
Basic Applied Topology Subprograms

Brad Nelson

Jun 01, 2022

CONTENTS:

1	Get Started	3
1.1	Installation	3
1.2	Quickstart Guide	5
1.3	Tutorials	9
1.4	Examples	69
1.5	API Reference	84
1.6	About BATS	136
2	Indices and tables	137
	Python Module Index	139
	Index	141

Python bindings for the BATS library. This includes:

- Simplicial, Cubical, and Cell Complexes
- Simplicial, Cubical, and Cellular Maps
- Homology and induced maps
- Persistent homology
- Zigzag homology
- A variety of topological constructions

Note that the C++ repository is the main library, and contains more features. This repository provides bindings for a subset of the functionality of BATS, and is under active development.

For background on what this repository does, refer to the paper [Persistent and Zigzag Homology: A Matrix Factorization Viewpoint](#) by Gunnar Carlsson, Anjan Dwaraknath, and Bradley J. Nelson.

GET STARTED

First, install BATS.py.

Now, you can load BATS in python via

```
import bats
```

Next, check out the [quickstart guide](#).

1.1 Installation

The easiest way to install bats is using pip

```
pip install bats-tda # gcc
```

To use clang (e.g. on a Mac) try

```
CC=clang pip install bats-tda # clang
```

Because bats uses OpenMP, it has to be compiled from source with a C++17 compliant compiler. This means installation can take a few minutes. You can pass --verbose to pip to see what is going on with installation.

If you don't have OpenMP, you can install with a package manager.

GCC (e.g. on Linux)

```
dnf install libgomp-devel # Fedora
```

```
apt-get install libgomp1-dev # Ubuntu
```

Clang (e.g. on Mac)

```
brew install libomp
```

1.1.1 Compiling from Source

You can also compile bats from the source code. This can be useful for debugging installation or contributing to bats.

```
conda create -n bats python=3  
conda install numpy matplotlib
```

if you want to use ipython notebooks, you may want to install

```
conda install ipython notebook
```

clone repository *use recursive option for submodules*

```
git clone --recurse-submodules git@github.com:CompTop/BATS.py.git
```

or if you want to use https protocol:

```
git clone --recurse-submodules https://github.com/CompTop/BATS.py.git
```

Assuming you cloned the repository successfully, just move to the root directory of the repository and install

```
cd BATS.py  
python setup.py install
```

If you want to use clang (for example, on a mac), try

```
CC=clang python setup.py install
```

1.1.2 Development

Some useful commands for development:

build in directory

```
python setup.py build_ext --inplace
```

update submodules

```
git submodule update --remote
```

pull and update submodules

```
git pull --recurse-submodules
```

force a new build (-f), use parallelism (-j)

```
python setup.py build_ext -f -j4  
python setup.py install
```

1.1.3 Testing

BATS.py uses the `unittest` framework. See documentation [here](#).

Running all tests

First, you need to have the `bats` module built in-place.

```
python setup.py build_ext --inplace
```

From the root of the repository, run

```
python -m unittest discover
```

Running individual test files

From `BATS.py/test`, you can run individual test files

```
python -m unittest simplicial.py
```

1.1.4 Upgrading

If you want to update BATS.py to the latest development version, you need to pull and re-build. From the repository root directory:

```
git pull --rebase-submodules
python setup.py build --force # rebuilds all pybind executables
python setup.py install
```

1.2 Quickstart Guide

Once you have [successfully installed bats](#), you can follow this guide to help you get started. You can download this guide as a Jupyter notebook [here](#).

You can find additional information and examples in the [tutorials](#) and [examples](#), and ultimately the [API reference](#).

First, import `bats`:

```
[1]: import bats
```

1.2.1 Simplicial Complexes and Homology

BATS offers two implementations of simplicial complexes: `SimplicialComplex` and `LightSimplicialComplex`. While the internal representations differ, they both have the same interface which can be used.

Simplices in `bats` should generally be assumed to be *ordered*, meaning that `[0, 1, 2]` is not the same as `[1, 2, 0]`. If you want to use *unordered* simplices, you can either add vertices in sorted order, or use a sorting algorithm before adding simplices to complexes.

Basic Applied Topology Subprograms

The add method will add simplices, assuming that all faces have previously been added. The add_recursive method will recursively add faces as needed.

```
[7]: X = bats.SimplicialComplex()
X.add_recursive([0,1,2])
X.add_recursive([2,3])
X.add([1,3])

X.get_simplices()

[7]: [[0], [1], [2], [3], [0, 1], [0, 2], [1, 2], [2, 3], [1, 3], [0, 1, 2]]
```

Now let's compute homology

```
[15]: R = bats.reduce(X, bats.F2()) # F2 is coefficient field

for k in range(R.maxdim()):
    print("dim H_{0}: {1}".format(k, R.hdim(k)))

dim H_0: 1
dim H_1: 1
```

The output of bats.reduce is a ReducedChainComplex which holds information used to compute homology.

For LightSimplicialComplex, you need to provide an upper bound on the number of vertices and maximum simplex dimension.

```
[13]: n = 4 # number of vertices
d = 2 # max simplex dimension
X = bats.LightSimplicialComplex(n, d)
X.add_recursive([0,1,2])
X.add_recursive([2,3])
X.add([1,3])

X.get_simplices()

[13]: [[0], [1], [2], [3], [0, 1], [0, 2], [1, 2], [2, 3], [1, 3], [0, 1, 2]]
```

```
[14]: R = bats.reduce(X, bats.F2())

for k in range(R.maxdim()):
    print("dim H_{0}: {1}".format(k, R.hdim(k)))

dim H_0: 1
dim H_1: 1
```

1.2.2 Persistent Homology

You can add simplices to a filtration by providing a parameter at which they first appear.

```
[17]: F = bats.FilteredSimplicialComplex()
F.add_recursive(0.0, [0,1,2])
F.add_recursive(1.0, [2, 3])
F.add(2.0, [1,3])
```

(continues on next page)

(continued from previous page)

```
F.complex().get_simplices()
```

```
[17]: [[0], [1], [2], [3], [0, 1], [0, 2], [1, 2], [2, 3], [1, 3], [0, 1, 2]]
```

again, we can use the `reduce` function, but now we get a `ReducedFilteredChainComplex`

```
[20]: R = bats.reduce(F, bats.F2())
```

```
for k in range(R.maxdim()):
    for p in R.persistence_pairs(k):
        print(p)

0 : (0,inf) <0,-1>
0 : (0,0) <1,0>
0 : (0,0) <2,1>
0 : (1,1) <3,3>
1 : (0,0) <2,0>
1 : (2,inf) <4,-1>
```

The output of `R.persistence_pairs(k)` is a vector of `PersistencePairs` for k-dimensional persistent homology.

A `PersistencePair` includes 5 pieces of information:

- * The dimension of the homology class.
- * The birth and death parameters of the homology class.
- * The simplex indices responsible for birth and death.

```
[28]: p = R.persistence_pairs(1)[-1]
print(p)
print(p.dim(), p.birth(), p.death(), p.birth_ind(), p.death_ind(), sep=' ', ')
1 : (2,inf) <4,-1>
1, 2.0, inf, 4, 18446744073709551615
```

infinite bars have a death index set to $2^{**64} - 1$

1.2.3 Maps

BATS makes dealing with maps between topological spaces and associated chain maps and induced maps on homology easy. The relevant class is a `CellularMap` which keeps track of what cells in one complex map to what cells in another.

We'll just look at a wrapper for `CellularMap`, called `SimplicialMap` which can be used to extend a map on the vertex set of a `SimplicialComplex` to a map of simplices.

First, we'll build identical simplicial complexes X and Y which are both cycle graphs on four vertices.

```
[30]: X = bats.SimplicialComplex()
X.add_recursive([0,1])
X.add_recursive([1,2])
X.add_recursive([2,3])
X.add_recursive([0,3])

Y = X
```

We then build a simplicial map from X to Y which is extended from a reflection of the vertices.

Basic Applied Topology Subprograms

```
[31]: f0 = [2, 1, 0, 3]
F = bats.SimplicialMap(X, Y, f0)
```

The map is extended by sending vertex i in X to vertex $f0[i]$ in Y . Next, we can apply the chain functor. We'll use F_3 coefficients.

```
[32]: CX = bats.Chain(X, bats.F3())
CY = bats.Chain(Y, bats.F3())
CF = bats.Chain(F, bats.F3())
```

Finally, we can compute homology of the chain complexes and the induced maps.

```
[41]: RX = bats.reduce(CX)
RY = bats.reduce(CY)

for k in range(RX.maxdim()+1):
    HFk = bats.InducedMap(CF, RX, RY, k)
    print("induced map in dimension {}:".format(k))
    print(HFk.tolist())

induced map in dimension 0:
[[1]]
induced map in dimension 1:
[[2]]
```

The induced map in dimension 0 is the identity. The induced map in dimension 1 is multiplication by $2 = -1 \bmod 3$

1.2.4 Zigzag Homology

We'll now compute a simple zigzag barcode, using the above example. We'll consider a diagram with two (identical) spaces, connected by a single edge which applies the reflection map in the above example.

```
[43]: D = bats.SimplicialComplexDiagram(2,1) # diagram with 2 nodes, 1 edge
D.set_node(0, X)
D.set_node(1, Y)
D.set_edge(0, 0, 1, F) # edge 0: F maps from node 0 to node 1
```

We can now apply the Chain and Hom functors to obtain a diagram of homology vector spaces and maps between them

```
[47]: CD = bats.ChainFunctor(D, bats.F3())
```

```
[51]: HD = bats.Hom(CD, 1) # computes homology in dimension 1
ps = bats.barcode(HD, 1) # extracts barcode
for p in ps:
    print(p)

1 : (0,1) <0,0>
```

This indicates there is a 1-dimensional homology bar, which is born in the space with index 0 and survives until the space with index 1. The $\langle 0, 0 \rangle$ indicates which generators are associated with the homology class in the diagram.

1.3 Tutorials

1.3.1 Complexes and Filtrations

SimplicialComplex

Documentation

```
from bats import SimplicialComplex
C = SimplicialComplex()
```

```
X = bats.SimplicialComplex()
X.add([0])
X.add([1])
X.add([0,1])
X.print_summary()
X.boundary(1).print()
```

Add Simplices

Simplices are added as python lists

```
C.add([0])
C.add([1])
C.add([0,1])
```

Get Simplices

You can get a list of all simplices in a given dimension

```
C.get_simplices(0) # [[0], [1]]
G.get_simplices(1) # [[0, 1]]
```

Cell Complex

Documentation

Filtrations

BATS also exposes functionality for filtered verisons of SimplicialComplex, ChainComplex, and ReducedChainComplex

```
import bats

# create FilteredSimplicialComplex
F = bats.FilteredSimplicialComplex()
F.add(0.0, [0])
F.add(0.0, [1])
```

(continues on next page)

(continued from previous page)

```
F.add(0.0, [2])
F.add(1.0, [0,1])
F.add(1.0, [0,2])
F.add(1.0, [1,2])

FC2 = bats.FilteredF2ChainComplex(F)

RFC2 = bats.ReducedFilteredF2ChainComplex(FC2)

# H1, first generator
p = RFC2.persistence_pairs(1)[0]

# extract a homology representative for the generator
v = RFC2.representative(p)
```

1.3.2 Data Sets

1.3.3 Diagrams

Diagrams

Several Types of diagrams are available for use

```
import bats

c1 = [{0,1}, {1,2}]
c2 = [{0,2}, {0,1}]
c3, f1, f2 = bats.bivariate_cover(c1, c2)

D = bats.CoverDiagram(3,2)
D.set_node(0, c1)
D.set_node(1, c3)
D.set_node(2, c2)
D.set_edge(0, 1, 0, f1)
D.set_edge(1, 1, 2, f2)

# Nerve Functor applied to cover diagram
ND = bats.NerveDiagram(D, 1)

# F2 Chain functor applied to diagram of spaces
CD = bats.F2Chain(ND)

# Hom functor applied to diagram of Chain complexes
HD = bats.Hom(CD, 0)

# extract barcode
PD = bats.barcode(HD, 0)

for p in PD:
    print(p)
```

```
0 : (0,2) <0,0>
0 : (1,1) <1,1>
```

1.3.4 Geometric Constructions

Rips, Dowker, Witness, Nerve, Cover Complexes

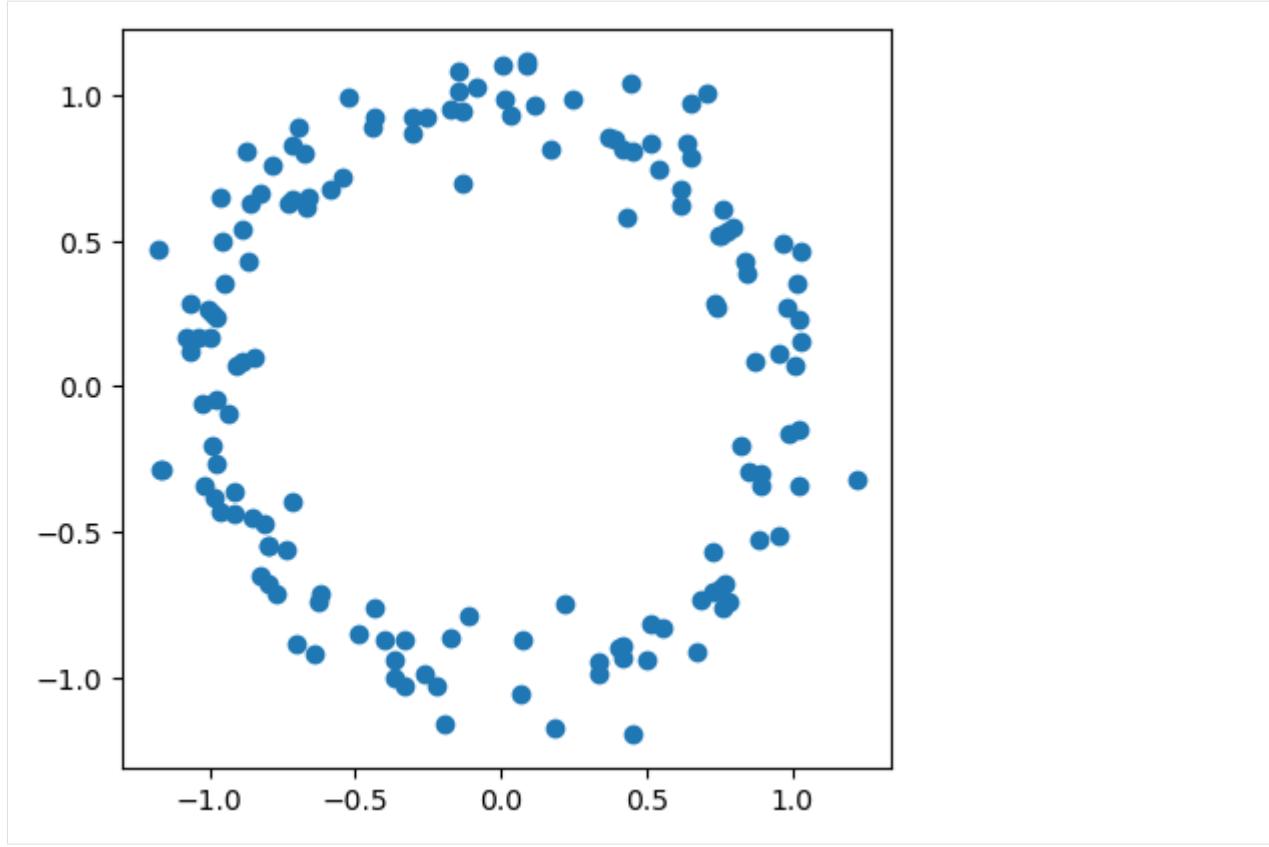
1.3.5 Rips Complex Tutorial

This is a quick Rips Filtration tutorial used to illustrate options provided in BATS.py.

```
[1]: import bats
import numpy as np
import matplotlib.pyplot as plt
import scipy.spatial.distance as distance
import bats
import time
```

```
[2]: # first, generate a circle
np.random.seed(0)

n = 150
X = np.random.normal(size=(n,2))
X = X / np.linalg.norm(X, axis=1).reshape(-1,1)
X = X + np.random.normal(size=(n,2), scale = 0.1 )
fig = plt.scatter(X[:,0], X[:,1])
fig.axes.set_aspect('equal')
plt.savefig('figures/RipsEx_data.png')
plt.show()
```



Rips filtrations are commonly used in conjunction with persistent homology to create features for finite dimensional metric spaces (point clouds). Given a metric space (X, d) , a Rips complex consists of simplices with a maximum pairwise distance between vertices is less than some threshold r :

$$X_r = \{(x_0, \dots, x_k) \mid x_i \in X, d(x_i, x_j) \leq r\}.$$

A Rips filtration is a filtration of Rips complexes $X_r \subseteq X_s$ if $r \leq s$.

```
[3]: # compute pairwise distances
D = distance.squareform(distance.pdist(X))

# Rips complex for full metric space
# ie., threshold r = infinity now
F = bats.LightRipsFiltration(bats.Matrix(D), np.inf, 2)

# compute with F2 coefficents
t0 = time.monotonic()
R = bats.reduce(F, bats.F2())
t1 = time.monotonic()
print("time of compute persistent homology: {} sec.".format(t1-t0))
time of compute persistent homology: 2.9647895510006492 sec.
```

Now you are able to see persistent diagrams.

```
[4]: # find persistence pairs at each dimension
ps = []
```

(continues on next page)

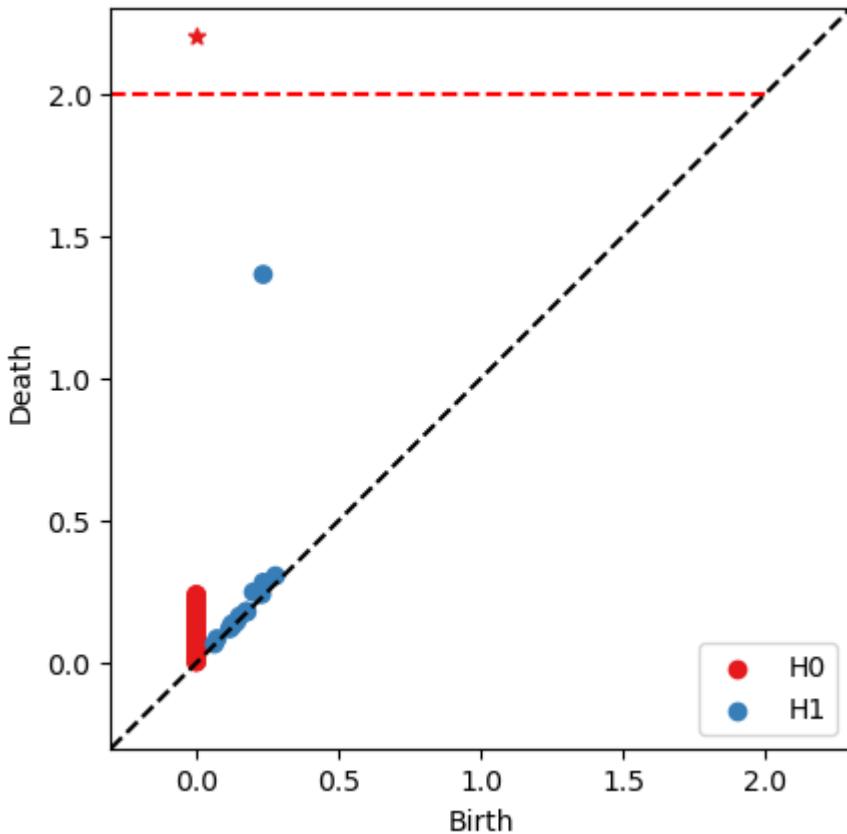
(continued from previous page)

```

for d in range(R.maxdim()):
    ps.extend(R.persistence_pairs(d))

# Draw persistent diagram
# 'tmax' is the axis maximum value
fig, ax = bats.persistence_diagram(ps, tmax = 2.0)
plt.show()

```



Efficient Computation

The number of simplices in Rips filtrations quickly grows with the size of the data set, and much effort has gone into developing efficient algorithms for computing persistent homology of Rips filtrations.

