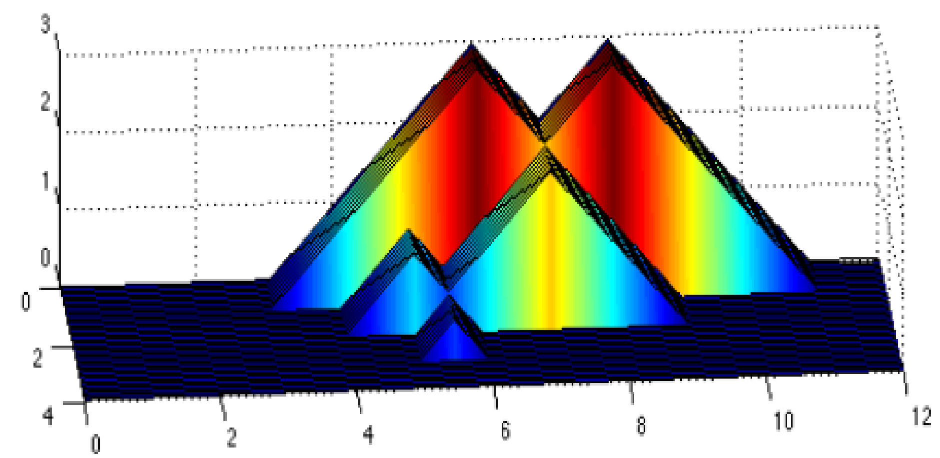








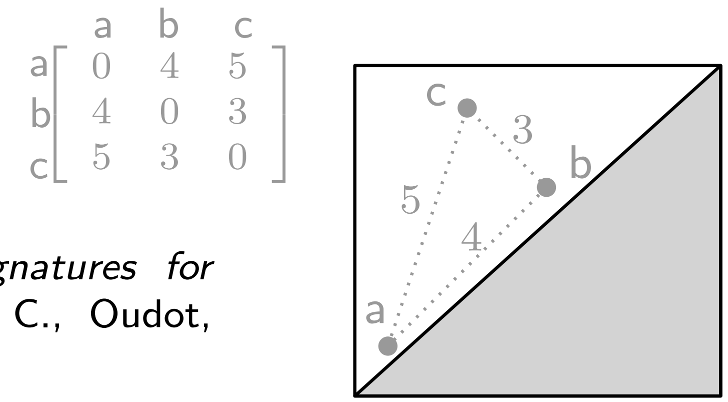


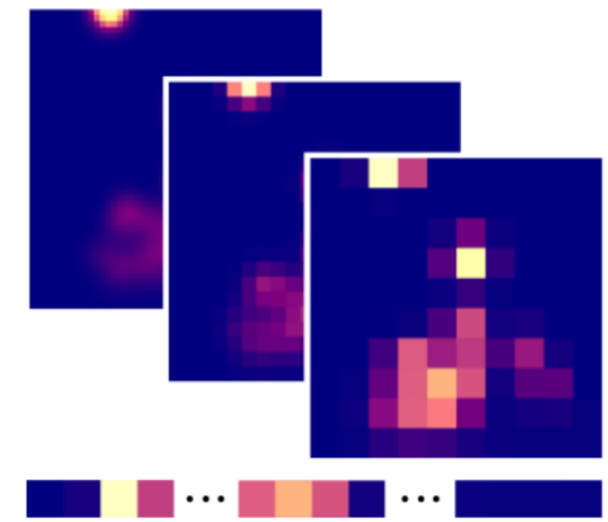


[*Statistical Topological Data Analysis using Persistence Landscapes*, Bubenik, JMLR, 2015]

Persistence representations

- images [*Persistence Images: A Stable Vector Representation of Persistent Homology*, Adams et al., JMLR, 2017]



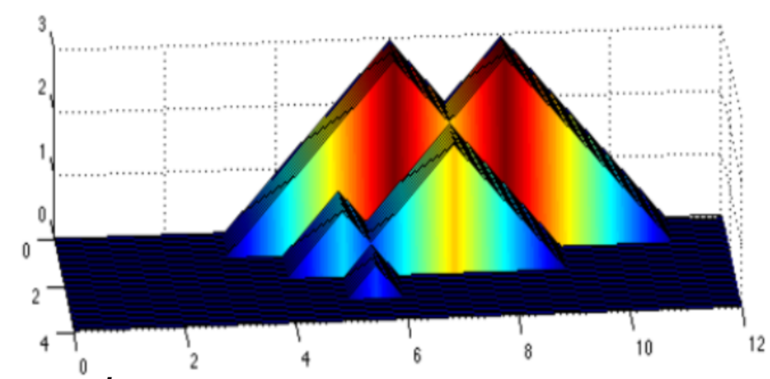


- finite metric spaces [*Stable topological signatures for points on 3D shapes*, C., Oudot, Ovsjanikov, SGP, 2015]

- polynomial roots or evaluations [*Tropical coordinates on the space of persistence barcodes*, Kalisnik, FoCM, 2018]

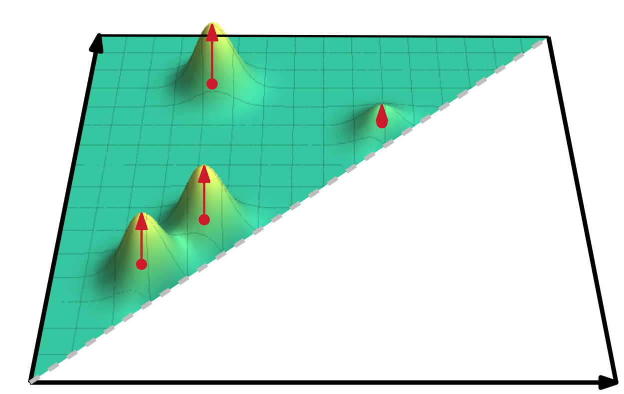
$$\{p_1, \ldots, p_n\} \mapsto (P_1(p_1, \ldots, p_n), \ldots, P_r(p_1, \ldots, p_n), \ldots)$$

- landscapes [*Statistical Topological Data Analysis using Persistence Landscapes*, Bubenik, JMLR, 2015]



- discrete measures:

→ Fisher information [*Persistence Fisher kernel: a Riemannian manifold kernel for persistence diagrams*, Le, Yamada, NeurIPS, 2018]



→ convolution with weighted kernel [*Persistence weighted Gaussian kernel for topological data analysis*, Kusano, Hiraoka, Fukumizu, ICML, 2016]

→ heat diffusion [*A stable multi-scale kernel for topological machine learning*, Reininghaus et al., CVPR, 2015]

→ optimal transport [*Sliced Wasserstein distance between PDs*, C., Cuturi, Oudot, ICML, 2017]

Persistence representations

In [20]:

```
from sklearn.preprocessing   import MinMaxScaler
from sklearn.pipeline        import Pipeline
from sklearn.svm             import SVC
from sklearn.ensemble        import RandomForestClassifier
from sklearn.neighbors       import KNeighborsClassifier

# Definition of pipeline
pipe = Pipeline([("Separator", gd.representations.DiagramSelector(limit=np.inf, point_type="finite")),
                 ("Scaler",    gd.representations.DiagramScaler(scalers=[([0,1], MinMaxScaler())])),
                 ("TDA",       gd.representations.PersistenceImage()),
                 ("Estimator", SVC())])

# Parameters of pipeline. This is the place where you specify the methods you want to use to handle diagrams
param =    [{"Scaler__use":         [False],
             "TDA":                 [gd.representations.SlicedWassersteinKernel()],
             "TDA__bandwidth":      [0.1, 1.0],
             "TDA__num_directions": [20],
             "Estimator":           [SVC(kernel="precomputed", gamma="auto")]},

            {"Scaler__use":         [False],
             "TDA":                 [gd.representations.PersistenceWeightedGaussianKernel()],
             "TDA__bandwidth":      [0.1, 0.01],
             "TDA__weight":         [lambda x: np.arctan(x[1]-x[0])],
             "Estimator":           [SVC(kernel="precomputed", gamma="auto")]},

            {"Scaler__use":         [True],
             "TDA":                 [gd.representations.PersistenceImage()],
             "TDA__resolution":     [ [5,5], [6,6] ],
             "TDA__bandwidth":      [0.01, 0.1, 1.0, 10.0],
             "Estimator":           [SVC()]},

            {"Scaler__use":         [True],
             "TDA":                 [gd.representations.Landscape()],
             "TDA__resolution":     [100],
             "Estimator":           [RandomForestClassifier()]},

            {"Scaler__use":         [False],
             "TDA":                 [gd.representations.BottleneckDistance()],
             "TDA__epsilon":        [0.1],
             "Estimator":           [KNeighborsClassifier(metric="precomputed")]}
           ]
```