Construction optimization

The first method that has been applied in several high-performance packages for Rips computations is to stop a filtration at the enclosing *radius* of the metric space, at which point the complex becomes contractible, which can reduce the total number of simplices in the filtration considerably without changing persistent homology.

```
[5]: # Two ways to find Enclosing Radius
r_enc = np.min(np.max(D, axis=0))
print("enclosing radius = {}".format(r_enc))
```

(continues on next page)

(continued from previous page)

```
r_enc = bats.enclosing_radius(bats.Matrix(D))
print("enclosing radius = {}".format(r_enc))

enclosing radius = 1.8481549465930773
enclosing radius = 1.8481549465930773
```

```
[6]: # Rips complex up to enclosing radius
t0 = time.monotonic()
F_enc = bats.LightRipsFiltration(bats.Matrix(D), r_enc, 2)
t1 = time.monotonic()
print("construction time: {} sec.".format(t1-t0))

# compute with F2 coefficents
t0 = time.monotonic()
R_enc = bats.reduce(F_enc, bats.F2())
t1 = time.monotonic()
print("reduction time: {} sec.".format(t1-t0))

construction time: 0.07883952199881605 sec.
reduction time: 1.04050104900125 sec.
```

You can see the obvious improvement with about 2x speedup.

Algorithm optimization

There are also many efficient algorithms provided in BATS.py:

- **Clearing/Compression**: two options without basis returned.
- **Cohomology**: faster on some filtrations.
- **Update Persistence**: suitable when there are several similar datasets needed to be computed PH (e.g., optimization on persistence penalty).
- **Extra Reduction**: perform extra reduction to eliminate nonzeros even after pivot has been found. Performs
- Combinations of the above options

```
[7]: def time_BATS_flags(X, flags=(bats.standard_reduction_flag(), bats.compute_basis_
flag())):

    t0 = time.monotonic()
    D = distance.squareform(distance.pdist(X))
    r_enc = bats.enclosing_radius(bats.Matrix(D))
    F_enc = bats.LightRipsFiltration(bats.Matrix(D), r_enc, 2)
    t0a = time.monotonic()
    R = bats.reduce(F_enc, bats.F2(), *flags)
    t1 = time.monotonic()
    print("{} sec.\n{} sec".format(t1 - t0a, t1 - t0))
```

```
[8]: flags = [
    (bats.standard_reduction_flag(), bats.compute_basis_flag()),
    (bats.standard_reduction_flag(),),
```

(continues on next page)

(continued from previous page)

```

(bats.standard_reduction_flag(), bats.clearing_flag()),
(bats.standard_reduction_flag(), bats.compression_flag()),
(bats.extra_reduction_flag(), bats.compute_basis_flag()),
(bats.extra_reduction_flag(),),
(bats.extra_reduction_flag(), bats.clearing_flag()),
(bats.extra_reduction_flag(), bats.compression_flag()),
]
labels = [
    "standard w/ basis\t",
    "standard w/ no basis\t",
    "standard w/ clearing\t",
    "standard w/ compression\t",
    "extra w/ basis\t\t",
    "extra w/ no basis\t",
    "extra w/ clearing\t",
    "extra w/ compression\t"
]
print("flags\t\t\tReduction time\tTotal Time")
for flag, label in zip(flags, labels):
    print("{}{}".format(label), end=' ')
    time_BATS_flags(X, flag)

flags          Reduction time  Total Time
standard w/ basis      1.019 sec.   1.117 sec
standard w/ no basis     0.521 sec.   0.602 sec
standard w/ clearing     0.519 sec.   0.607 sec
standard w/ compression   0.492 sec.   0.577 sec
extra w/ basis           0.905 sec.   0.985 sec
extra w/ no basis         0.301 sec.   0.386 sec
extra w/ clearing          0.304 sec.   0.388 sec
extra w/ compression       0.184 sec.   0.268 sec

```

[9]: # add some noise to the original datasets
to create a similar datasets to show the performance of update persistence
X2 = X + np.random.normal(size=(n, 2), scale = 0.001)

[10]: # PH computation on X1
D = distance.squareform(distance.pdist(X))
r_enc = bats.enclosing_radius(bats.Matrix(D))
F_X = bats.LightRipsFiltration(bats.Matrix(D), r_enc, 2)
R = bats.reduce(F_X, bats.F2())

PH computation on X2 by update persistence on X1
t0 = time.monotonic()
D2 = distance.squareform(distance.pdist(X2))
r_enc2 = bats.enclosing_radius(bats.Matrix(D2))
F_Y = bats.LightRipsFiltration(bats.Matrix(D2), r_enc2, 2) # generate a RipsFiltration
UI = bats.UpdateInfoLightFiltration(F_X, F_Y) # find updating information
R.update_filtration_general(UI)
t1 = time.monotonic()
print("compute PH by updating persistence needs: {:.3f} sec.".format(t1 - t0))

```
compute PH by updating persistence needs: 0.492 sec.
```

Advantages of updating persistence over the other options:

1. still keep the basis information;
2. with a comparable speedup with other optimization algorithms.

1.3.6 Lower-star Filtrations

A lower-star filtration on a simplicial complex X is a filtration extended from a function on the vertex set $f : X_0 \rightarrow \mathbb{R}$. The lower-star filtration of X at parameter a is the maximal sub-complex of X on the inverse image

$$f^{-1}((-\infty, a]) \subseteq X_0 \quad (1.1)$$

The parameter at which a simplex $(x_0, \dots, x_k) \in X$ appears at is

$$f : (x_0, \dots, x_k) \mapsto \max\{f(x_i) \mid i = 0, \dots, k\} \quad (1.2)$$

Application to images

A common use of lower-star filtrations is to building filtrations on images using pixel values.

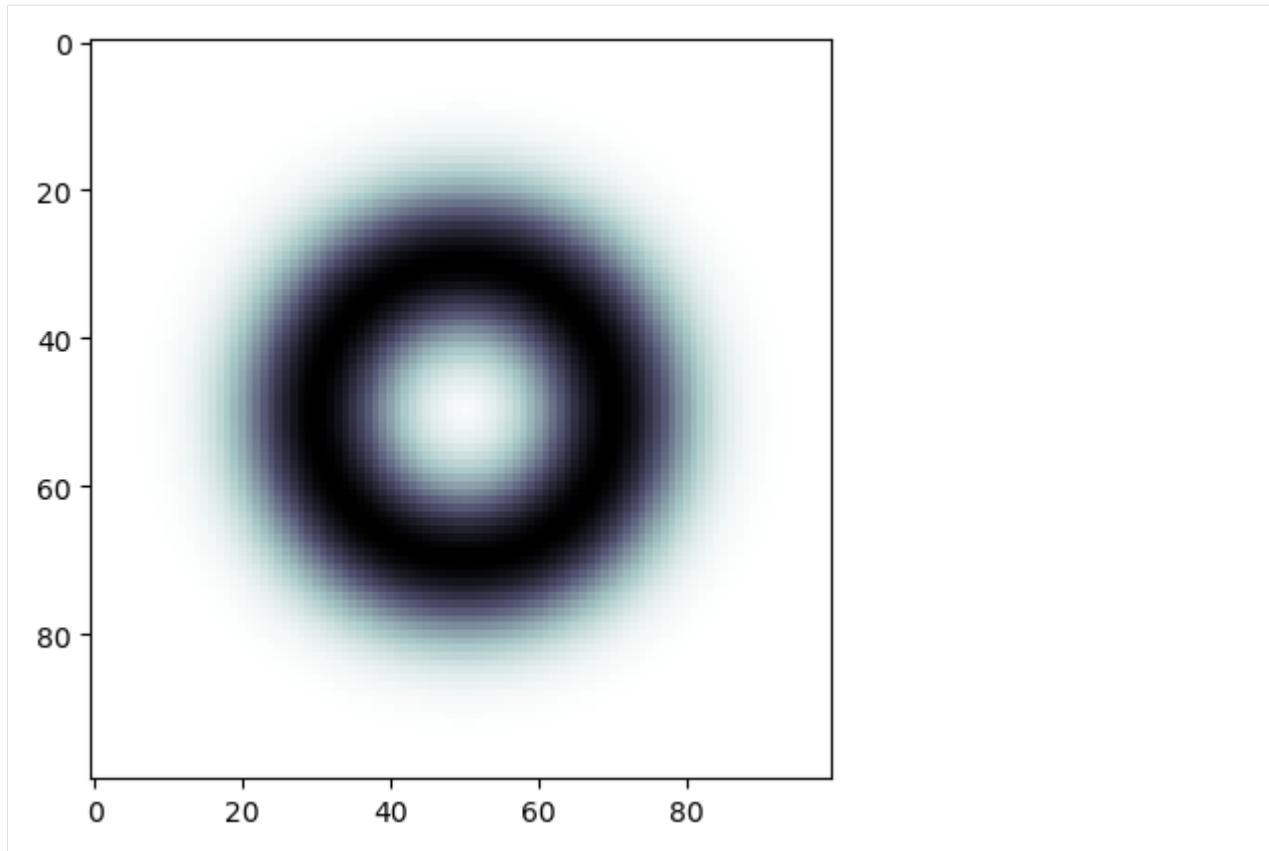
```
[1]: import numpy as np
import matplotlib.pyplot as plt

n = 100
img = np.empty((n,n))

def rad(i,j, n):
    return np.sqrt((i - n/2)**2 + (j - n/2)**2)

for i in range(n):
    for j in range(n):
#        print(i,j, rad(i,j,n))
        img[i,j] = 1-np.exp(-(rad(i,j,n) - 20)**2/100)

plt.imshow(img, cmap='bone')
plt.show()
```



This image is a function on the square discretized into pixels. We can build a simplicial complex which triangulates the square with vertices (0-simplices) identified with the pixels using the Freudenthal triangulation.

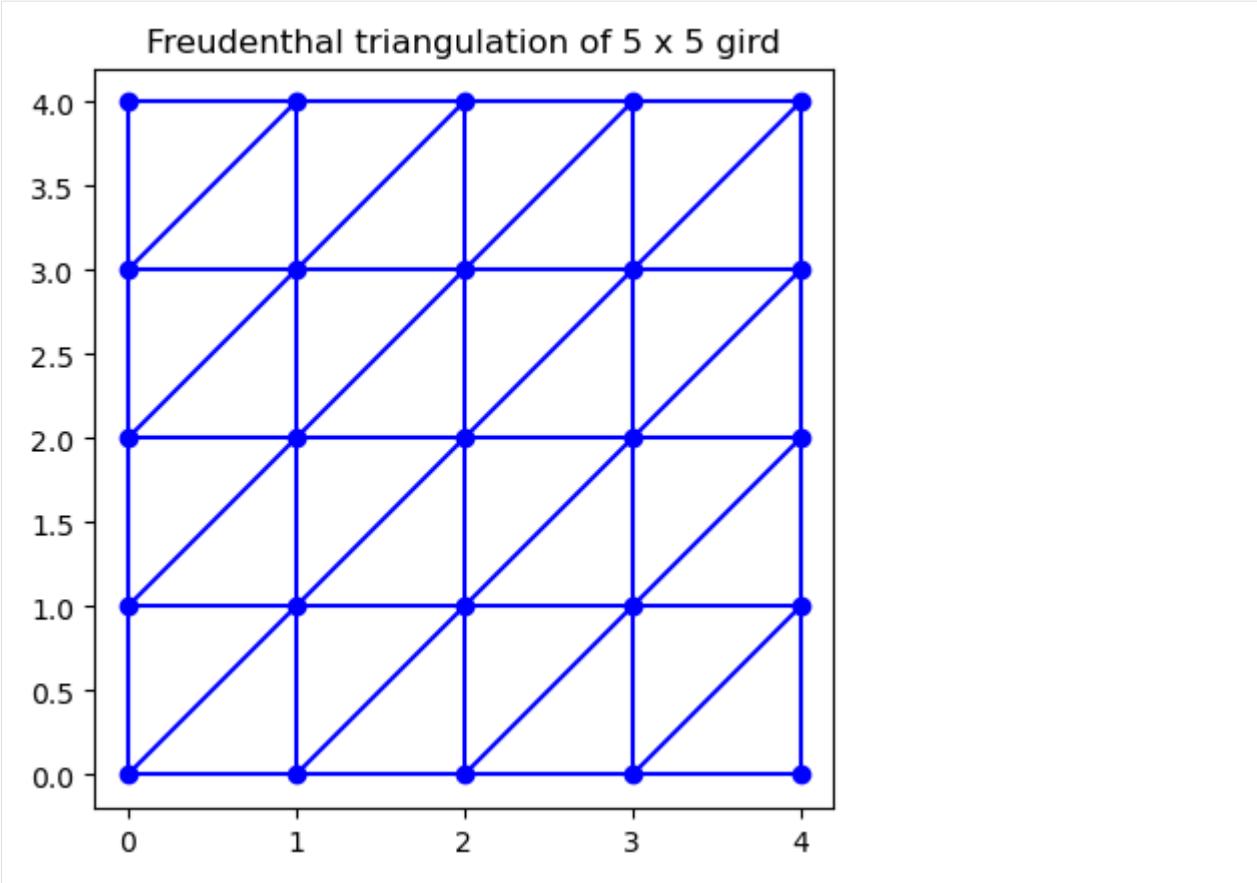
```
[2]: import bats

m = 5
X = bats.Freudenthal(m, m)
xx, yy = np.meshgrid(np.arange(m), np.arange(m))
xx = xx.flatten()
yy = yy.flatten()

fig, ax = plt.subplots()

# scatter vertices
ax.scatter(xx, yy, c='b')
# plot edges
for e in X.get_simplices(1):
    ax.plot(xx[e], yy[e], c='b')

ax.set_title("Freudenthal triangulation of {} x {} grid".format(m))
ax.set_aspect('equal')
plt.show(fig)
```



To compute the lower-star filtration, we can use `bats.lower_star_filtration`, which returns the filtration value for each simplex as well as a map back to the largest value vertex in the simplex.

```
[3]: X = bats.Freudenthal(n, n)
vals, imap = bats.lower_star_filtration(X, img.flatten()) # computes filtration
# parameter to
F = bats.Filtration(X, vals)
```

We can visualize the filtration on vertices:

```
[4]: def show_sublevel(ax, F, alpha, xx, yy):
    """
    visualize the sublevelset (-inf, alpha]
    """
    Xa = F.sublevelset(alpha)
    ax.scatter(xx, yy, color=(0,0,1,0.01))

    # 0-simplices
    X0 = Xa.get_simplices(0)
    ax.scatter(xx[X0], yy[X0], color=(0,0,1,1))

    ax.set_title(r"$f^{-1}((-\infty, \{])".format(alpha))

alphas = [0.2, 0.5, 0.8, 1.0]
fig, axs = plt.subplots(1, len(alphas), figsize = (5*len(alphas), 5))
```

(continues on next page)

(continued from previous page)

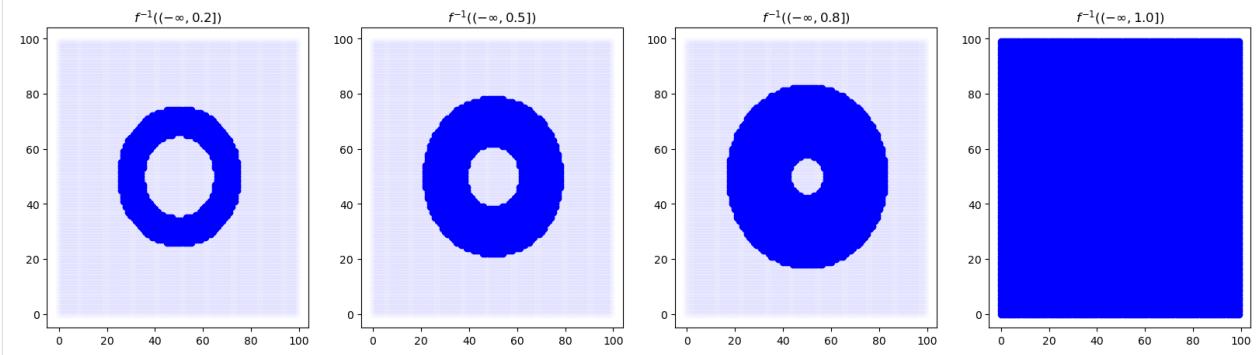
```

xx, yy = np.meshgrid(np.arange(n), np.arange(n))
xx = xx.flatten()
yy = yy.flatten()

for ax, alpha in zip(axes, alphas):
    show_sublevel(ax, F, alpha, xx, yy)

plt.show(fig)

```

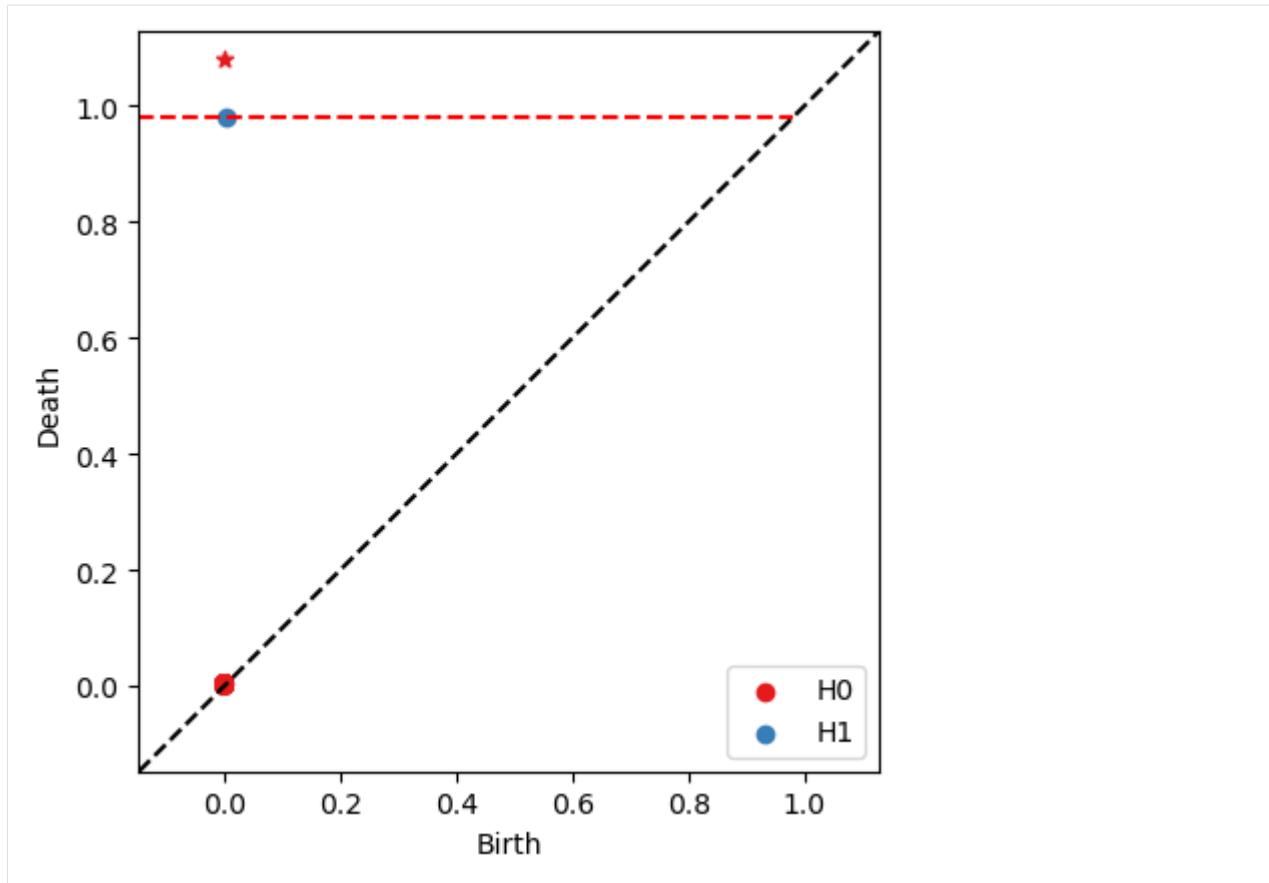


And compute persistent homology

```

[5]: RF = bats.reduce(F, bats.F2())
ps = RF.persistence_pairs(0) + RF.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()

```



We see a prominent H_1 pair corresponding to the annulus

```
[6]: for p in ps:
    if p.length() > 0.5:
        print(p)

0 : (0,inf) <3050,-1>
1 : (0.00378158,0.981684) <20537,10001>
```

Visualization

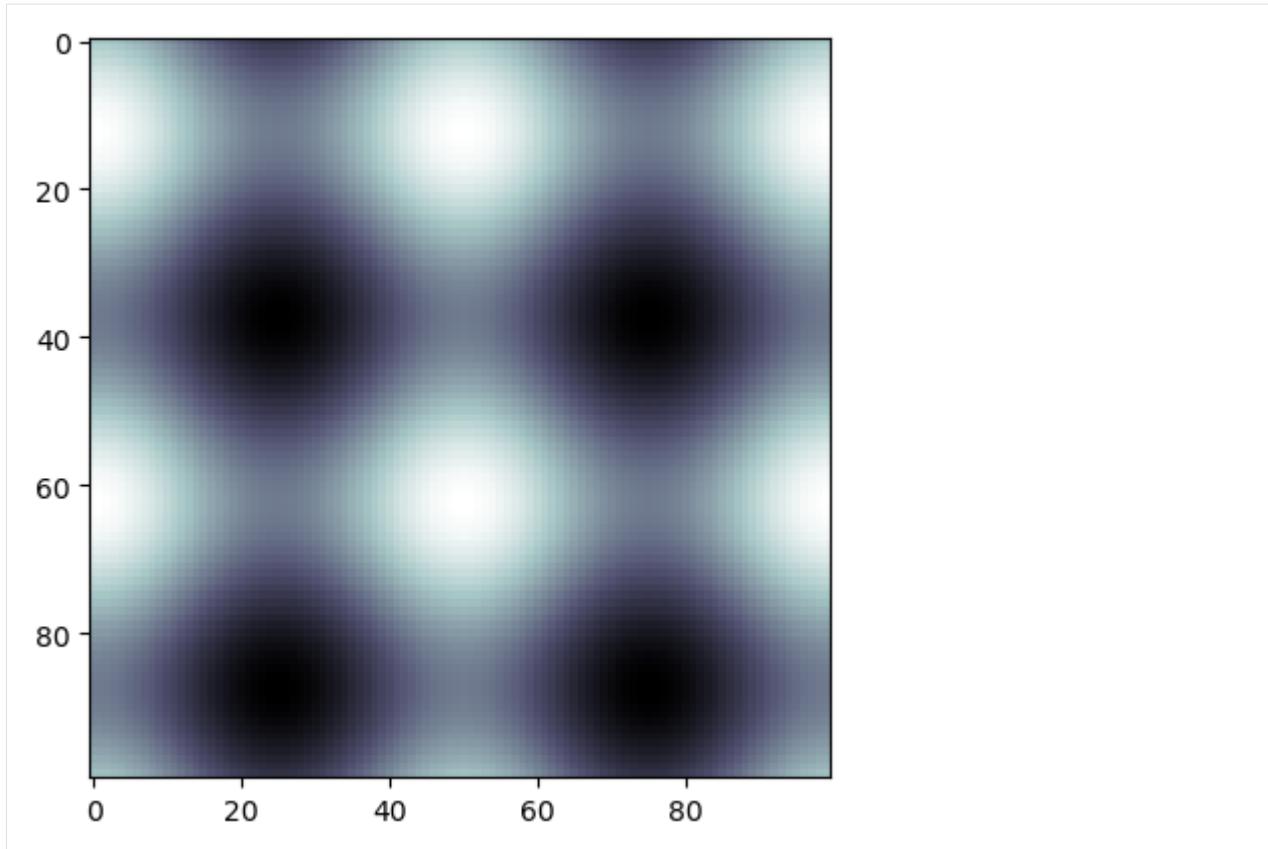
Homology Generators

You can extract homology generators from a `ReducedFilteredChainComplex`, the output of `bats.reduce`.

Let's look at a new image with several long bars in H_0 and H_1

```
[7]: img2 = np.empty((n,n))
for i in range(n):
    for j in range(n):
        img2[i,j] = np.sin(4*np.pi*i/n) + np.cos(4*np.pi*j/n)

plt.imshow(img2, cmap='bone')
plt.show()
```



```
[8]: vals, imap = bats.lower_star_filtration(X, img2.flatten()) # computes filtration
      ↪parameter to
F = bats.Filtration(X, vals)

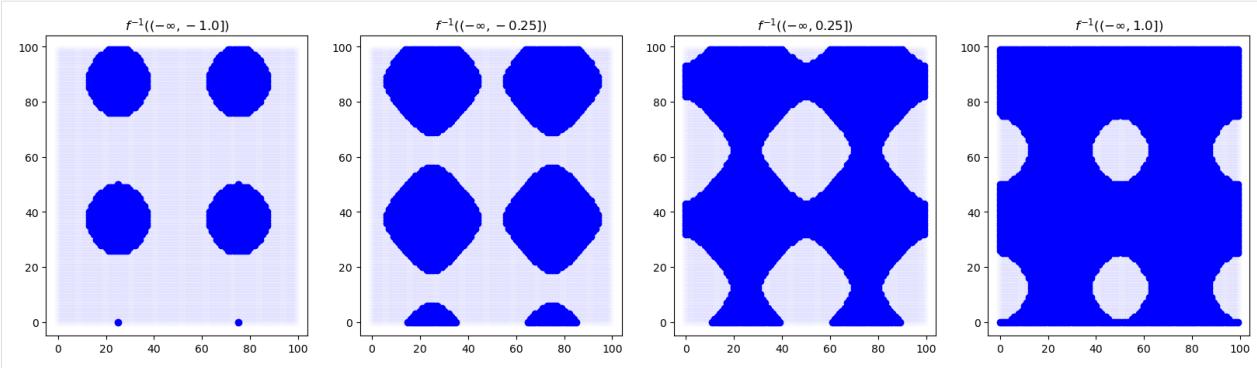
alphas = [-1.0, -0.25, 0.25, 1.0]
fig, axs = plt.subplots(1, len(alphas), figsize = (5*len(alphas), 5))

xx, yy = np.meshgrid(np.arange(n), np.arange(n))
xx = xx.flatten()
yy = yy.flatten()

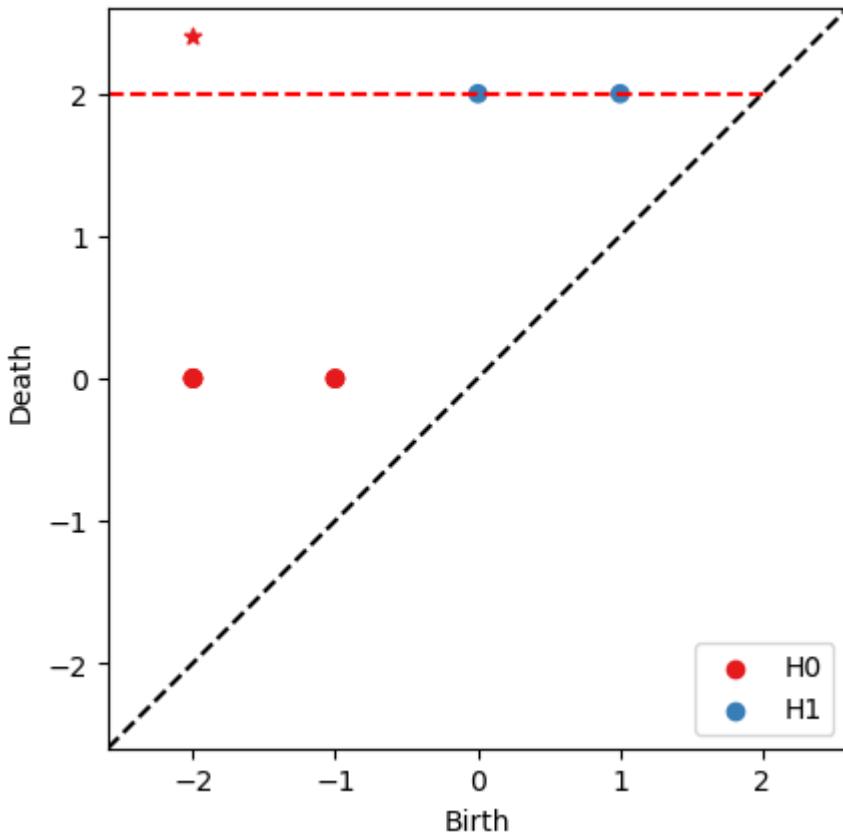
for ax, alpha in zip(axs, alphas):
    show_sublevel(ax, F, alpha, xx, yy)

plt.show(fig)
```

Basic Applied Topology Subprograms



```
[9]: RF = bats.reduce(F, bats.F2())
ps = RF.persistence_pairs(0) + RF.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



```
[10]: nzps = [p for p in ps if p.length() > 0]
for p in nzps:
    print(p)

0 : (-1.99803,inf) <3725,-1>
0 : (-1.99803,0.00197327) <3775,10979>
0 : (-1.99803,-0.00197327) <8825,18948>
0 : (-1.99803,-0.00197327) <8875,19098>
```

(continues on next page)

(continued from previous page)

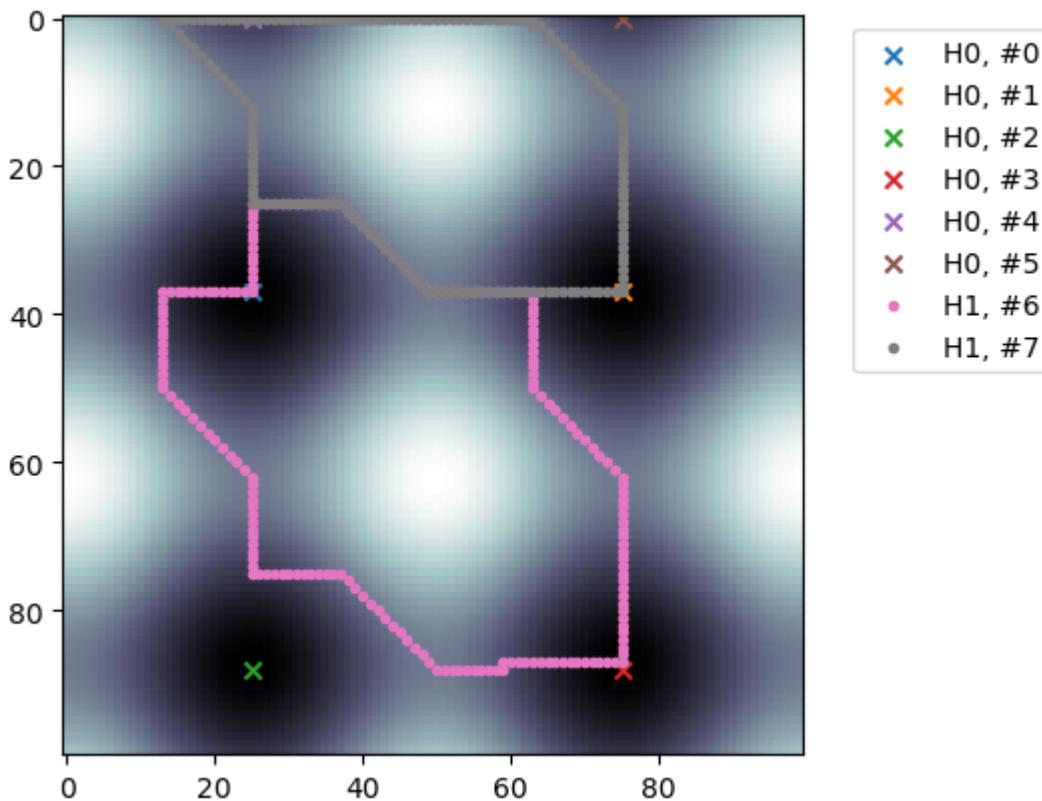
```
0 : (-1,-0.00197327) <51,4048>
0 : (-1,-0.00197327) <151,4198>
1 : (0.00197327,1.99803) <26177,12575>
1 : (1,1.99803) <203,2675>
```

```
[11]: # visualization of representatives
plt.imshow(img2, origin='upper', cmap='bone')

for pi, p in enumerate(nzps):
    c = RF.representative(p, False)
    d = p.dim()
    supp = np.unique([X.get_simplex(d, i) for i in c.nzinds()])

    xx = supp % n
    yy = supp // n
    if p.dim() == 0:
        marker='x'
    else:
        marker='.'
    plt.scatter(xx[:,],yy[:,],marker=marker,label="H{}, #{}".format(d, pi))

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



Critical Simplices

Critical simplices are the simplices whose addition to the filtration causes a homology class to be born or killed.