Learn persistence representations

Learn persistence representations

[*PersLay: A Neural Network Layer for Persistence Diagrams and New Graph Topological Signatures*, C., Chazal, Ike, Lacombe, Royer, Umeda, AISTATS, 2019]

$$\mathrm{PersLay}(D) = \rho\left(\mathsf{op}\{w(p) \cdot \phi(p)\}_{p \in D}\right)$$

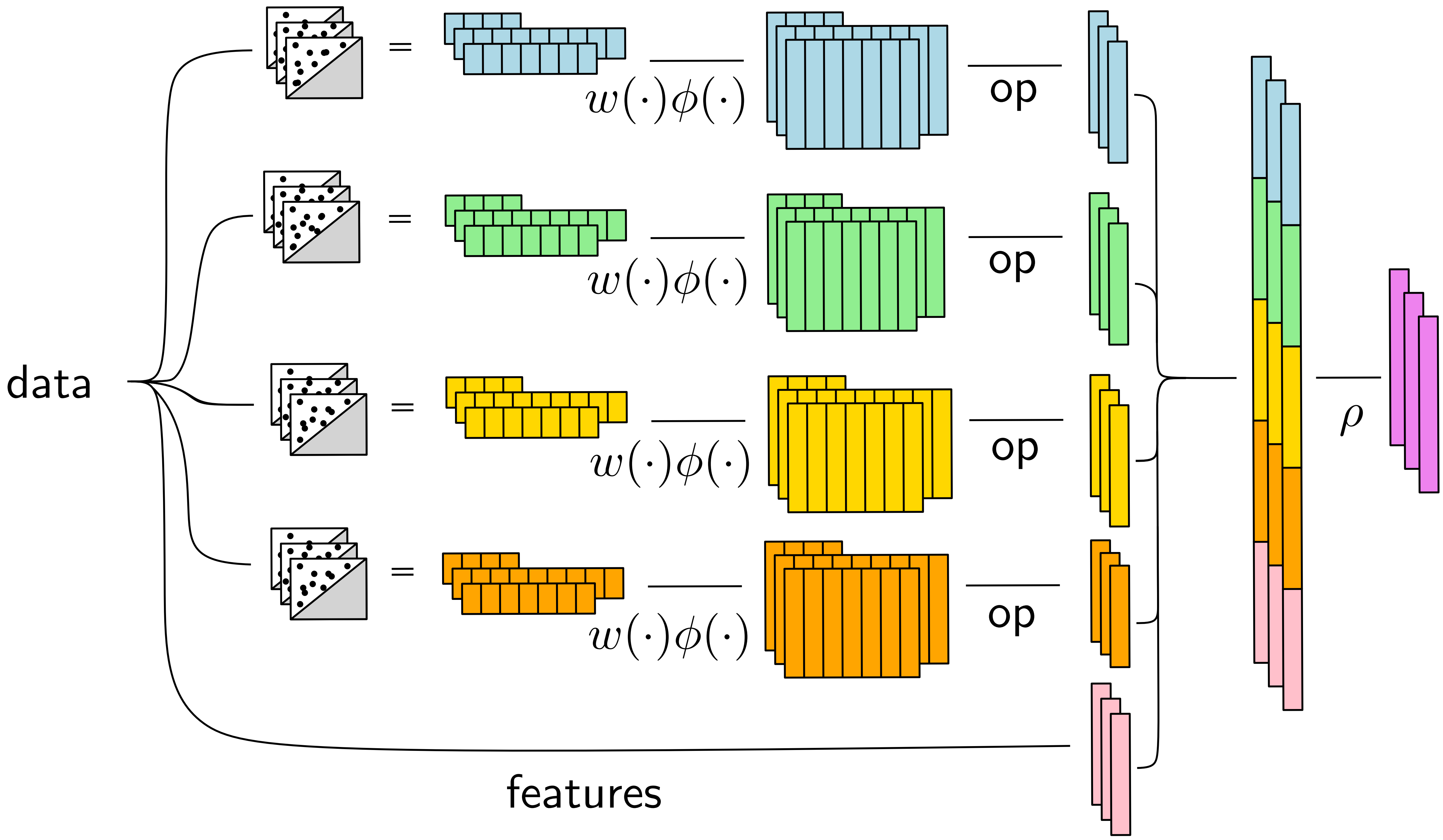
Permutation-invariant operation

Weight function

Point transformation

Learn persistence representations

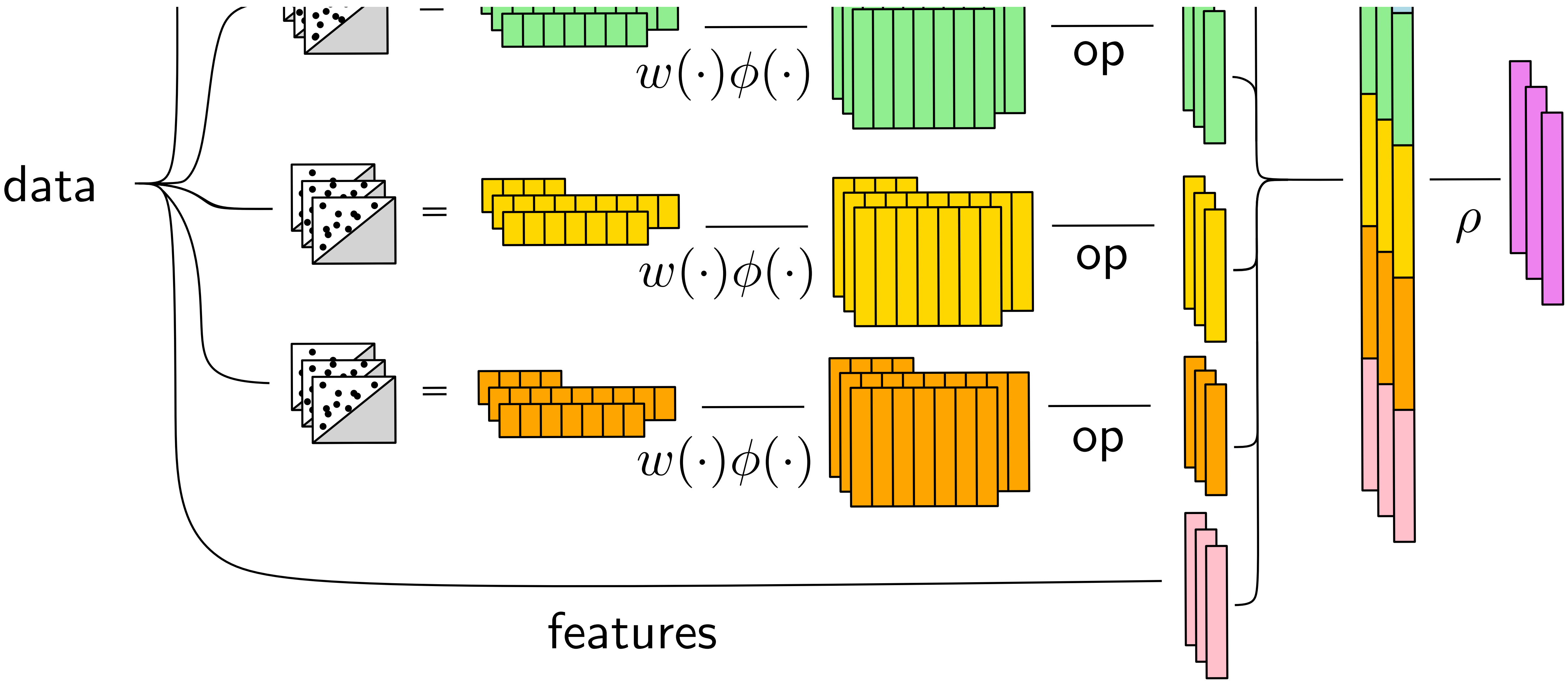
[*PersLay: A Neural Network Layer for Persistence Diagrams and New Graph Topological Signatures*, C., Chazal, Ike, Lacombe, Royer, Umeda, AISTATS, 2019]



Learn persistence representations

[*PersLay: A Neural Network Layer for Persistence Diagrams and New Graph Topological Signatures*, C., Chazal, Ike, Lacombe, Royer, Umeda, AISTATS, 2019]

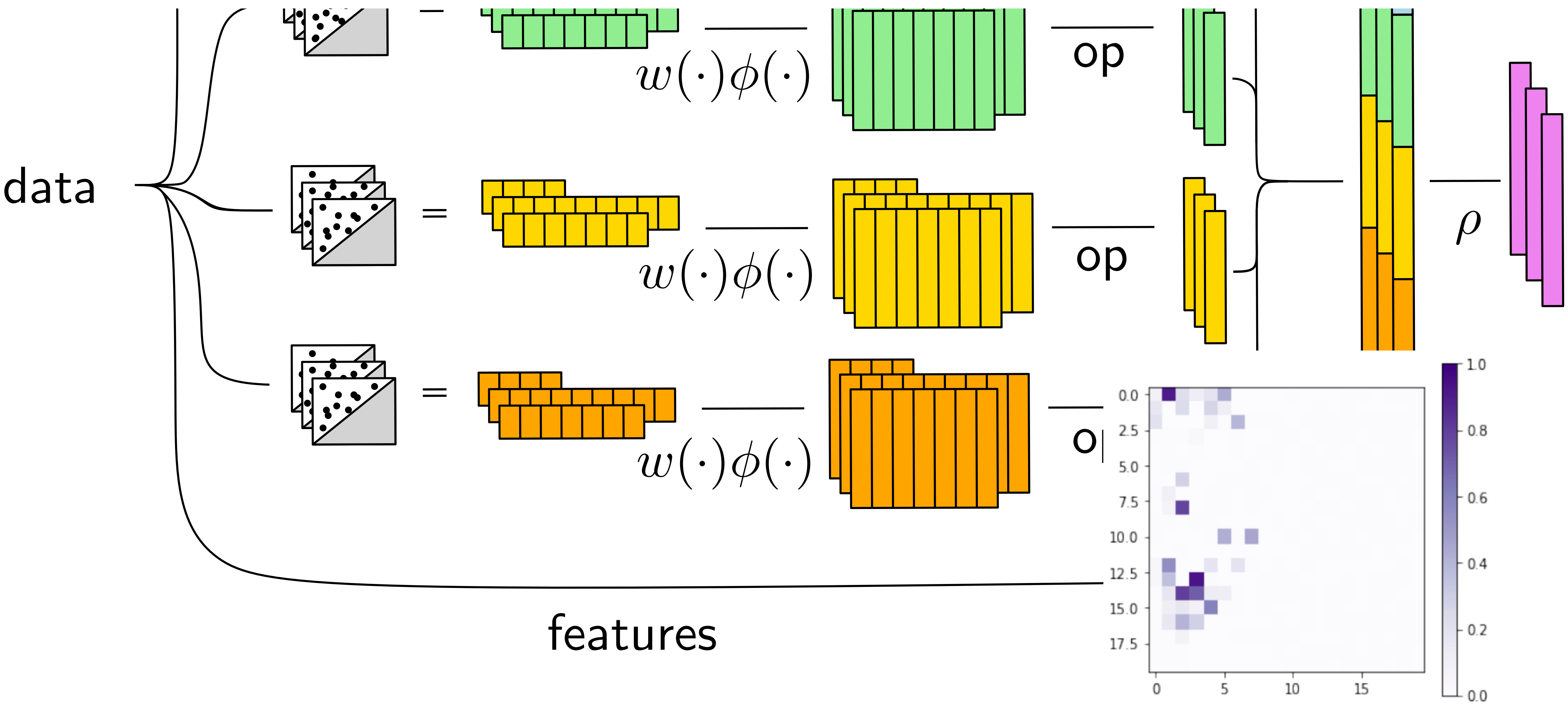
```
In [ ]: rho = tensorflow.identity
        phi = gudhi.tensorflow.perslay.GaussianPerslayPhi((100, 100), ((-.5, 1.5), (-.5, 1.5)), .1)
        weight = gudhi.tensorflow.perslay.GridPerslayWeight(np.array(
                    numpy.random.uniform(size=[100,100]),dtype=np.float32),((-0.01, 1.01),(-0.01, 1.01)))
        perm_op = tensorflow.math.reduce_sum

        perslay = gudhi.tensorflow.perslay.Perslay(phi=phi, weight=weight, perm_op=perm_op, rho=rho)
        vectors = perslay(diagrams)
```



Learn persistence representations

[*PersLay: A Neural Network Layer for Persistence Diagrams and New Graph Topological Signatures*, C., Chazal, Ike, Lacombe, Royer, Umeda, AISTATS, 2019]

In [ ]:
```
rho = tensorflow.identity
phi = gudhi.tensorflow.perslay.GaussianPerslayPhi((100, 100), ((-.5, 1.5), (-.5, 1.5)), .1)
weight = gudhi.tensorflow.perslay.GridPerslayWeight(np.array(
            numpy.random.uniform(size=[100,100]),dtype=np.float32),((-0.01, 1.01),(-0.01, 1.01)))
perm_op = tensorflow.math.reduce_sum

perslay = gudhi.tensorflow.perslay.Perslay(phi=phi, weight=weight, perm_op=perm_op, rho=rho)
vectors = perslay(diagrams)
```



Weight function learnt

Learn filtrations

Learn filtrations

A persistence diagram D is made of all $(\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in \mathbb{R}^2$ where σ_+ (resp. σ_-) is positive (resp. negative), and $\mathcal{F}$ is the filtration function.

Thus we can define the gradient of a point $p = (\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in D$ as

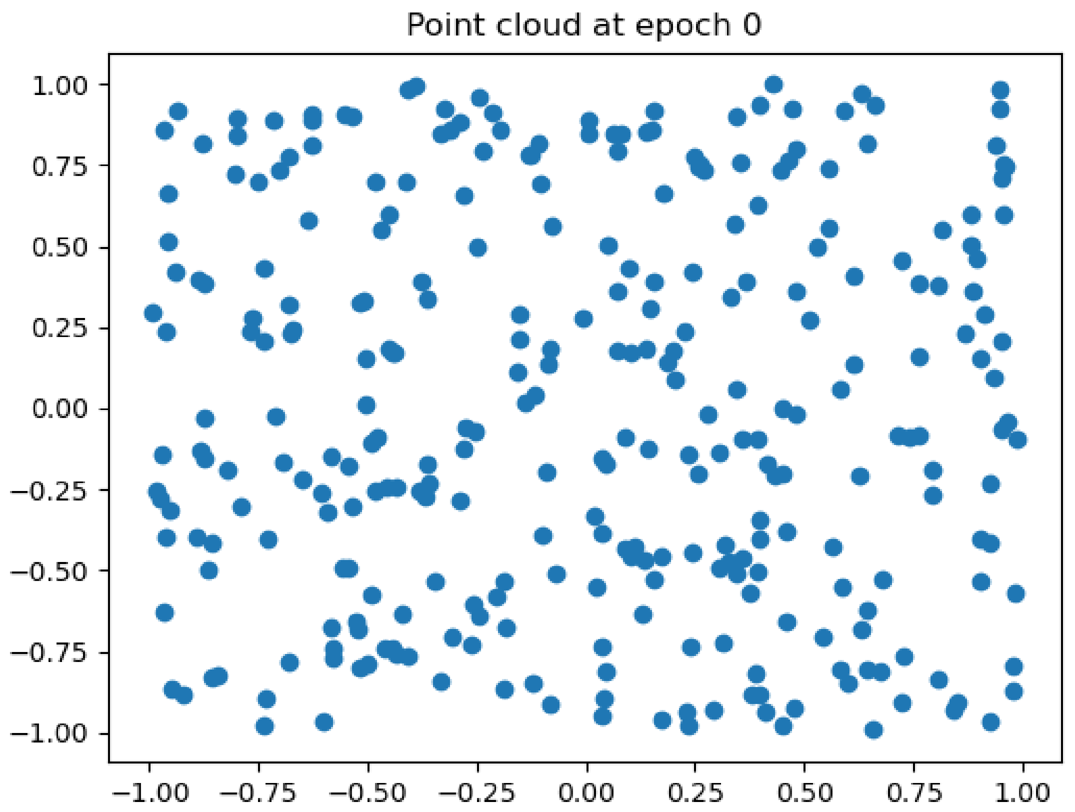
$$\nabla p = [\nabla \mathcal{F}(\sigma_+), \nabla \mathcal{F}(\sigma_-)]$$

Learn filtrations

A persistence diagram D is made of all $(\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in \mathbb{R}^2$ where σ_+ (resp. σ_-) is positive (resp. negative), and $\mathcal{F}$ is the filtration function.