Lower-star filtrations provide an inverse map back to the pixel which gave the critical simplex its filtration value. This is the `imap` which we computed in `vals, imap = bats.lower_star_filtration(...)`

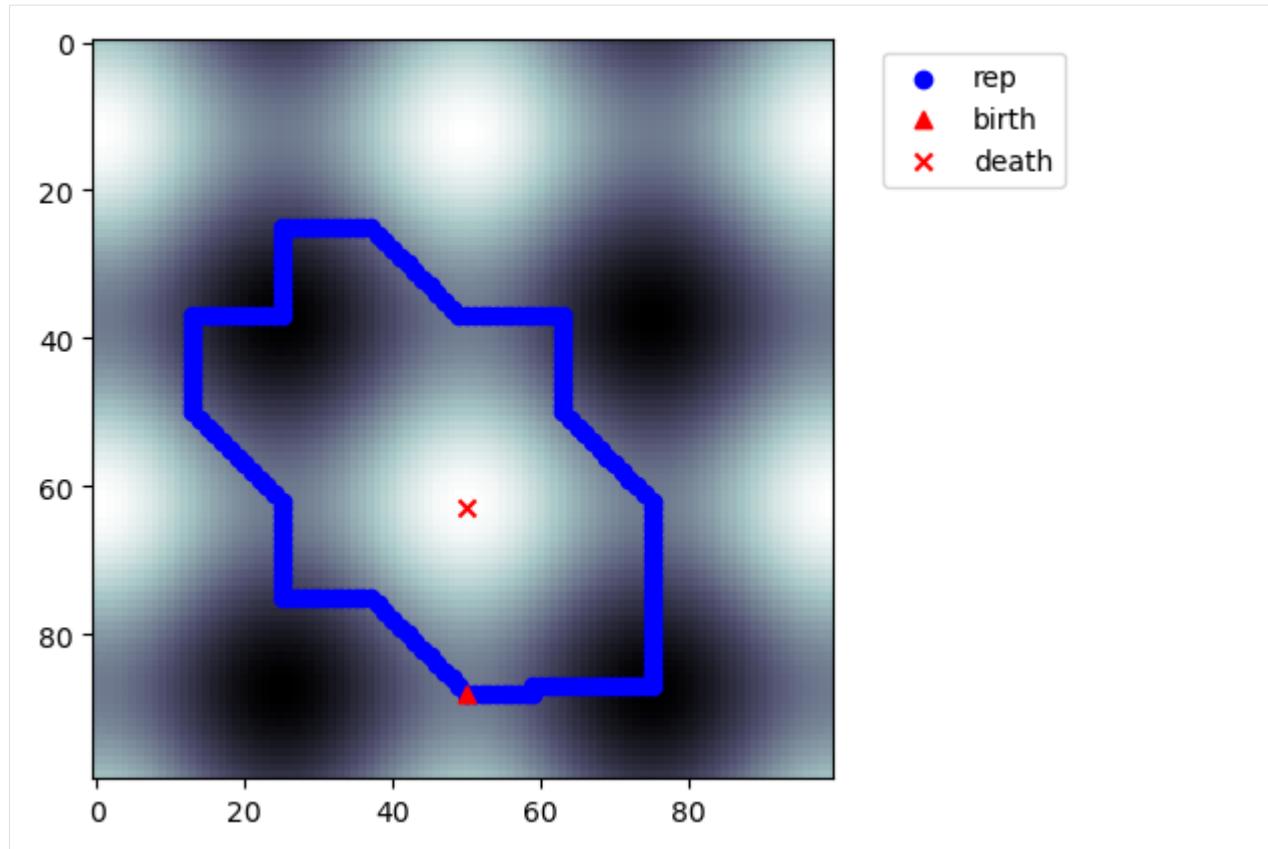
```
[12]: p = nzps[-2]
d = p.dim()
plt.imshow(img2, cmap='bone')

# representative
c = RF.representative(p)
supp = np.unique([X.get_simplex(d, i) for i in c.nzinds()])

xx = supp % n
yy = supp // n
plt.scatter(xx[:,],yy[:,],c='b',label="rep")

bskel = np.array([imap[d][p.birth_ind()]])
xx = bskel % n
yy = bskel // n
plt.scatter(xx[:,],yy[:,], c='r', marker='^', label="birth")
if p.death_ind() != 18446744073709551615:
    dskel = np.array([imap[d+1][p.death_ind()]])
    xx = dskel % n
    yy = dskel // n
    plt.scatter(xx[:,],yy[:,], c='r', marker='x', label="death")

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```

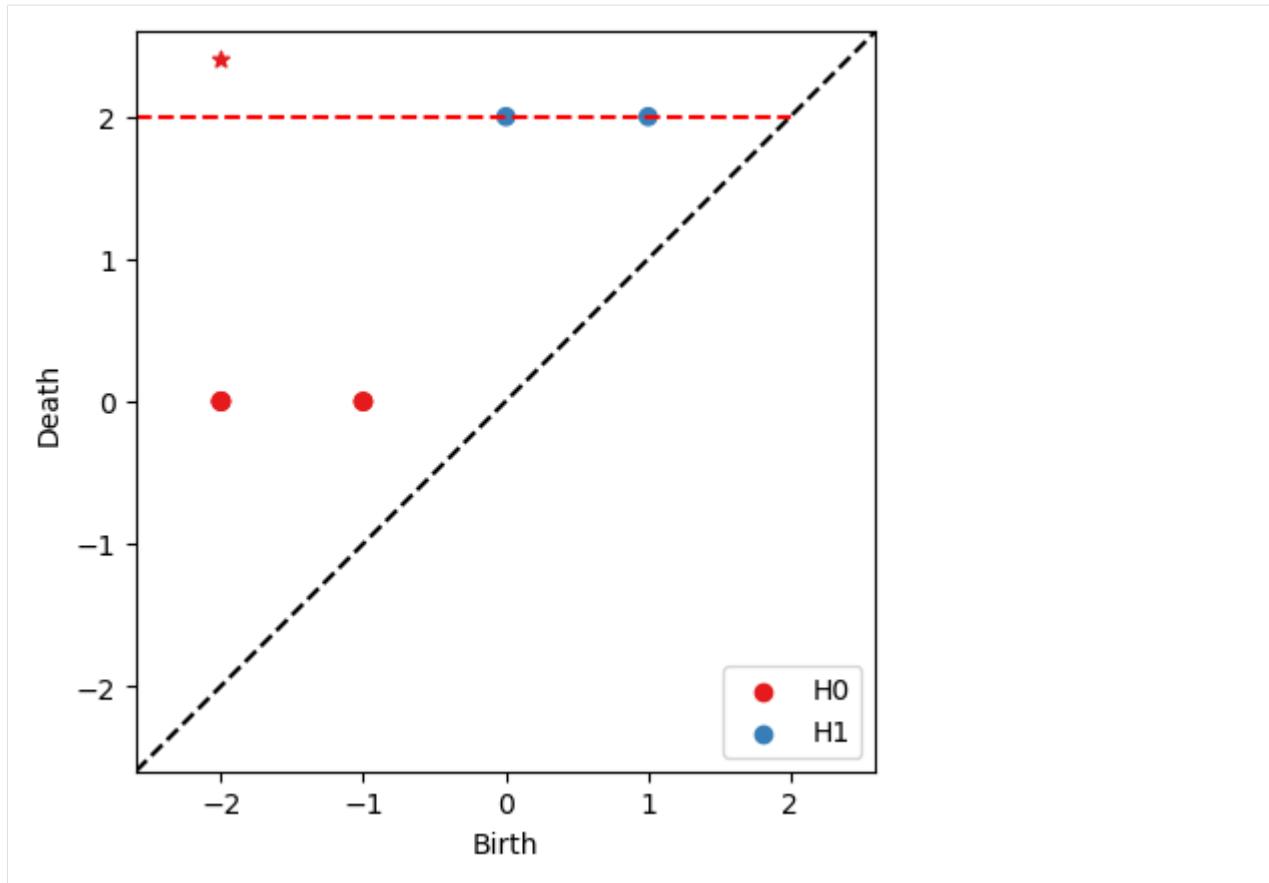


Non-Uniqueness of Generators

Homology generators are not unique, as we can add any element of $\text{img } \partial_{k+1}$ to a representative of H_k and stay in the homology class. However, the critical simplices will be the same.

This behavior can be seen using different reduction options in BATS.

```
[13]: RF = bats.reduce(F, bats.F2(), bats.standard_reduction_flag(), bats.clearing_flag(),
                     bats.compute_basis_flag())
ps = RF.persistence_pairs(0) + RF.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



```
[14]: nzps = [p for p in ps if p.length() > 0]

p = nzps[-2]
d = p.dim()
plt.imshow(img2, cmap='bone')

# representative
c = RF.representative(p)
supp = np.unique([X.get_simplex(d, i) for i in c.nzinds()])

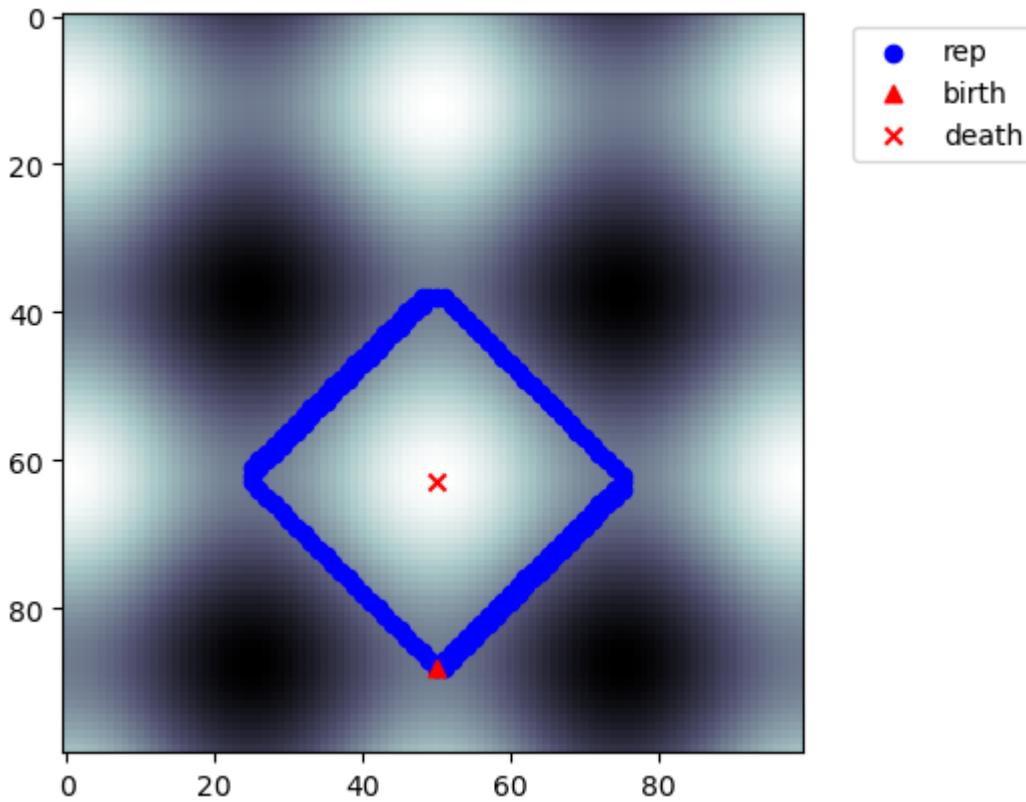
xx = supp % n
yy = supp // n
plt.scatter(xx[:,], yy[:,], c='b', label="rep")

bskel = np.array([imap[d][p.birth_ind()]])
xx = bskel % n
yy = bskel // n
plt.scatter(xx[:,], yy[:,], c='r', marker='^', label="birth")
if p.death_ind() != 18446744073709551615:
    dskel = np.array([imap[d+1][p.death_ind()]])
    xx = dskel % n
    yy = dskel // n
    plt.scatter(xx[:,], yy[:,], c='r', marker='x', label="death")
```

(continues on next page)

(continued from previous page)

```
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```

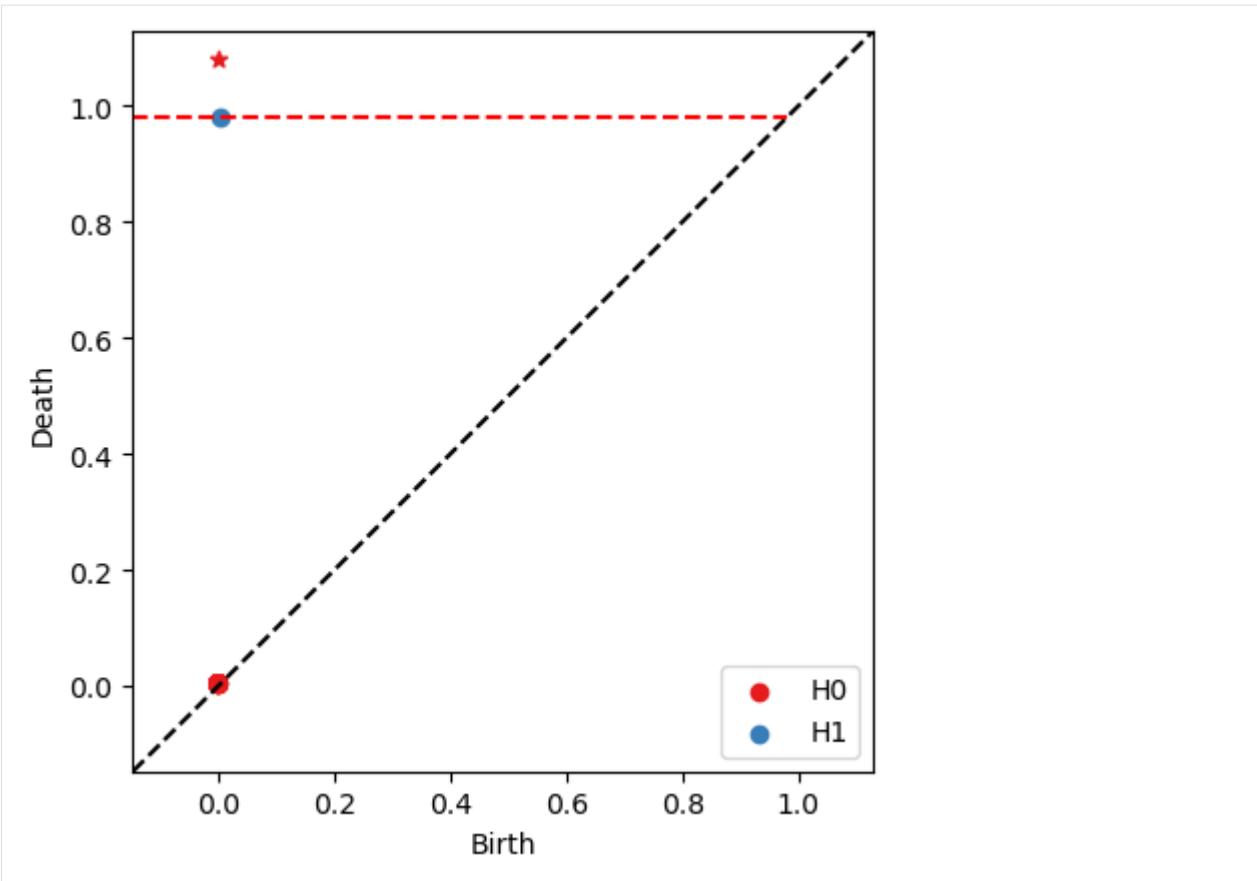


Cubical Complexes

Lower-star filtrations can be applied to cubical complexes as well (the inverse map is not computed)

```
[15]: X = bats.Cube(n,n)
       vals = bats.lower_star_filtration(X, img)
       F = bats.Filtration(X, vals)
```

```
[16]: RF = bats.reduce(F, bats.F2())
       ps = RF.persistence_pairs(0) + RF.persistence_pairs(1)
       bats.persistence_diagram(ps)
       plt.show()
```



```
[17]: for p in ps:
    if p.length() > 0.5:
        print(p)

0 : (0,inf) <3050,-1>
1 : (0.00378158,0.981684) <13747,5000>
```

1.3.7 Linear Algebra

Fields

The fields `F2 = ModP<int, 2>`, `F3 = ModP<int, 3>`, and `F5 = ModP<int, 5>` are supported. Additional fields can be added to `libbats.cpp` if desired.

```
from bats import F2
print(F2(1) + F2(1)) # should be 0
```

Vectors

Sparse vectors for each supported field class are available: F2Vector, F3Vector, etc. as well as an IntVector.

The easiest way to construct these vectors is from a list of tuples, where each tuple contains a index-value pair (where the value is an integer - it will be cast to the relevant field).

```
from bats import F2Vector
v = F2Vector([(0,1), (2,1)])
```

Matrices

CSCMatrix

```
A = bats.CSCMatrix(2,2,[0,1,2],[0,1],[-1,-1])
A.print() # prints matrix
A(0,0) # returns -1
```

To construct a CSCMatrix using `scipy.sparse`:

```
import scipy.sparse as sparse

# create 2x2 identity matrix
data = [1, 1]
row = [0,1,2]
col = [0, 1]
A = sparse.csc_matrix((data, col, row), shape=[2,2])

# create BATS CSCMatrix
Ab = bats.CSCMatrix(*A.shape, A.indptr, A.indices, A.data)
```

You could also construct the `scipy.sparse csc_matrix` in a variety of different ways before passing to BATS.

Column Matrices

Column matrices can be created from CSCMatrix

```
A = bats.CSCMatrix(2,2,[0,1,2],[0,1],[1,1]) # 2x2 identity
C = bats.IntMat(A)
C2 = bats.F2Mat(A)
C3 = bats.F3mat(A)
CQ = bats.RationalMat(A)
```

You can also use `bats.Mat` and pass in the field.

```
C = bats.Mat(A) # IntMat
C = bats.Mat(A, bats.F2()) # F2Mat
```

In order to get the contents of a ColumnMatrix in Python, use the `tolist()` method

```
C.tolist()
```

You can add columns of the appropriate type to a column matrix

Basic Applied Topology Subprograms

```
A = bats.F2Mat(3,0)
A.append_column(bats.F2Vector([(0,1), (1,1)]))
A.append_column(bats.F2Vector([(0,1), (2,1)]))
A.append_column(bats.F2Vector([(1,1), (2,1)]))
np.array(A.tolist()) # this will display the matrix in a nice way
```

To generate an identity matrix, just pass the desired size and relevant field type to `bats.Identity`

```
I = bats.Identity(3, bats.F2())
```

Dense Matrices

BATS dense matrices are by default stored in row major order to be compatible with numpy.

```
import bats
import numpy as np
Bnp = np.array([[0,1,2],[3,4,5]], dtype=np.float)
B = bats.Matrix(Bnp)
Bnp2 = np.array(B)
```

1.3.8 Maps

Topological Maps

All topological constructions use `bats.CellularMap` to represent topological maps.

To specify a `CellularMap`, you need to provide a map for cells in each dimension. These should be stored as `bats.IntMat`, which provide oriented boundaries. First let's define a `CellComplex` representing the circle with 2 vertices and 2 edges.

```
import numpy as np
import bats

X = bats.CellComplex()
X.add_vertices(2)
X.add([[-1,1],[1,-1],1])
X.add([[-1,1],[1,-1],1])
```

We can verify that `X` has the expected betti numbers mod-2

```
X2 = bats.Chain(X, bats.F2())
R2 = bats.ReducedF2ChainComplex(X2)
print(R2.hdim(0), R2.hdim(1)) # 1, 1
```

Now, we'll define a map from the cell complex to itself, via a doubling. Each vertex maps to (0), and each edge maps to the sum of edges

```
import scipy.sparse as sparse
M = bats.CellularMap(1)
M0_dense = np.array([[1,1],[0,0]])
A = sparse.csc_matrix(M0_dense)
```

(continues on next page)

(continued from previous page)

```
M[0] = bats.Mat(bats.CSCMatrix(*A.shape, A.indptr, A.indices, A.data))
M1_dense = np.array([[1,1],[1,1]])
A = sparse.csc_matrix(M1_dense)
M[1] = bats.Mat(bats.CSCMatrix(*A.shape, A.indptr, A.indices, A.data))
```

`M` now contains the map that we want to represent. We can now apply the chain functor

```
M2 = bats.Chain(M, bats.F2())
```

And to compute induced maps, we need to supply a `ReducedChainComplex` for both the domain and range of the map. We computed these above. The output is a `bats` column matrix.