Thus we can define the gradient of a point $p = (\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in D$ as

$$\nabla p = [\nabla \mathcal{F}(\sigma_+), \nabla \mathcal{F}(\sigma_-)]$$



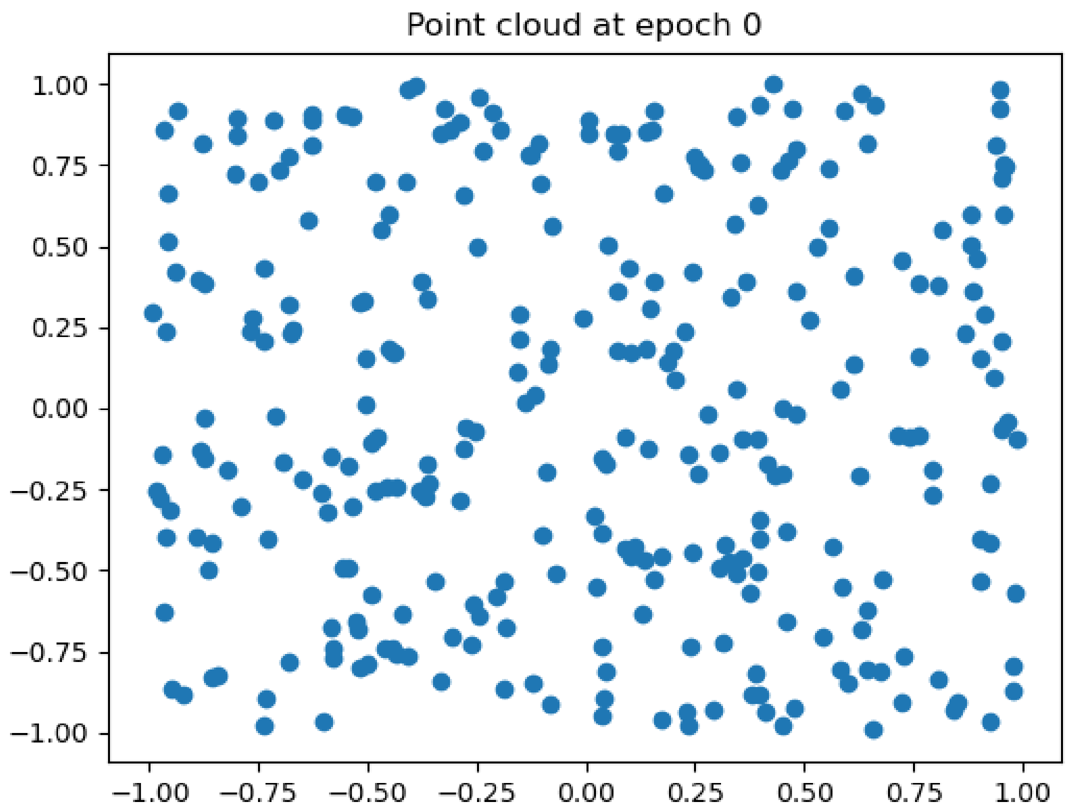
Ex: VR filtration parametrized by a point cloud

Learn filtrations

A persistence diagram D is made of all $(\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in \mathbb{R}^2$ where σ_+ (resp. σ_-) is positive (resp. negative), and $\mathcal{F}$ is the filtration function.

Thus we can define the gradient of a point $p = (\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in D$ as

$$\nabla p = [\nabla \mathcal{F}(\sigma_+), \nabla \mathcal{F}(\sigma_-)]$$



In [ ]:
```
optimizer = tensorflow.keras.optimizers.SGD(learning_rate=0.1)
for epoch in tqdm(range(200+1)):

    with tensorflow.GradientTape() as tape:
        dgm = gudhi.tensorflow.RipsLayer(maximum_edge_length=1., homology_dimensions=[1]).call(X)[0][0]
        # Opposite of the squared distances to the diagonal
        persistence_loss = -tensorflow.math.reduce_sum(tf.square(.5*(dgm[:,1]-dgm[:,0])))
        # Unit square regularization
        regularization = tensorflow.reduce_sum(tf.maximum(tf.abs(X)-1, 0))
        loss = persistence_loss + regularization
    gradients = tape.gradient(loss, [X])

    # We also apply a small random noise to the gradient to ensure convergence
    np.random.seed(epoch)
    gradients[0] = gradients[0] + numpy.random.normal(loc=0., scale=.001, size=gradients[0].shape)

    optimizer.apply_gradients(zip(gradients, [X]))
```

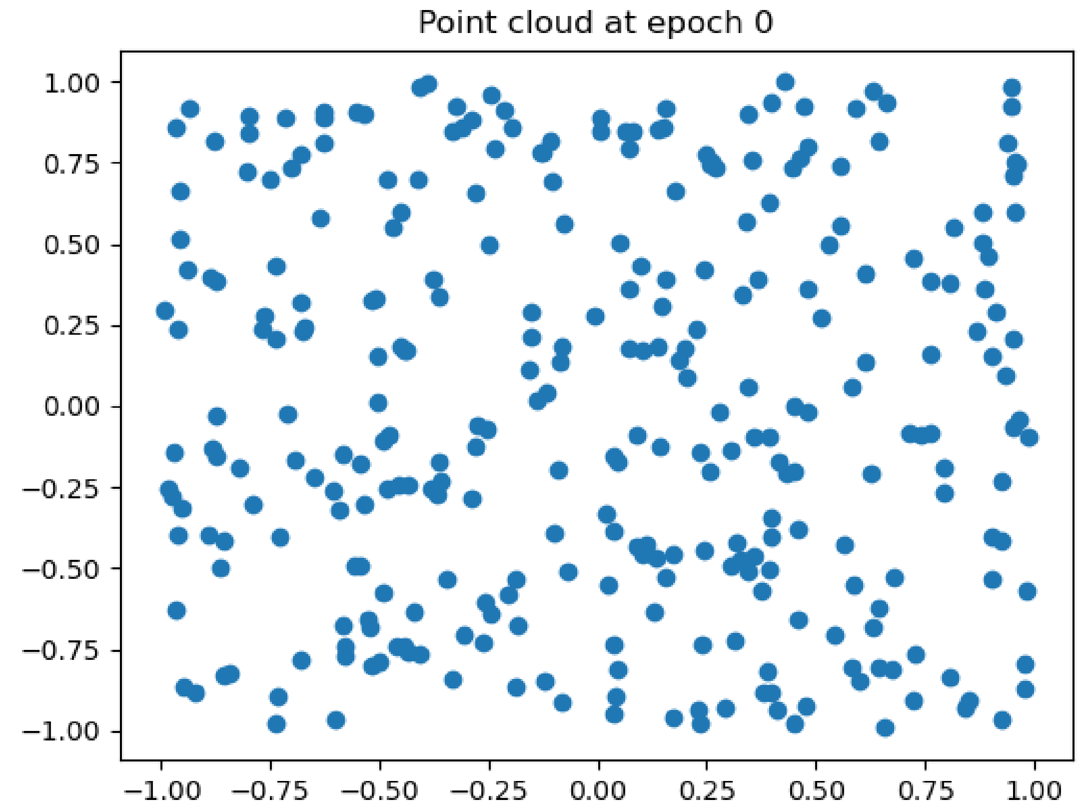
Learn filtrations

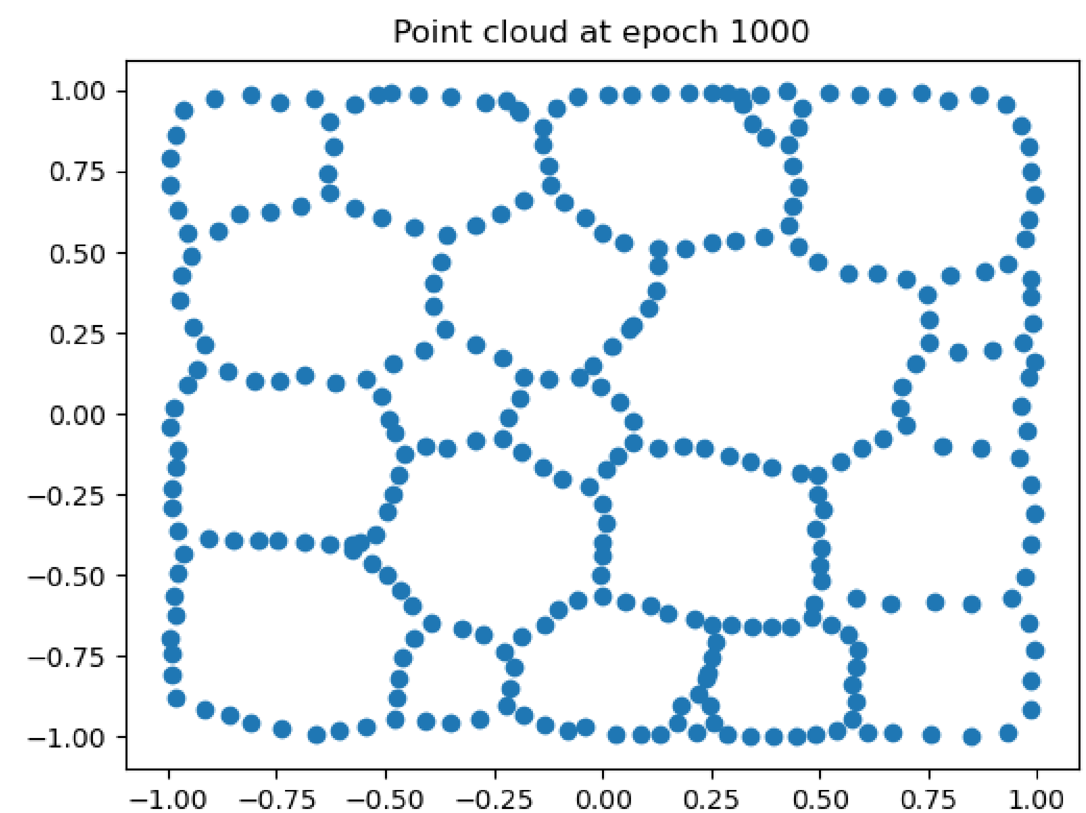
A persistence diagram D is made of all $(\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in \mathbb{R}^2$ where σ_+ (resp. σ_-) is positive (resp. negative), and $\mathcal{F}$ is the filtration function.

Thus we can define the gradient of a point $p = (\mathcal{F}(\sigma_+), \mathcal{F}(\sigma_-)) \in D$ as

$$\nabla p = [\nabla \mathcal{F}(\sigma_+), \nabla \mathcal{F}(\sigma_-)]$$

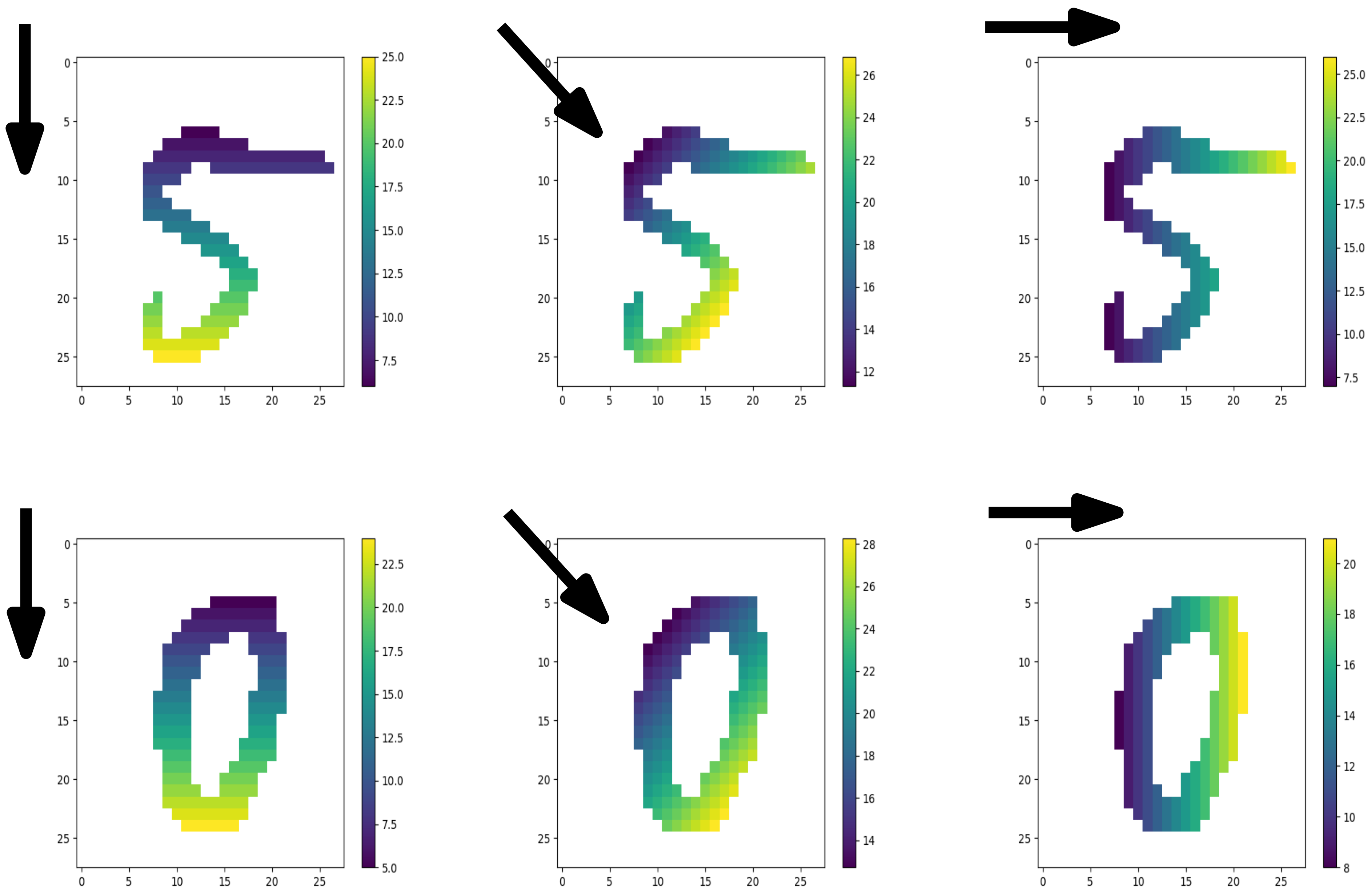
In [ ]:
```
optimizer = tensorflow.keras.optimizers.SGD(learning_rate=0.1)
for epoch in tqdm(range(200+1)):

    with tensorflow.GradientTape() as tape:
        dgm = gudhi.tensorflow.RipsLayer(maximum_edge_length=1
        # Opposite of the squared distances to the diagonal
        persistence_loss = -tensorflow.math.reduce_sum(tf.squa
        # Unit square regularization
        regularization = tensorflow.reduce_sum(tf.maximum(tf.al
        loss = persistence_loss + regularization
    gradients = tape.gradient(loss, [X])

    # We also apply a small random noise to the gradient to en
    np.random.seed(epoch)
    gradients[0] = gradients[0] + numpy.random.normal(loc=0.,

    optimizer.apply_gradients(zip(gradients, [X]))
```