```
for dim in range(2):
    Mtildes = bats.InducedMap(M2, R2, R2, dim)
    print(Mtildes.tolist()) # [[1]], [[0]]
```

Note the doubling map on the circle creates the zero map on H_1 .

Algorithmic Constructions

There are a variety of common situations in which a `CellularMap` can be constructed algorithmically. `bats` provides a `SimplicialMap` and `CubicalMap` for `SimplicialComplex` and `CubicalComplex` types.

SimplicialMap

A simplicial map f is extended from a map on zero-cells of simplicial complexes. Let's create a noisy circle data set for example.

```
import numpy as np

def gen_circle(n, r=1.0, sigma=0.1):
    X = np.random.randn(n, 2)
    X = r * X / np.linalg.norm(X, axis=1).reshape(-1, 1)
    X += sigma * np.random.randn(n, 2)
    return X

np.random.seed(0)

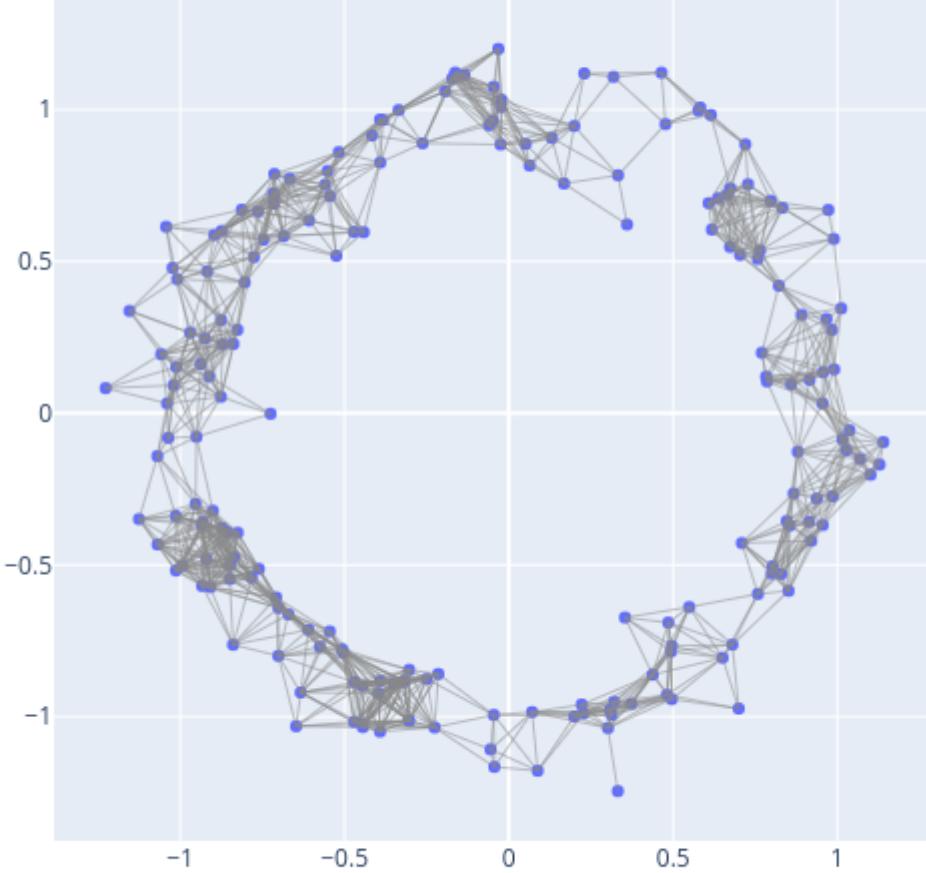
X = gen_circle(200)
```

Now we'll create a `SimplicialComplex` using the Rips construction

```
from bats.visualization.plotly import ScatterVisualization
import scipy.spatial.distance as distance

pdist = distance.squareform(distance.pdist(X, 'euclidean'))

R = bats.RipsComplex(bats.Matrix(pdist), 0.25, 2)
fig = ScatterVisualization(R, pos=X)
fig.update_layout(width=600, height=600, showlegend=False)
fig.show()
```



We now can create an inclusion map (identity map) via

```
f = bats.SimplicialMap(R, R)
```

Now, we can compute the induced map on homology to see we get the identity on H1:

```
R2 = bats.ReducedChainComplex(R, bats.F2())
F2 = bats.Chain(f, bats.F2())
Ftil = bats.InducedMap(F2, R2, R2, 1)
Ftil.tolist() # [[1, 0], [0, 1]]
```

Let's now do a non-inclusion SimplicialMap. We'll get a greedy cover of the data, and threshold to k points.

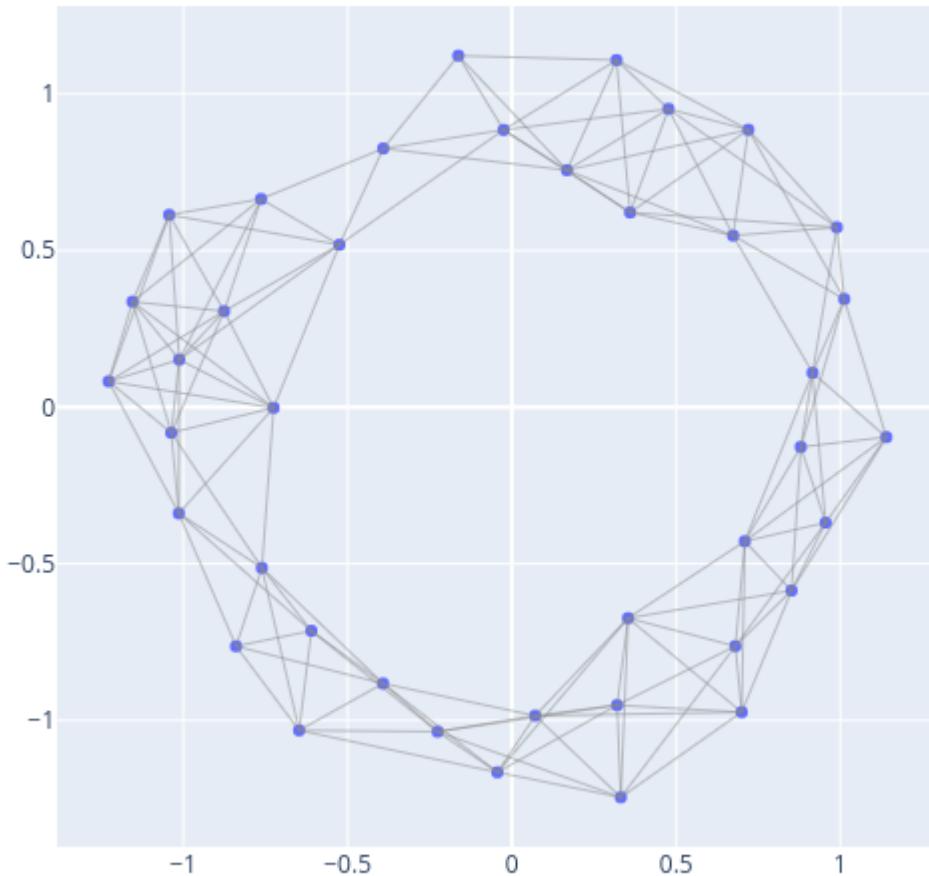
We'll construct a new Rips complex, where the parameter is increased by twice the Hausdorff distance to the full set.

```

k = 40
inds, dists = bats.greedy_landmarks_hausdorff(bats.Matrix(pdist), 0)
inds = inds[:k]
eps = dists[k-1]
eps # hausdorff distance from subset to total data set

Xk = X[inds]
pdist_k = np.array(pdist[inds][:,inds], copy=True)
Rk = bats.RipsComplex(bats.Matrix(pdist_k), 0.25 + 2*eps, 2)
fig = ScatterVisualization(Rk, pos=Xk)
fig.update_layout(width=600, height=600, showlegend=False)
fig.show()

```



We'll now define a map from the full data set to the sub-sampled data by sending points to their nearest neighbor

Basic Applied Topology Subprograms

```
from scipy.spatial import cKDTree
tree = cKDTree(Xk)
ds, f0 = tree.query(X, k=1)
```

f_0 is now the map for vertices of R to vertices of R_k . We can extend the map

```
f = bats.SimplicialMap(R, Rk, f0)
```

Now, we can go through the process of computing the induced map on homology

```
Rk2 = bats.ReducedChainComplex(Rk, bats.F2())
F2 = bats.Chain(f, bats.F2())
Ftil = bats.InducedMap(F2, R2, Rk2, 1)
Ftil.tolist() # [[0, 1]]
```

We see the small H_1 generator is killed.

We can visualize this (See visualization for details).

```
from bats.visualization.plotly import MapVisualization
fig = MapVisualization(pos=(X, Xk), cpx=(R, Rk), maps=(f,))
fig.show_generator(0, color='green', group_suffix=0)
fig.show_generator(1, color='red', group_suffix=1)
fig.show()
```

Generator

CubicalMap

Right now, `bats.CubicalMap` only supports inclusions - syntax is the same as that for `bats.SimplicialMap`, but inputs are `bats.CubicalComplex` objects.

1.3.9 Chain Complexes and Reduction

Chain Complexes

A chain complex can be obtained from a simplicial or cell complex via the chain functor

```
X = bats.SimplicialComplex()
X.add([0])
X.add([1])
X.add([0, 1])

C2 = bats.F2ChainComplex(X)
C3 = bats.F3ChainComplex(X)
```

You can also use `bats.Chain`:

```
C2 = bats.Chain(X, bats.F2())
C3 = bats.Chain(X, bats.F3())
```

Reduced Chain Complex

```
R2 = bats.ReducedF2ChainComplex(C2)

R2.hdim(1) # = 1

v = bats.F2Vector([1], [bats.F2(1)])
print(v[0], v[1], v[2]) # 0 1 0

R2.find_preferredRepresentative(v, 0)
print(v[0], v[1], v[2]) # 1 0 0

# get preferred rep for first basis element in dim 0
v = R2.get_preferredRepresentative(0, 0)
print(v[0], v[1], v[2]) # 1 0 0
```

bats.reduce

You can use `bats.reduce` to create a reduced chain complex from either a `SimplicialComplex`, or `ChainComplex`. For a chain complex:

```
C = bats.Chain(X, bats.F2())
R = bats.reduce(C)
```

You can also skip the explicit chain complex construction for `SimplicialComplex`

```
R = bats.reduce(X, bats.F2())
```

Reduction Flags

You can provide a variety of flags to `bats.reduce` which govern behavior of the reduction algorithm.

Algorithm Flags

- `bats.standard_reduction_flag()` - standard reduction algorithm
- `bats.extra_reduction_flag()` - eliminates all entries to the right of a pivot, even when not necessary for reduction

Optimization Flags

- `bats.clearing_flag()` - performs clearing optimization
- `bats.compression_flag()` - performs compression optimization

Basis Flags By default, when you pass in flags, the basis is not computed, which is fine when you just want the betti numbers or a persistence diagram. If you want to compute the basis, you can use `bats.compute_basis_flag()`, with either no optimizations, or the compression optimization. You can not compute the basis with the clearing optimization.

Valid Combinations When using flags, you must always first provide an algorithm flag. This is followed by an optional algorithm flag, and then an optional basis flag.

Basic Applied Topology Subprograms

```
flags = (
    bats.standard_reduction_flag(), bats.compression_flag(), bats.compute_basis_flag()
)

R = bats.reduce(X, bats.F2(), *flags)
```

- You can only get homology generators if you compute a basis
- You must compute a basis and not use optimizations to compute induced maps.

Performance Computing a basis will always be slower than not computing a basis.

Clearing or compression optimizations will almost always be faster than not using them. Which optimization is better can be problem dependent.

The choice of reduction algorithm can also affect performance. The `bats.extra_reduction_flag()` is sometimes faster than the standard reduction - this may be because it encourages sparsity.

As a suggestion, try using

```
bats.standard_reduction_flag(), bats.compression_flag()
```

and see how this compares to just using

```
bats.standard_reduction_flag()
```

Reducing Matrices Manually

At a lower level, you can use the reduction algorithm on a matrix

```
A = bats.F2Mat(3,0)
A.append_column(bats.F2Vector([(0,1), (1,1)]))
A.append_column(bats.F2Vector([(0,1), (2,1)]))
A.append_column(bats.F2Vector([(1,1), (2,1)]))
U = bats.Identity(3, bats.F2())

R =
p2c = bats.reduce_matrix(A, U)
```

This will modify the matrices A and U in-place so A is reduced, and U is the applied change of basis to columns. I.e. it maintains the invariant `A * inv(U)`. `p2c` will be the pivot-to-column map for the reduced matrix.

1.3.10 Cohomology

The BATS interface for cohomology is currently under development.

Let's look at the cohomology algorithm applied to a Vietoris-Rips filtration

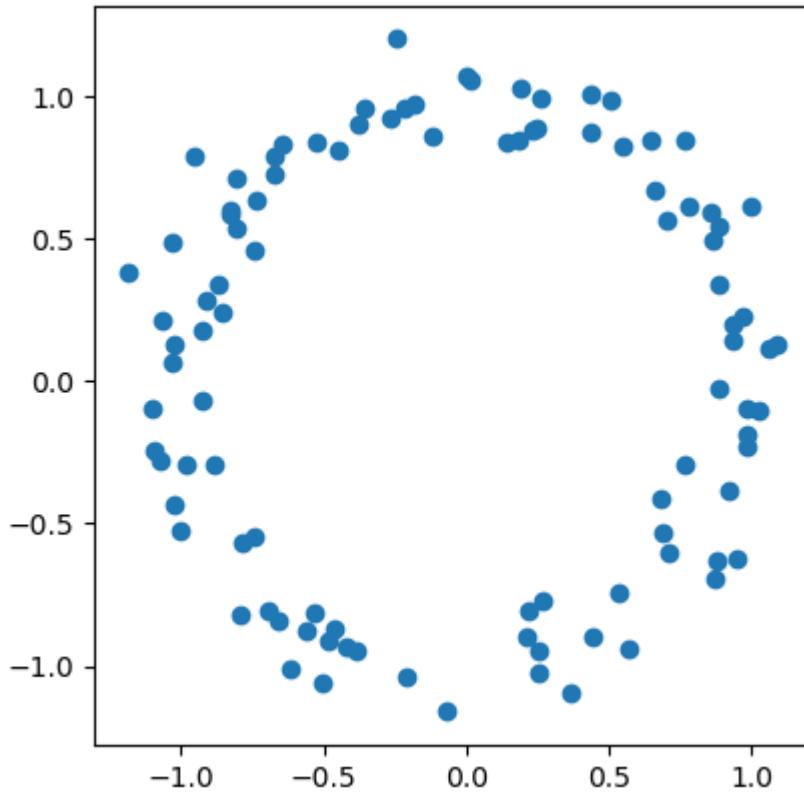
```
[1]: import bats
import numpy as np
import matplotlib.pyplot as plt
import time

n = 100
```

(continues on next page)

(continued from previous page)

```
np.random.seed(0)
X = np.random.normal(size=(n,2))
X = X / np.linalg.norm(X, axis=1).reshape(-1,1)
X = X + np.random.normal(size=(n,2), scale = 0.1 )
fig = plt.scatter(X[:,0], X[:,1])
fig.axes.set_aspect('equal')
plt.show(fig)
```

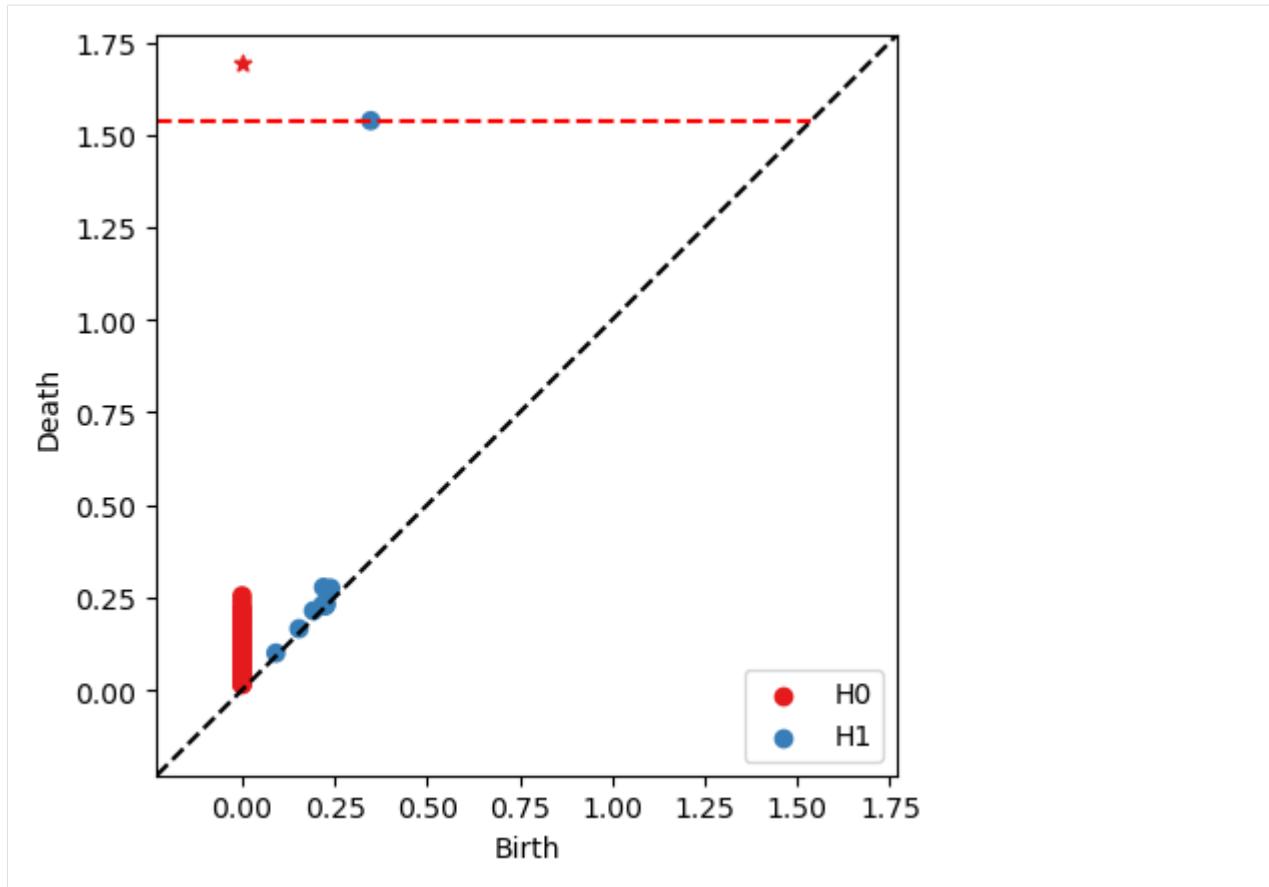


The standard interface for computing persistent homology is through bats.reduce

```
[2]: R = bats.RipsFiltration(bats.DataSet(bats.Matrix(X)), bats.Euclidean(), np.inf, 2)
t0 = time.monotonic()
RR = bats.reduce(R, bats.F2(), bats.standard_reduction_flag(), bats.clearing_flag())
t1 = time.monotonic()
print("{} sec. elapsed".format(t1 - t0))

0.260012122002081 sec. elapsed
```

```
[3]: ps = RR.persistence_pairs(0) + RR.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



```
[4]: for p in ps:
    if p.length() > 0.5:
        print(p)

0 : (0,inf) <0,-1>
1 : (0.347661,1.53775) <477,36355>
```

Currently, to compute cohomology, we create a `FilteredDGVectorSpace` (Filtered differential graded vector space) of degree +1.

```
[5]: t0 = time.monotonic()
RC = bats.FilteredF2DGVectorSpace(R, +1)
t1 = time.monotonic()
print("{} sec. elapsed".format(t1 - t0))

0.05527746499865316 sec. elapsed
```

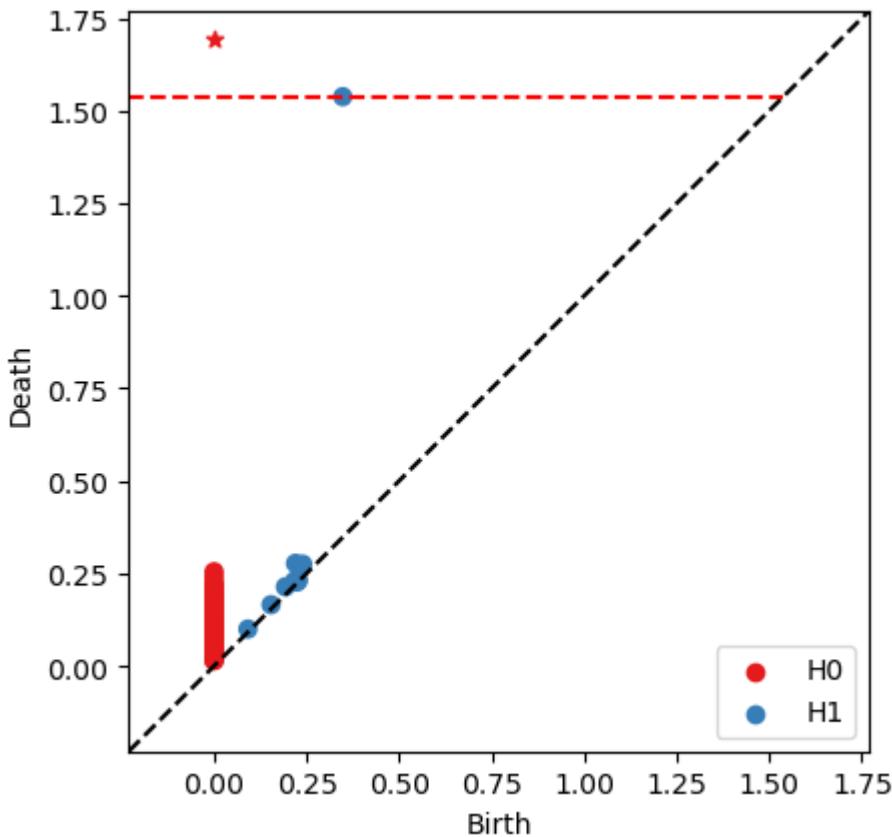
We can then obtain a reduced DGVS

```
[6]: t0 = time.monotonic()
RRC = bats.ReducedFilteredF2DGVectorSpace(RC, bats.standard_reduction_flag(), bats.
    ↵clearing_flag())
t1 = time.monotonic()
print("{} sec. elapsed".format(t1 - t0))

0.016537620998860803 sec. elapsed
```

We see here that we obtain the same persistence barcode

```
[7]: ps = RRC.persistence_pairs(0) + RRC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



```
[8]: for p in ps:
    if p.length() > 0.5:
        print(p)

0 : (0,inf) <0,-1>
1 : (0.347661,1.53775) <477,36355>
```

You can use the same optimization flags as you can with ChainComplexes. For Vietoris-Rips filtrations, standard reduction with the clearing optimization tends to perform best.

```
[9]: flags = [
    (bats.standard_reduction_flag(),),
    (bats.standard_reduction_flag(), bats.clearing_flag()),
    (bats.extra_reduction_flag(),),
    (bats.extra_reduction_flag(), bats.clearing_flag()),

]

for fs in flags:
    print("\n")
```

(continues on next page)

(continued from previous page)

```
print(*fs)
t0 = time.monotonic()
RC = bats.FilteredF2DGVectorSpace(R, +1)
RRC = bats.ReducedFilteredF2DGVectorSpace(RC, *fs)
t1 = time.monotonic()
print("Cohomology: {} sec. elapsed".format(t1 - t0))
tco = t1 - t0

t0 = time.monotonic()
RR = bats.reduce(R, bats.F2(), *fs)
t1 = time.monotonic()
print("Homology:   {} sec. elapsed".format(t1 - t0))
tho = t1 - t0

print("{} ratio".format(tco/tho))
```

```
<bats.linalg.standard_reduction_flag object at 0x7ffb708904f0>
Cohomology: 8.077598232004675 sec. elapsed
Homology:  0.2805818199994974 sec. elapsed
28.788744160327795 ratio
```

```
<bats.linalg.standard_reduction_flag object at 0x7ffb70890530> <bats.linalg.clearing_
←flag object at 0x7ffb70890570>
Cohomology: 0.06173948499781545 sec. elapsed
Homology:  0.293913795001572 sec. elapsed
0.2100598408369544 ratio
```

```
<bats.linalg.extra_reduction_flag object at 0x7ffb708905b0>
Cohomology: 0.22504864599613938 sec. elapsed
Homology:  0.25382902700221166 sec. elapsed
0.8866150914811587 ratio
```

```
<bats.linalg.extra_reduction_flag object at 0x7ffb708905f0> <bats.linalg.clearing_flag_
←object at 0x7ffb70890630>
Cohomology: 0.22575879400392296 sec. elapsed
Homology:  0.22911387599742739 sec. elapsed
0.9853562688907498 ratio
```

Levelset Example

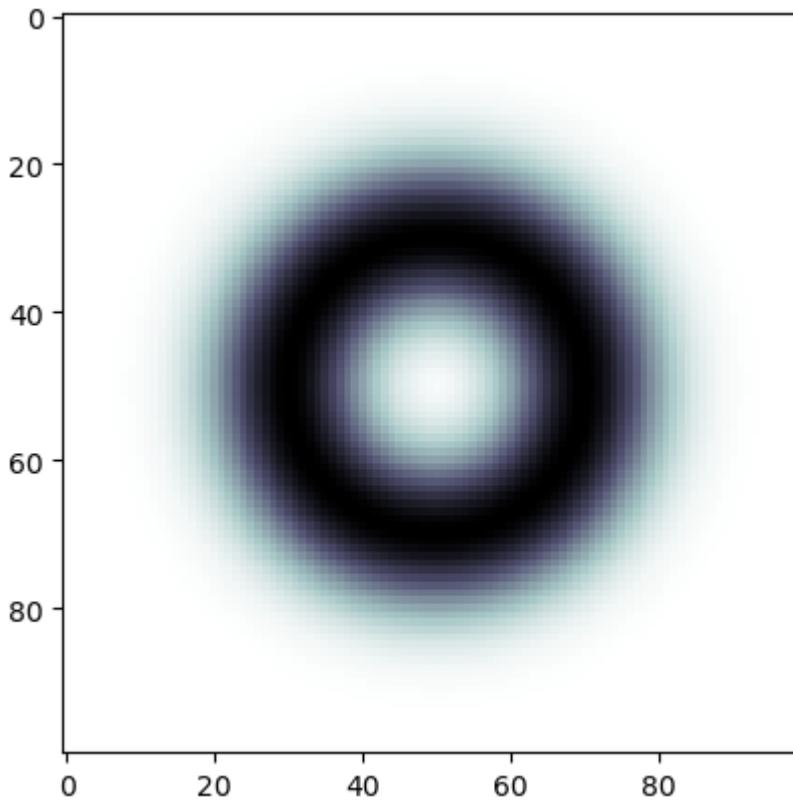
```
[10]: import numpy as np
import matplotlib.pyplot as plt

n = 100
img = np.empty((n,n))

def rad(i,j, n):
    return np.sqrt((i - n/2)**2 + (j - n/2)**2)

for i in range(n):
    for j in range(n):
#        print(i,j, rad(i,j,n))
        img[i,j] = 1-np.exp(-(rad(i,j,n) - 20)**2/100)

plt.imshow(img, cmap='bone')
plt.show()
```



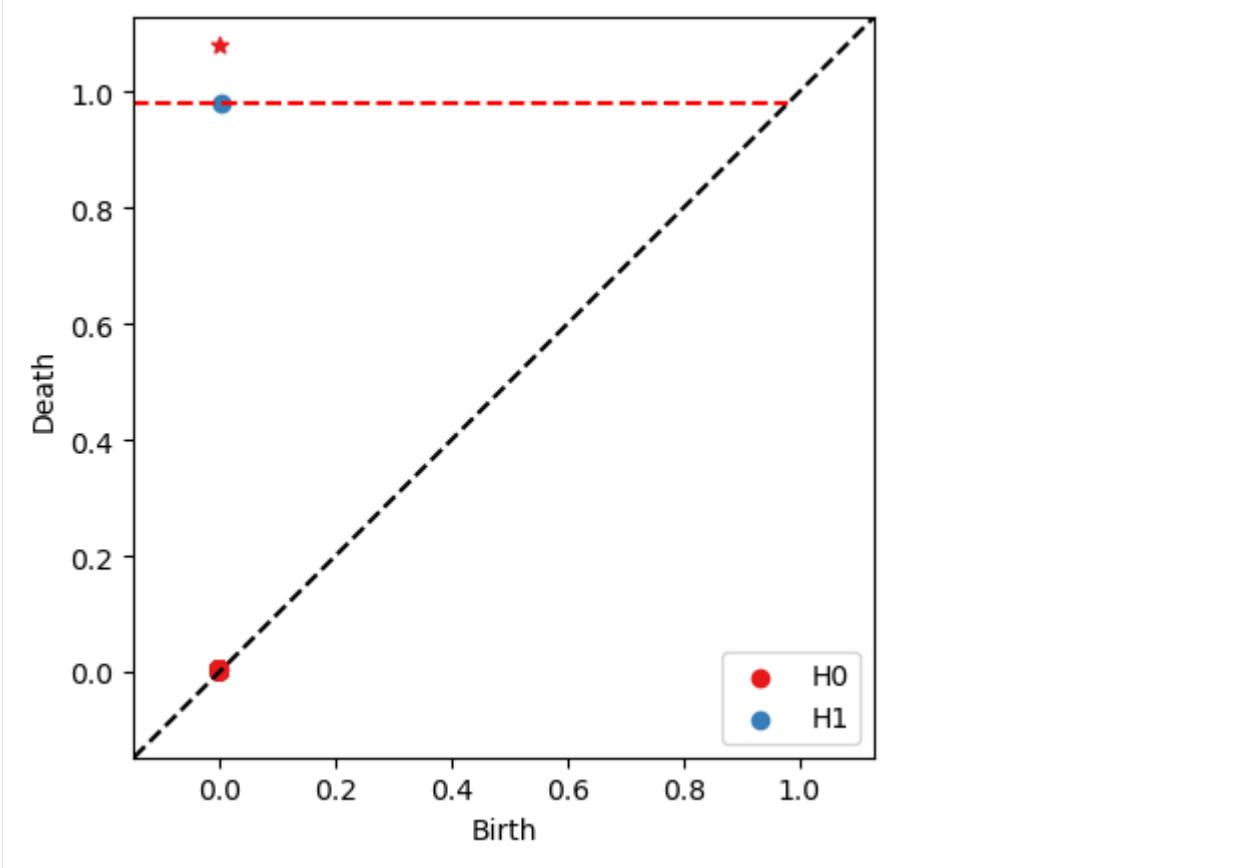
Standard reduction:

```
[11]: X = bats.Freudenthal(n, n)
vals, imax = bats.lower_star_filtration(X, img.flatten()) # computes filtration
# parameter to
F = bats.Filtration(X, vals)
```

(continues on next page)

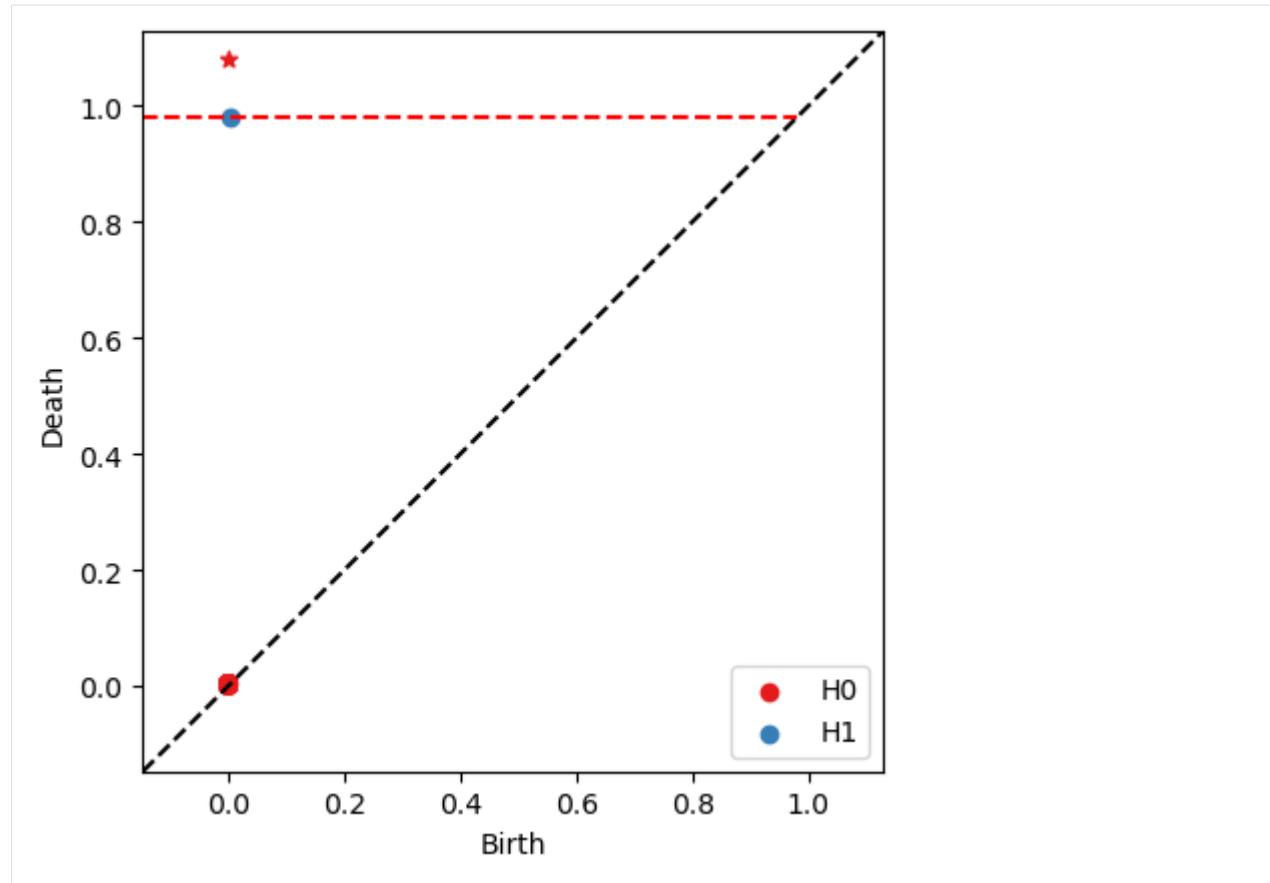
(continued from previous page)

```
RF = bats.reduce(F, bats.F2())
ps = RF.persistence_pairs(0) + RF.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



cohomology:

```
[12]: FC = bats.FilteredF2DGVectorSpace(F, +1)
RF = bats.ReducedFilteredF2DGVectorSpace(FC, *fs)
ps = RF.persistence_pairs(0) + RF.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



For levelset filtrations, cohomology may not perform better than homology:

```
[13]: flags = [
    (bats.standard_reduction_flag(),),
    (bats.standard_reduction_flag(), bats.clearing_flag()),
    (bats.extra_reduction_flag(),),
    (bats.extra_reduction_flag(), bats.clearing_flag()),

]

for fs in flags:
    print("\n")
    print(*fs)
    t0 = time.monotonic()
    RC = bats.FilteredF2DGVectorSpace(F, +1)
    RRC = bats.ReducedFilteredF2DGVectorSpace(RC, *fs)
    t1 = time.monotonic()
    print("Cohomology: {} sec. elapsed".format(t1 - t0))
    tco = t1 - t0

    t0 = time.monotonic()
    RR = bats.reduce(F, bats.F2(), *fs)
    t1 = time.monotonic()
    print("Homology: {} sec. elapsed".format(t1 - t0))
    tho = t1 - t0
```

(continues on next page)

(continued from previous page)

```
print("{} ratio".format(tco/tho))

<bats.linalg.standard_reduction_flag object at 0x7ffb708819b0>
Cohomology: 0.06967114299914101 sec. elapsed
Homology: 0.049795851002272684 sec. elapsed
1.399135502191964 ratio

<bats.linalg.standard_reduction_flag object at 0x7ffb708819f0> <bats.linalg.clearing_
←flag object at 0x7ffb70881a30>
Cohomology: 0.04750711200176738 sec. elapsed
Homology: 0.020269027998438105 sec. elapsed
2.3438278345379064 ratio