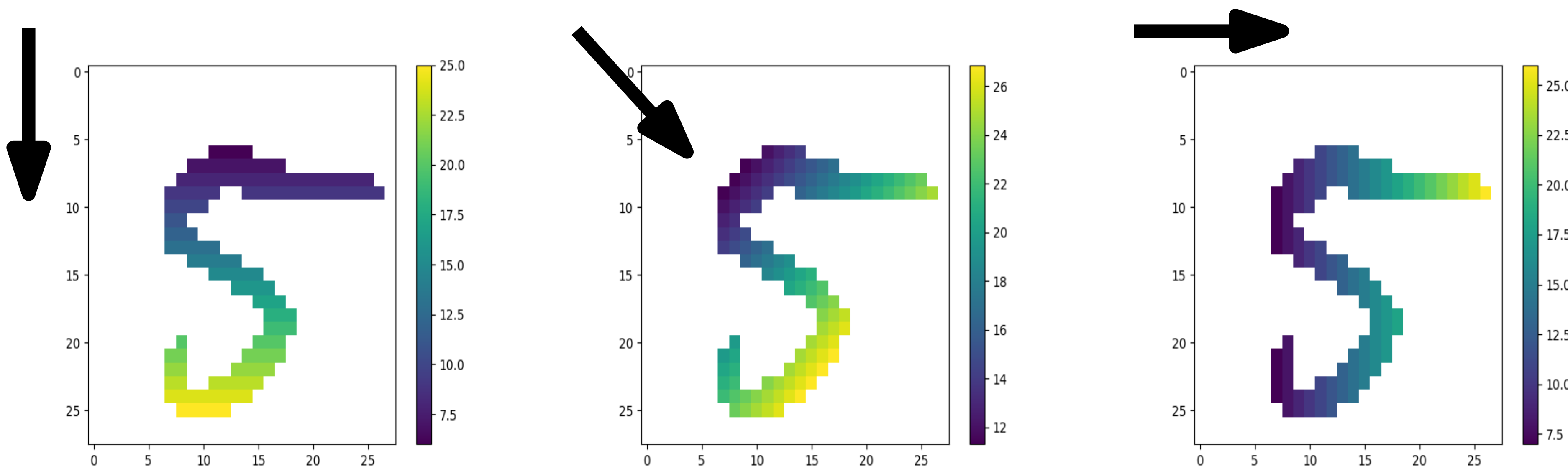
Learn filtrations

Ex: images filtered by a direction parameterized by angle



Learn filtrations

Ex: images filtered by a direction parameterized by angle

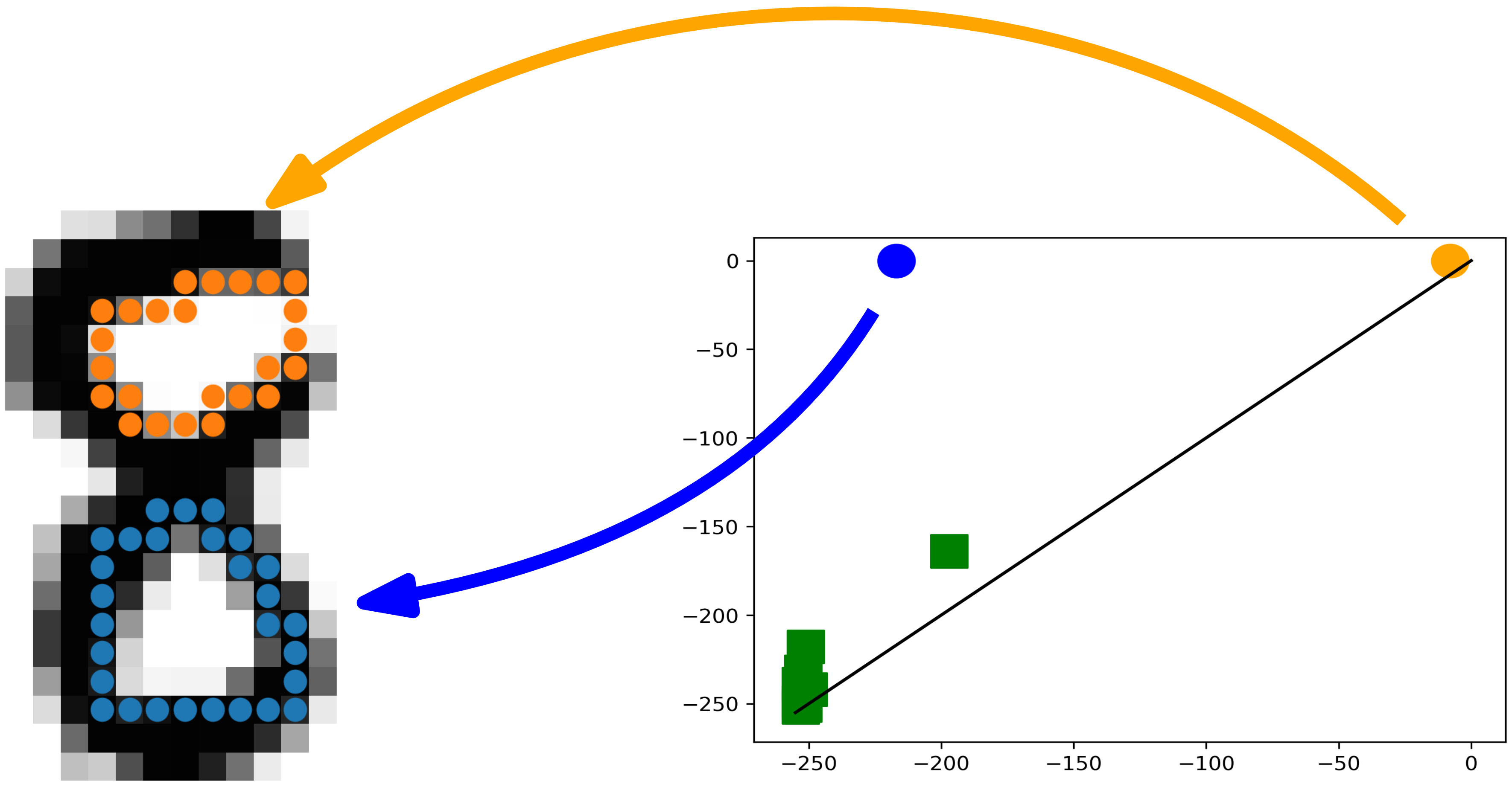


Dataset	Baseline	Before	After	Difference	Dataset	Baseline	Before	After	Difference
vs01	100.0	61.3	99.0	**+37.6**	vs26	99.7	98.8	98.2	-0.6
vs02	99.4	98.8	97.2	-1.6	vs28	99.1	96.8	96.8	0.0
vs06	99.4	87.3	98.2	**+10.9**	vs29	99.1	91.6	98.6	**+7.0**
vs09	99.4	86.8	98.3	**+11.5**	vs34	99.8	99.4	99.1	-0.3
vs16	99.7	89.0	97.3	**+8.3**	vs36	99.7	99.3	99.3	-0.1
vs19	99.6	84.8	98.0	**+13.2**	vs37	98.9	94.9	97.5	**+2.6**
vs24	99.4	98.7	98.7	0.0	vs57	99.7	90.5	97.2	**+6.7**
vs25	99.4	80.6	97.2	**+16.6**	vs79	99.1	85.3	96.9	**+11.5**

What's next?