<bats.linalg.extra_reduction_flag object at 0x7ffb70881a70>
Cohomology: 0.06759179900109302 sec. elapsed
Homology: 0.03991230500105303 sec. elapsed
1.693507779099947 ratio

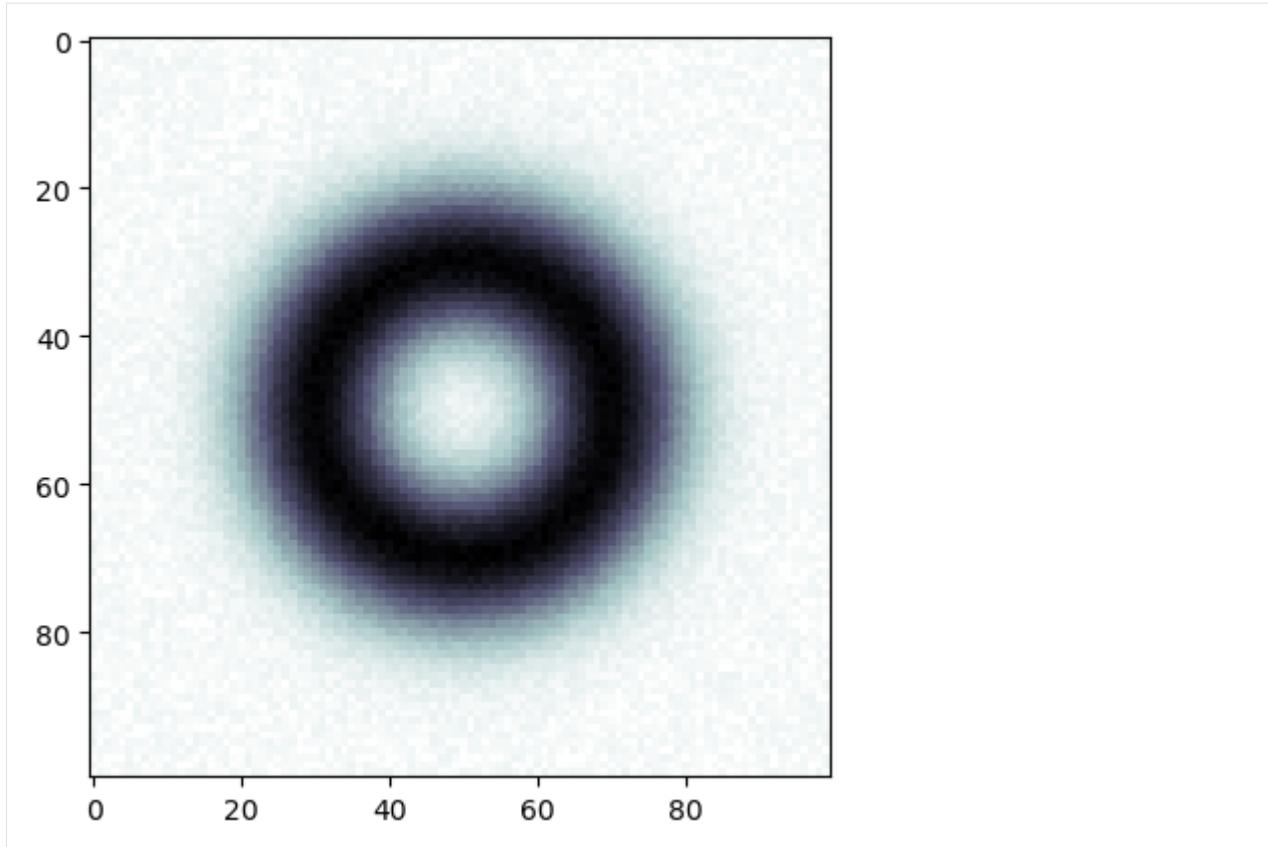
<bats.linalg.extra_reduction_flag object at 0x7ffb70881ab0> <bats.linalg.clearing_flag_
←object at 0x7ffb70881af0>
Cohomology: 0.04434516499895835 sec. elapsed
Homology: 0.02141492500231834 sec. elapsed
2.070759761902394 ratio
```

Updating Filtrations

You can also update ReducedFilteredDGVectorSpace objects with new filtration values

```
[14]: np.random.seed(0)
img2 = img + 0.05*np.random.rand(*img.shape)

plt.imshow(img2, cmap='bone')
plt.show()
```



```
[15]: degree = +1 # homology
# need to compute basis for update.
flags = (bats.standard_reduction_flag(), bats.clearing_flag(), bats.compute_basis_flag())

# original filtration
X = bats.Freudenthal(n, n)
vals, imax = bats.lower_star_filtration(X, img.flatten()) # computes filtration_
˓→parameter to
F = bats.Filtration(X, vals)

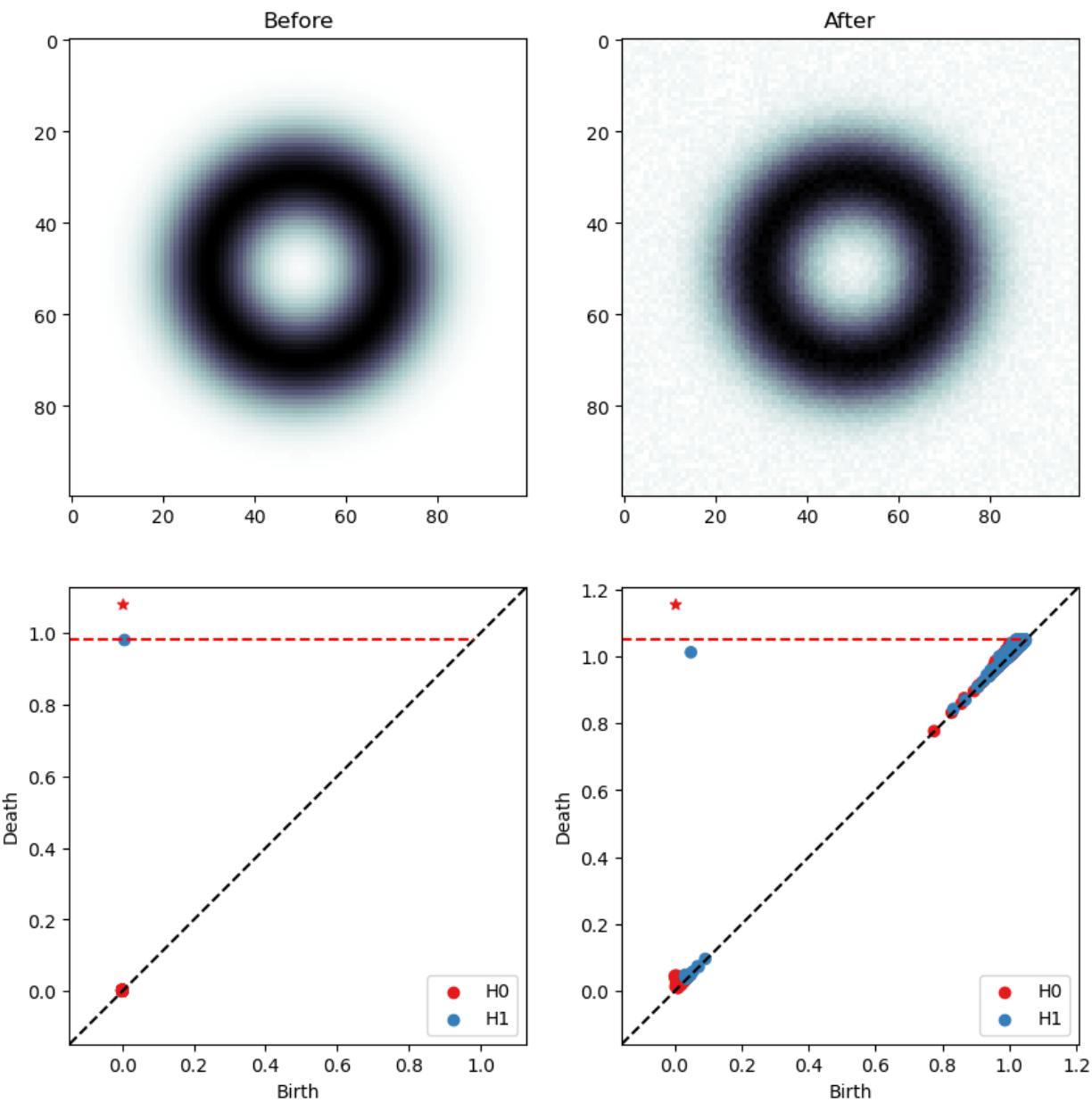
t0 = time.monotonic()
CF = bats.FilteredF2DGVectorSpace(F, degree)
RF = bats.ReducedFilteredF2DGVectorSpace(CF, *flags)
t1 = time.monotonic()
print("{:.3f} sec. to reduce".format(t1 - t0))
ps1 = RF.persistence_pairs(0) + RF.persistence_pairs(1)

vals2, imap2 = bats.lower_star_filtration(X, img2.flatten()) # computes filtration_
˓→parameter to
t0 = time.monotonic()
RF.update_filtration(vals2)
t1 = time.monotonic()
print("{:.3f} sec. to update".format(t1 - t0))
ps2 = RF.persistence_pairs(0) + RF.persistence_pairs(1)
```

Basic Applied Topology Subprograms

```
0.187 sec. to reduce  
0.046 sec. to update
```

```
[16]: fig, ax = plt.subplots(2,2, figsize=(10,10))  
ax[0,0].imshow(img, cmap='bone')  
ax[0,0].set_title('Before')  
ax[0,1].imshow(img2, cmap='bone')  
ax[0,1].set_title('After')  
bats.persistence_diagram(ps1, ax=ax[1,0])  
bats.persistence_diagram(ps2, ax=ax[1,1])  
  
plt.show(fig)
```



[]:

1.3.11 Relative Homology

Relative homology is currently only supported for the `bats.SimplicialComplex` type.

```
[1]: import bats
```

```
[2]: def print_cpx(X):
    for k in range(X.maxdim() + 1):
        print("{}: {} cells".format(k, X.ncells(k)))
```

```
[3]: X = bats.SimplicialComplex()
X.add_recursive((0,1,2))
print_cpx(X)
```

```
0: 3 cells
1: 3 cells
2: 1 cells
```

```
[4]: A = bats.SimplicialComplex()
A.add_recursive((0,1))
A.add_recursive((1,2))
A.add_recursive((0,2))
print_cpx(A)
```

```
0: 3 cells
1: 3 cells
```

Build the relative chain complex $C_*(X, A)$

```
[5]: CXA = bats.Chain(X,A,bats.F2())
```

```
[6]: RXA = bats.reduce(CXA)
```

```
[7]: for k in range(RXA.maxdim()+1):
    print("dim H_{}: {}".format(k, RXA.hdim(k)))
```

```
dim H_0: 0
dim H_1: 0
dim H_2: 1
```

Induced Maps

If we have a map $f : X \rightarrow Y$, $A \subseteq X$ and $B \subseteq Y$ with $f(A) \subseteq B$, then we can compute an induced map on relative homology.

```
[8]: Y = bats.SimplicialComplex()
Y.add_recursive((0,1,2))
Y.add_recursive((1,2,3))
print_cpx(Y)
```

Basic Applied Topology Subprograms

```
0: 4 cells  
1: 5 cells  
2: 2 cells
```

```
[9]: B = bats.SimplicialComplex()  
B.add_recursive((1,2,3))  
B.add_recursive((0,1))  
B.add_recursive((0,2))  
print_cpx(B)
```

```
0: 4 cells  
1: 5 cells  
2: 1 cells
```

Let's now compute Homology

```
[10]: CYB = bats.Chain(Y, B, bats.F2())  
RYB = bats.reduce(CYB)
```

```
[11]: for k in range(RYB.maxdim()+1):  
    print("dim H_{0}: {1}".format(k, RYB.hdim(k)))  
  
dim H_0: 0  
dim H_1: 0  
dim H_2: 1
```

We'll use the inclusion map $X \rightarrow Y$.

```
[12]: f = bats.SimplicialMap(X, Y)
```

```
[13]: F = bats.Chain(f, X, A, Y, B, bats.F2())
```

```
[22]: for k in range(3):  
    print("induced map in dimension {0}".format(k))  
    Ftil = bats.InducedMap(F, RXA, RYB,k)  
    print(Ftil.tolist())  
  
induced map in dimension 0  
[]  
induced map in dimension 1  
[]  
induced map in dimension 2  
[[1]]
```

We see that the induced map is an isomorphism in all dimensions (the first two dimension have 0-dimensional homology).

[]:

1.3.12 Visualization

Visualization of Simplicial Complexes and Generators

In this section, we'll visualize simplicial complexes using `plotly`.

```
import numpy as np
import bats
from bats.visualization.plotly import ScatterVisualization
import scipy.spatial.distance as distance

np.random.seed(0)
```

Let's generate a figure-8 as a data set.

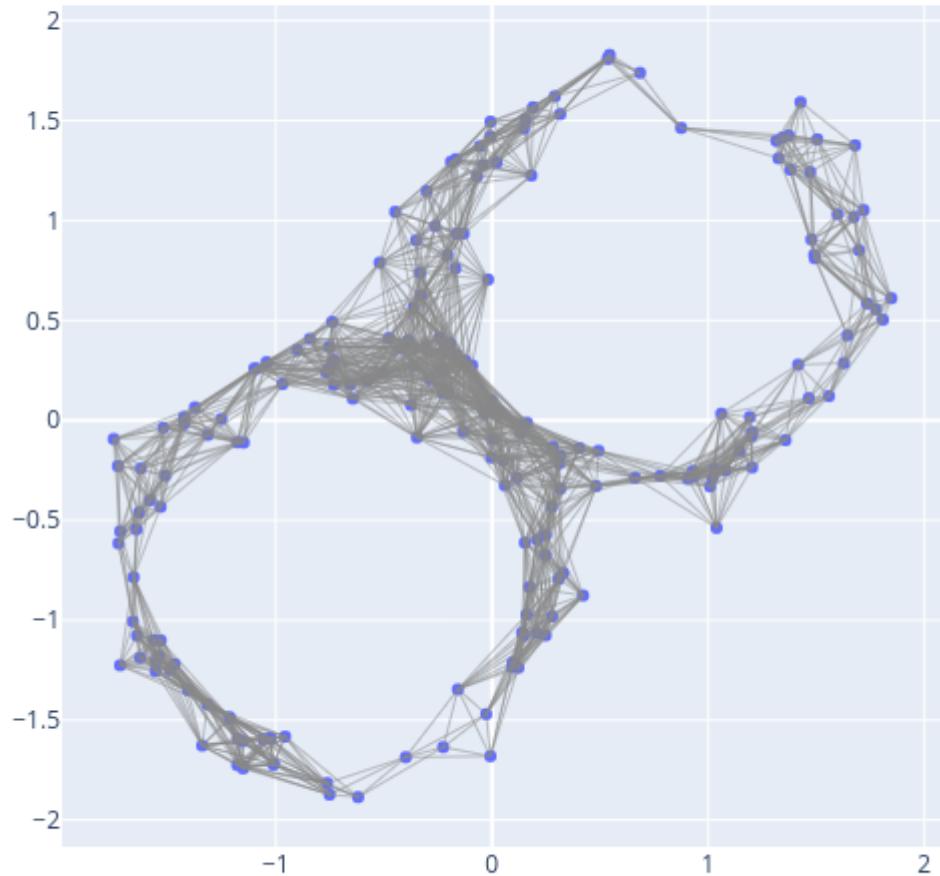
```
def gen_fig_8(n, r=1.0, sigma=0.1):
    X = np.random.randn(n, 2)
    X = r * X / np.linalg.norm(X, axis=1).reshape(-1, 1)
    X += sigma * np.random.randn(n, 2) + np.random.choice([-1/np.sqrt(2), 1/np.sqrt(2)], 
    size=(n, 1))
    return X

n = 200
X = gen_fig_8(n)
```

First, we'll construct a Rips complex on the data.