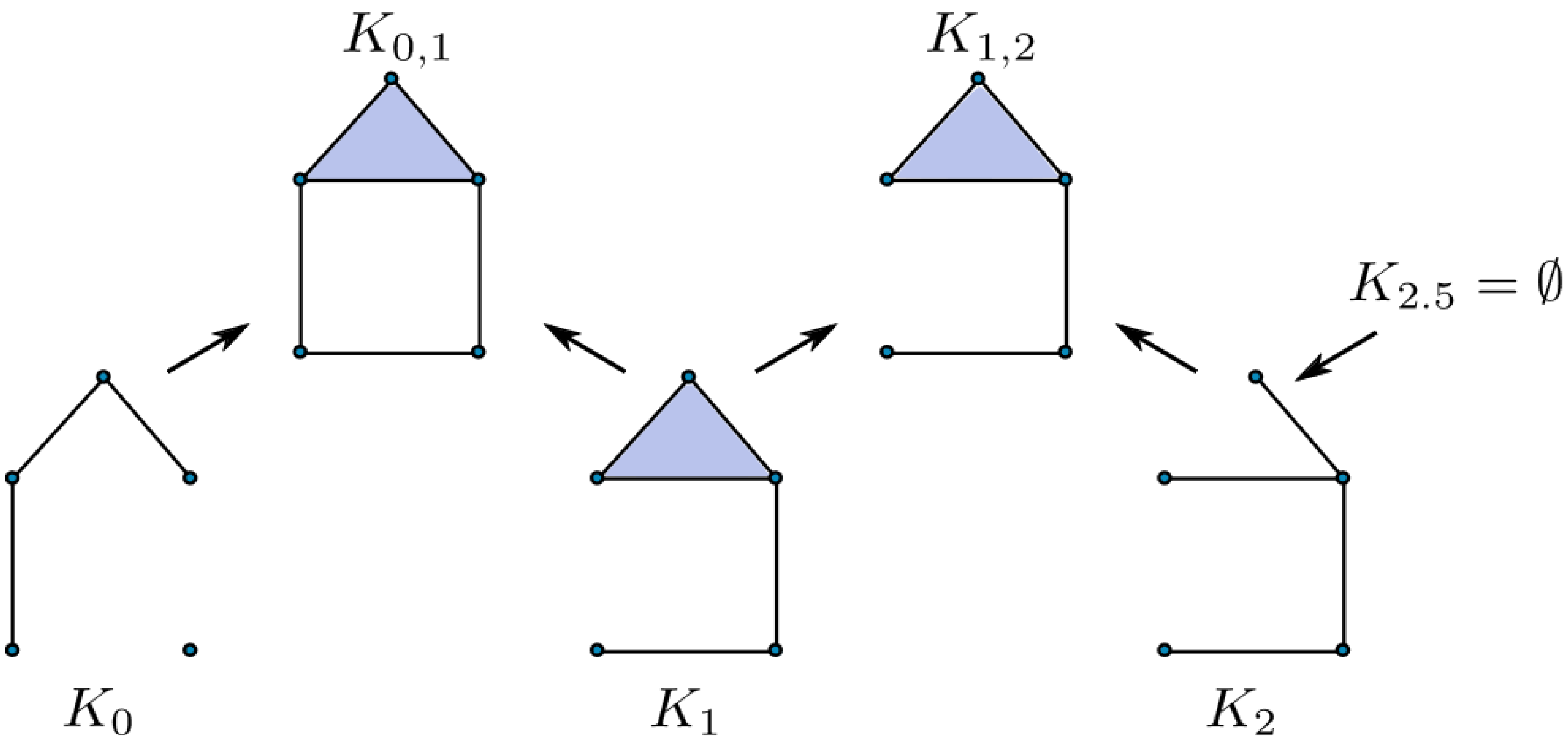
What's next?

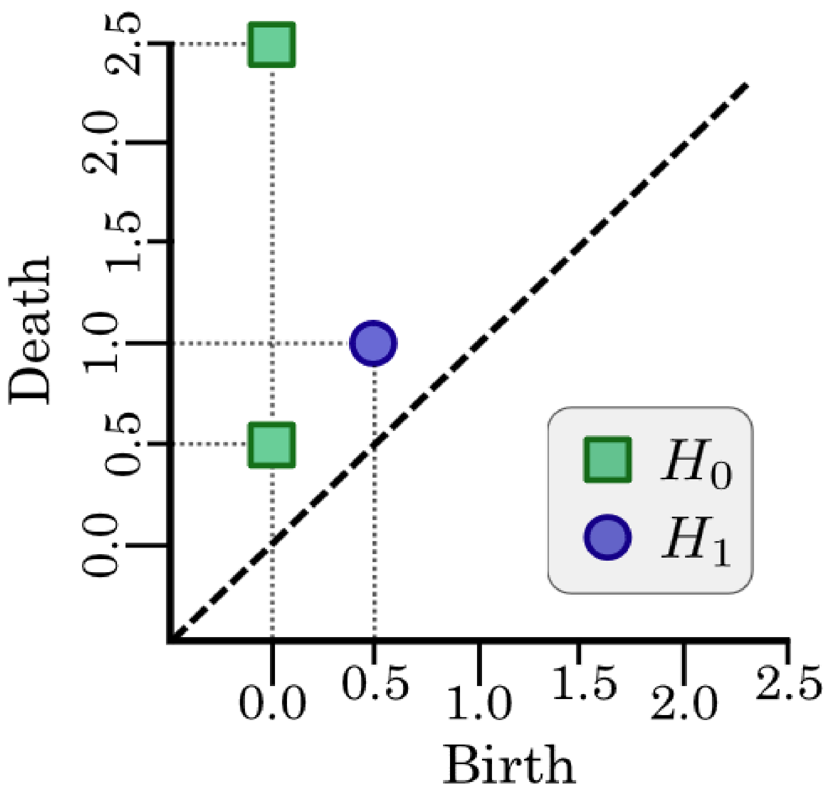
- Representative cycles



What's next?

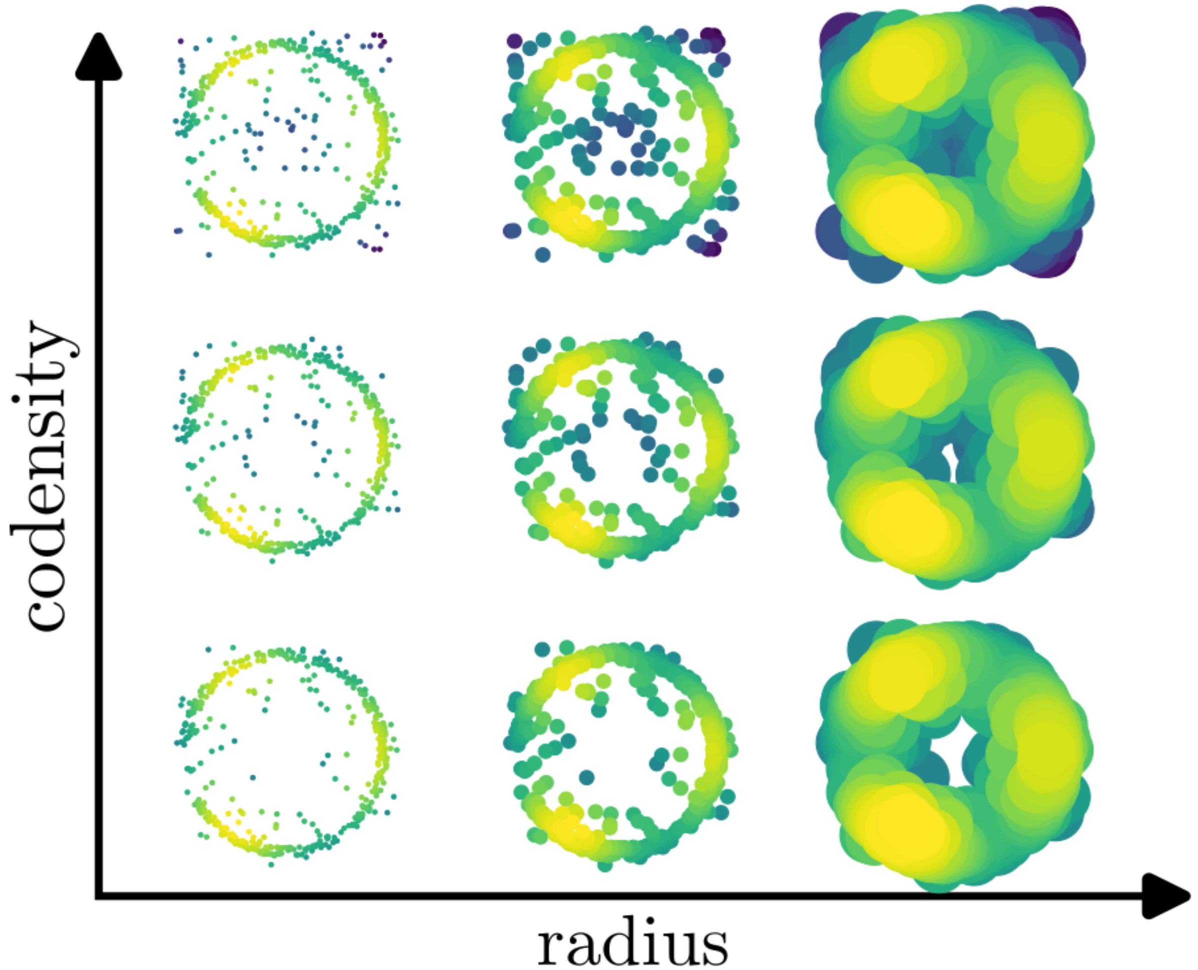
- Representative cycles
- Zigzag persistence





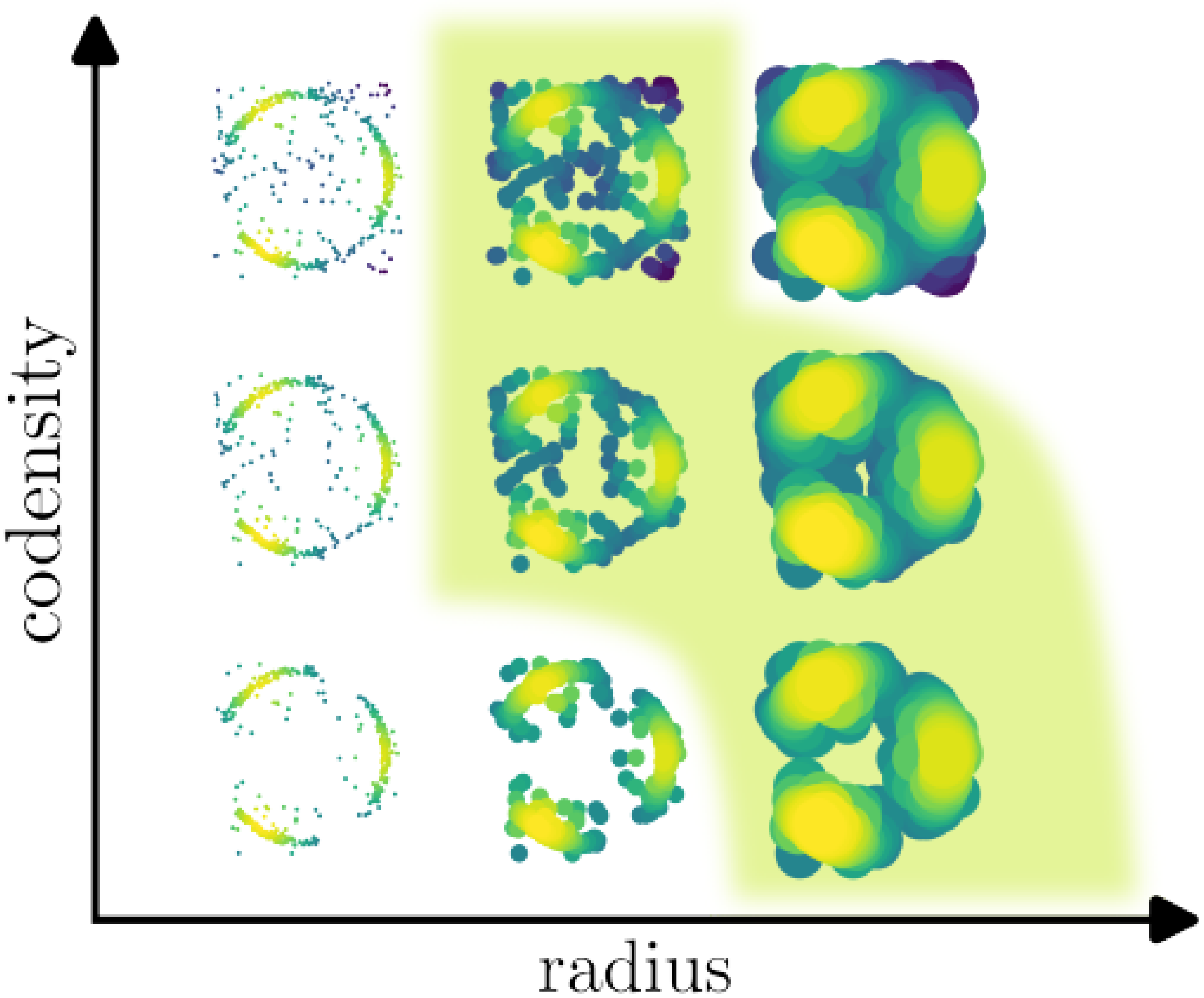
What's next?

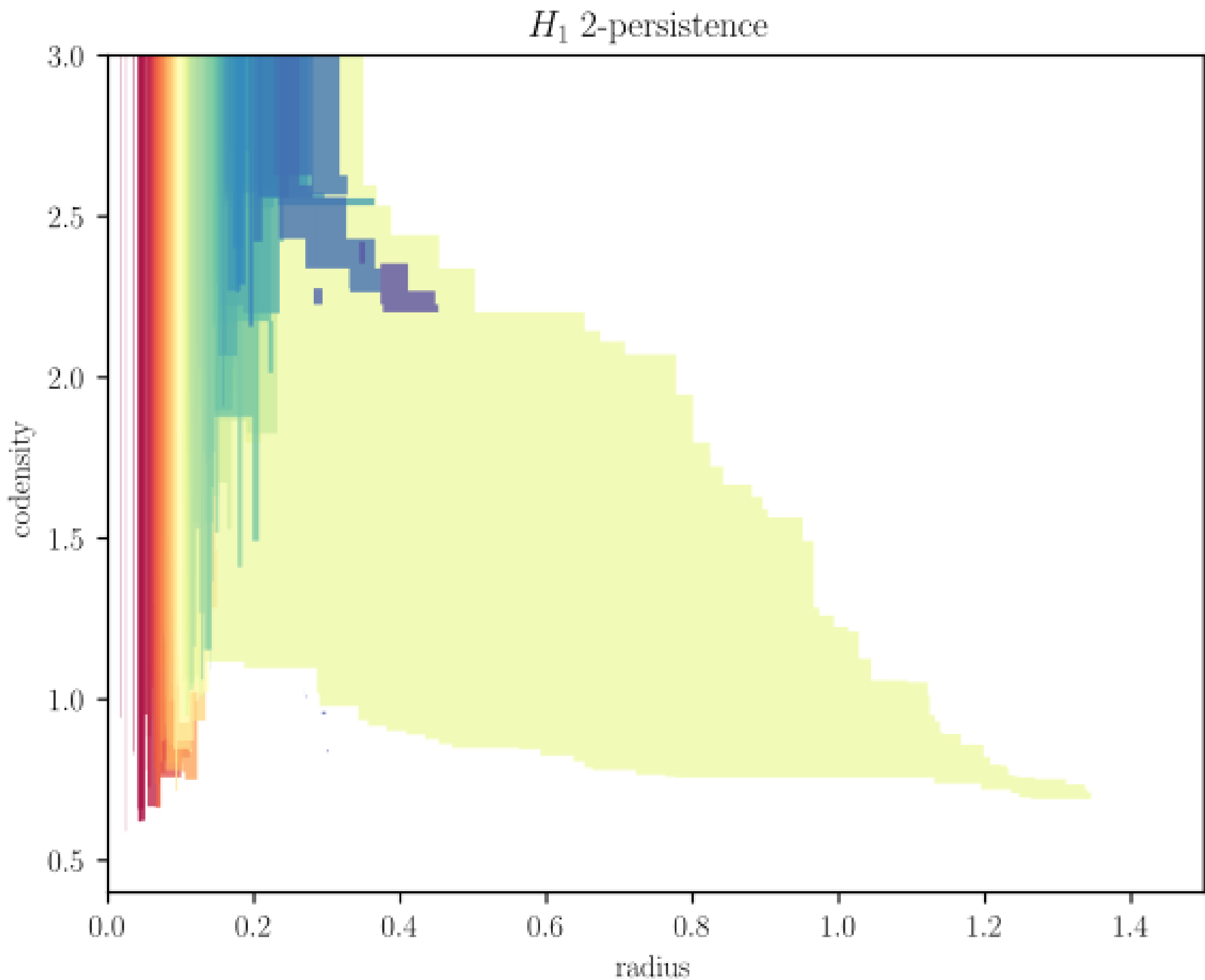
- Representative cycles
- Zigzag persistence
- Multi-parameter persistence



What's next?

- Representative cycles
- Zigzag persistence
- Multi-parameter persistence





Thanks!!