```
pdist = distance.squareform(distance.pdist(X, 'euclidean'))

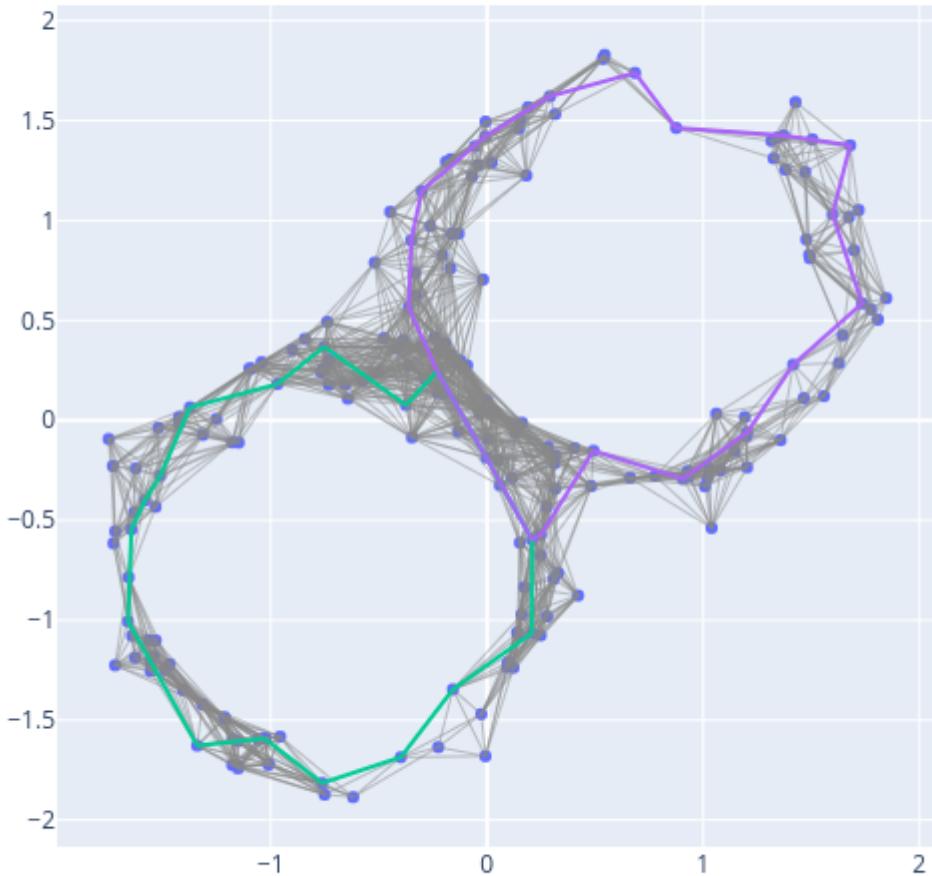
R = bats.RipsComplex(bats.Matrix(pdist), 0.5, 2)
fig = ScatterVisualization(R, pos=X)
fig.update_layout(width=600, height=600, showlegend=False)
fig.show()
```



A `ScatterVisualization` object inherits from a `plotly Figure`, so you can add additional traces, update layout, or call any methods you'd like.

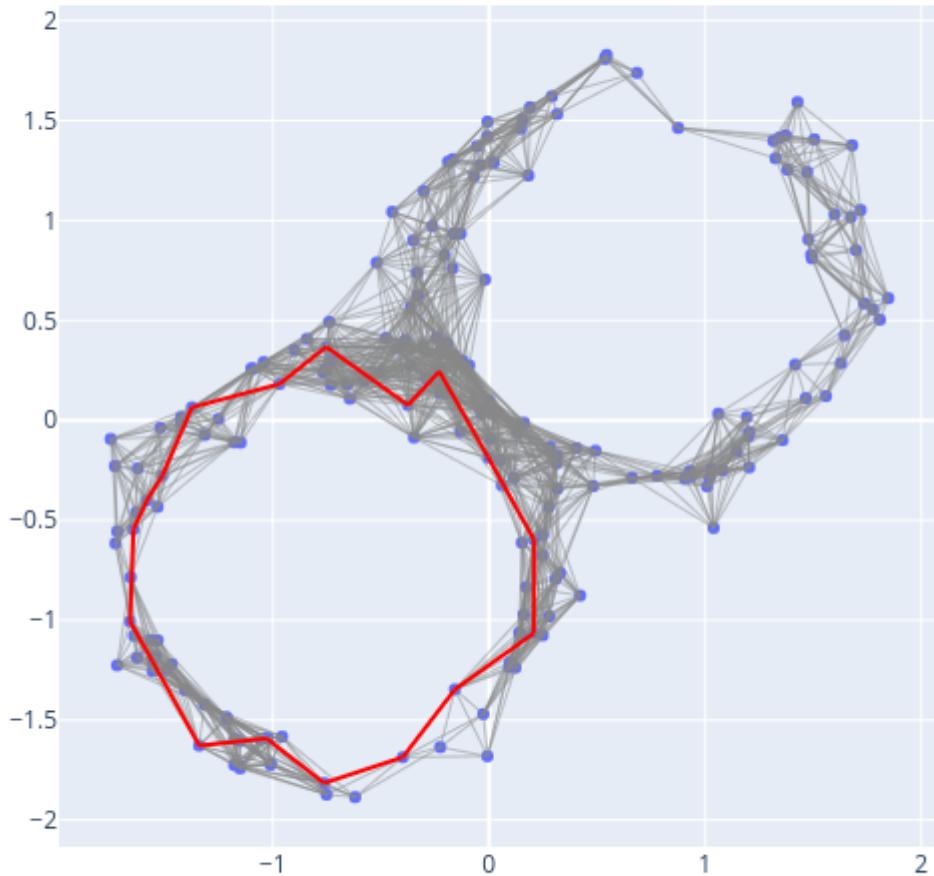
Now, let's visualize generators

```
fig.show_generators(1)  
fig.show()
```



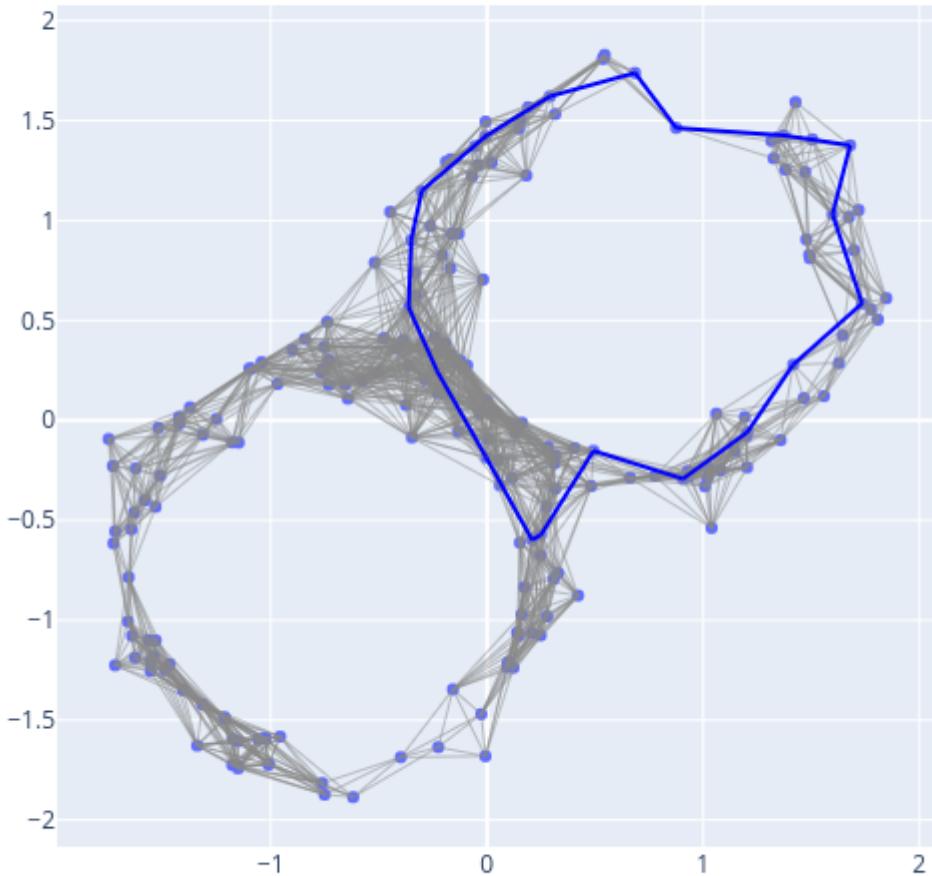
Let's now look at a single generator:

```
fig.reset() # resets figure to have no generators
fig.show_generator(0, hdim=1, color='red')
fig.show()
```



Let's visualize the second generator by passing in the representative 1-chain:

```
RC = bats.ReducedChainComplex(R, bats.F2())
r = RC.get_preferredRepresentative(1, 1)
fig.reset()
fig.show_chain(r, color='blue')
fig.show()
```



Visualization of Maps

You can visualize a `SimplicialMap` with a `MapVisualization`.

```
import numpy as np
import bats
from bats.visualization.plotly import MapVisualization
import scipy.spatial.distance as distance

np.random.seed(0)
```

Let's generate a cylinder in three dimensions:

Basic Applied Topology Subprograms

```
def gen_cylinder(n, r=1.0, sigma=0.1):
    X = np.random.randn(n, 2)
    X = r * X / np.linalg.norm(X, axis=1).reshape(-1, 1)
    X = np.hstack((X, r*np.random.rand(n, 1) - r/2))
    return X

X = gen_cylinder(500)
```

We'll generate a Rips Complex with parameter 0.25

```
pdist = distance.squareform(distance.pdist(X, 'euclidean'))
R = bats.RipsComplex(bats.Matrix(pdist), 0.25, 2)
```

Let's investigate the inclusion of the lower half of the cylinder

```
inds = np.where(X[:, 2] < 0)[0]
Xi = X[inds]
pdisti = distance.squareform(distance.pdist(Xi, 'euclidean'))
Ri = bats.RipsComplex(bats.Matrix(pdisti), 0.25, 2)
```

The inclusion map is

```
M = bats.SimplicialMap(Ri, R, inds)
```

Now, we can construct a visualization

```
fig = MapVisualization(pos=(Xi, X), cpx=(Ri, R), maps=(M,))
fig.update_layout(scene_aspectmode='manual',
                  scene_aspectratio=dict(x=1, y=1, z=0.5))
fig.show()
```

The `show_generator` method will visualize homology generators in the domain by visualizing the preferred representative used in calculations. By default, the image of the chain is visualized in the range, as well as the preferred representative for the homology class.

```
fig.reset()
fig.show_generator(1)
fig.show()
```

The `reset` method clears the visualization of chains/generators. You can also change the color of chains and homology. `group_suffix` can be used to group visualizations in the legend - try using the legend to toggle visualizations below:

```
fig.reset()
fig.show_generator(5, group_suffix=0)
fig.show_generator(3, color='orange', hcolor='black', group_suffix=1)
fig.show()
```

Let's now create a second map from the full data set to a projection onto two coordinates.

```
Xp = X[:, :2]
pdistp = distance.squareform(distance.pdist(Xp, 'euclidean'))
Rp = bats.RipsComplex(bats.Matrix(pdistp), 0.25, 2)
M2 = bats.SimplicialMap(R, Rp) # inclusion map
```

We can visualize the three spaces, with the maps between them, and mix 2-dimensional and 3-dimensional visualizations. Note that the homology class visualized in `show_generator(1)` is killed by the second map.

```
fig = MapVisualization(pos=(Xi,X, Xp), cpx=(Ri,R,Rp), maps=(M,M2))
fig.update_layout(scene_aspectmode='manual',
                  scene_aspectratio=dict(x=1, y=1, z=0.5))
fig.show_generator(1, color='green', hcolor='black', group_suffix=1)
fig.show_generator(5, color='red', hcolor='blue', group_suffix=5)
fig.show()
```

1.3.13 Zigzag Homology

There are two ways to compute zigzag homology in BATS.

1. Zigzag homology of a diagram of spaces over a line graph
2. Zigzag homology of a zigzag-filtered space

(2) is a special case of (1) which uses different data structures and functions for (potential) memory efficiency and performance gains.

Zigzag homology of a diagram of spaces

There are three categories you can use to create a diagram of spaces

1. Simplicial complexes/cellular maps: `SimplicialComplexDiagram`
2. Cubical complexes/cellular maps: `CubicalComplexDiagram`
3. Cell complexes/cellular maps: `CellComplexDiagram`

For zigzag homology, you want to create a diagram of spaces on a directed line graph

```
* --> * <-- * --> * --> ...
```

Where the arrows can go in any direction. This means you will have n nodes (spaces) and $n-1$ edges (maps).

?> It is important to arrange your nodes in order i.e. there should be an edge between node 0 and 1, an edge between 1 and 2, etc. BATS will assume this, but won't check.

Let's use the following example of complexes:

```
0 <- 0 -> 0
 |
 1 <- 1 -> 1
```

The spaces on the left and right are identical (two vertices, numbered “0” and “1” connected by a single edge), and the space in the middle has the edge removed. The maps are identical inclusion maps. We expect a 0-dimensional zigzag barcode that looks like

```
*-----*-----*
      *
```

Simplicial Complexes

```
import bats

n = 3 # number of spaces

D = bats.SimplicialComplexDiagram(n, n-1)

# first, we add SimplicialComplexes to the diagram
X = bats.SimplicialComplex()
X.add_recursive([0,1])
D.set_node(0, X) # left node
D.set_node(2, X) # right node

Y = bats.SimplicialComplex()
Y.add([0])
Y.add([1])
D.set_node(1, Y)

# now, we add SimplicialMaps
f = bats.SimplicialMap(Y, X, [0, 1])
D.set_edge(0, 1, 0, f) # edge 0 maps space at node 1 to space at node 0. The map is f
D.set_edge(1, 1, 2, f) # edge 1 maps space at node 1 to space at node 2. The map is f
```

note that the type of `f` is a `CellularMap`. `SimplicialMap` is just a convenient way to construct cellular maps which are also simplicial maps.

```
type(f)
```

```
bats.libbats.CellularMap
```

Now, we apply the `Chain` functor. This creates a new diagram of `ChainComplexes` and `ChainMaps`. We simply have to provide the diagram of spaces and the field we wish to use. Let's do `F3` coefficients.

```
CD = bats.Chain(D, bats.F3())
```

Now, we apply the `Homology` functor. This creates a new diagram of `ReducedChainComplexes` and induced maps on Homology, stored as matrices.

```
HD = bats.Hom(CD, 0) # 0 is homology dimension
HD.edge_data(0).tolist() # [[1, 1]]
```

```
[[1, 1]]
```

To extract the barcode, we use the `barcode` function. We pass in the homology dimension for book-keeping. The output is a list of `PersistencePairs` which tell us about birth and death indices of homology classes.

```
ps = bats.barcode(HD, 0) # homology dimension is 0
for p in ps:
    print(p) # output is dimension : (birth, death) <birth basis index, death basis_index>
```

```
0 : (0,2) <0,0>
0 : (1,1) <1,1>
```

We see a class that is born at index 0 in the diagram and dies at index 2, and a second class that is only present at index 1, just as we expect.

Cubical Complexes

We'll now do the same example with a diagram of CubicalComplexes

```
n = 3 # number of spaces

D = bats.CubicalComplexDiagram(n, n-1)

# first, we add CubicalComplexes to the diagram
X = bats.CubicalComplex(1) # 1 is dimension of cubical complex
X.add_recursive([0,1])
D.set_node(0, X) # left node
D.set_node(2, X) # right node

Y = bats.CubicalComplex(1)
Y.add([0, 0])
Y.add([1, 1])
D.set_node(1, Y)

# now, we add maps
f = bats.CubicalMap(Y, X) # inclusion of Y into X
D.set_edge(0, 1, 0, f) # edge 0 maps space at node 1 to space at node 0. The map is f
D.set_edge(1, 1, 2, f) # edge 1 maps space at node 1 to space at node 2. The map is f
```

Again, the type of `f` is a `CellularMap` even though we constructed it as a `CubicalMap`.

Again, we then apply the Chain and Hom functors and then extract the barcode

```
CD = bats.Chain(D, bats.F3())
HD = bats.Hom(CD, 0) # 0 is homology dimension
print("Induced map: {}".format(HD.edge_data(0).tolist())) # [[1, 1]]

ps = bats.barcode(HD, 0)
for p in ps:
    print(p) # output is dimension : (birth, death) <birth basis index, death basis_index>
```

```
Induced map: [[1, 1]]
```

```
0 : (0,2) <0,0>
0 : (1,1) <1,1>
```

Cell Complexes

Now we'll use CellComplexes. These are more general, but also not combinatorially defined so they can be a bit more work to deal with if (it's not too bad in this simple example).

```
n = 3 # number of spaces

D = bats.CellComplexDiagram(n, n-1)

X = bats.CellComplex()
X.add_vertices(2)
# first list is boundary indices, second list is boundary coefficients, final argument ↴ is dimension
X.add([0,1], [-1,1], 1)
D.set_node(0, X) # left node
D.set_node(1, X) # right node

Y = bats.CellComplex()
Y.add_vertices(2)
D.set_node(0, Y)

# now we create the map
f = bats.CellularMap(0) # 0-dimensional Cellular map.
f[0] = bats.IntMat(bats.CSCMatrix(2,2,[0,1,2],[0,1],[1,1])) # CSCMatrix to specify map
print("f[0] = {}".format(f[0].tolist())) # identity map on 0-cells
D.set_edge(0, 1, 0, f) # edge 0 maps space at node 1 to space at node 0. The map is f
D.set_edge(1, 1, 2, f) # edge 1 maps space at node 1 to space at node 2. The map is f
```

```
f[0] = [[1, 0], [0, 1]]
```

```
CD = bats.Chain(D, bats.F3())
HD = bats.Hom(CD, 0) # 0 is homology dimension
print("Induced map: {}".format(HD.edge_data(0).tolist())) # [[1, 1]]

ps = bats.barcode(HD, 0)
for p in ps:
    print(p) # output is dimension : (birth, death) <birth basis index, death basis ↴ index>
```

```
Induced map: [[1, 1]]
0 : (0,2) <0,0>
0 : (1,1) <1,1>
```

Zigzag-Filtered Spaces

The above example also works for a zigzag filtered space. You can construct a zigzag filtration in a way that is similar to a regular filtration. You simply need to provide entry and exit times.

Our original example had discrete indices, but zigzag filtrations have continuous parameters. We'll say that the edge was removed for a small interval of radius 0.01 around parameter 1. Note that if a cell is added and removed at the same parameter, both copies of the cell will be considered present at that instant and the zigzag barcode will be different than if you were to remove one copy and then add the other.

```
eps = 0.01 # infinitesimal
X = bats.ZigzagSimplicialComplex()
X.add(0, 2, [0]) # vertex is present for interval [0,2]
X.add(0, 2, [1]) # vertex is present for interval [0,2]
X.add(0, 1-eps, [0,1]) # edge is present at index 0 but not at index 1
X.add(1+eps, 2, [0,1]) # edge is added back and survives until parameter 2
```

You can see the tuples of entry/exit times of cells stored in a single list for each cell

```
X.vals()
```

```
[[[(0.0, 2.0)], [(0.0, 2.0)]], [[(0.0, 0.99), (1.01, 2.0)]]]
```

we can now print the Zigzag barcode

```
ps = bats.ZigzagBarcode(X, 0, bats.F2()) # second argument is maximum homology dimension
for p in ps[0]:
    print(p)
```

The behavior here is a bit different than the diagram. We see that there are some 0-length zigzag bars at parameters 0 and 2 which are due to the addition of the 0-cells before 1-cells. To filter these out, we can do the following:

```
for p in ps[0]:
    if p.length() > 0:
        print(p)
```

```
0 : (0,2) <0(1),0(0)>
0 : (0.99,1.01) <0(0),1(1)>
```

1.3.14 Counting Operations in BATS

Counting field arithmetic operations and column operations can be useful when investigating performance of different algorithms.

To compile BATS to count operations, you need to pass the compile flag `-DBATS_OPCOUNT` when installing BATS.

```
CFLAGS="-DBATS_OPCOUNT" python setup.py build_ext --force -j8
python setup.py install
```

Note that this currently will only work with the F2 field, and you must produce a `ReducedChainComplex` without going through the `reduce` interface. It will also slow down the code, so it is not turned on by default.

Basic Applied Topology Subprograms

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import scipy.spatial.distance
import bats

np.random.seed(0)
```

```
[2]: bats.reset_field_ops()
bats.reset_column_ops()
bats.get_field_ops(), bats.get_column_ops()

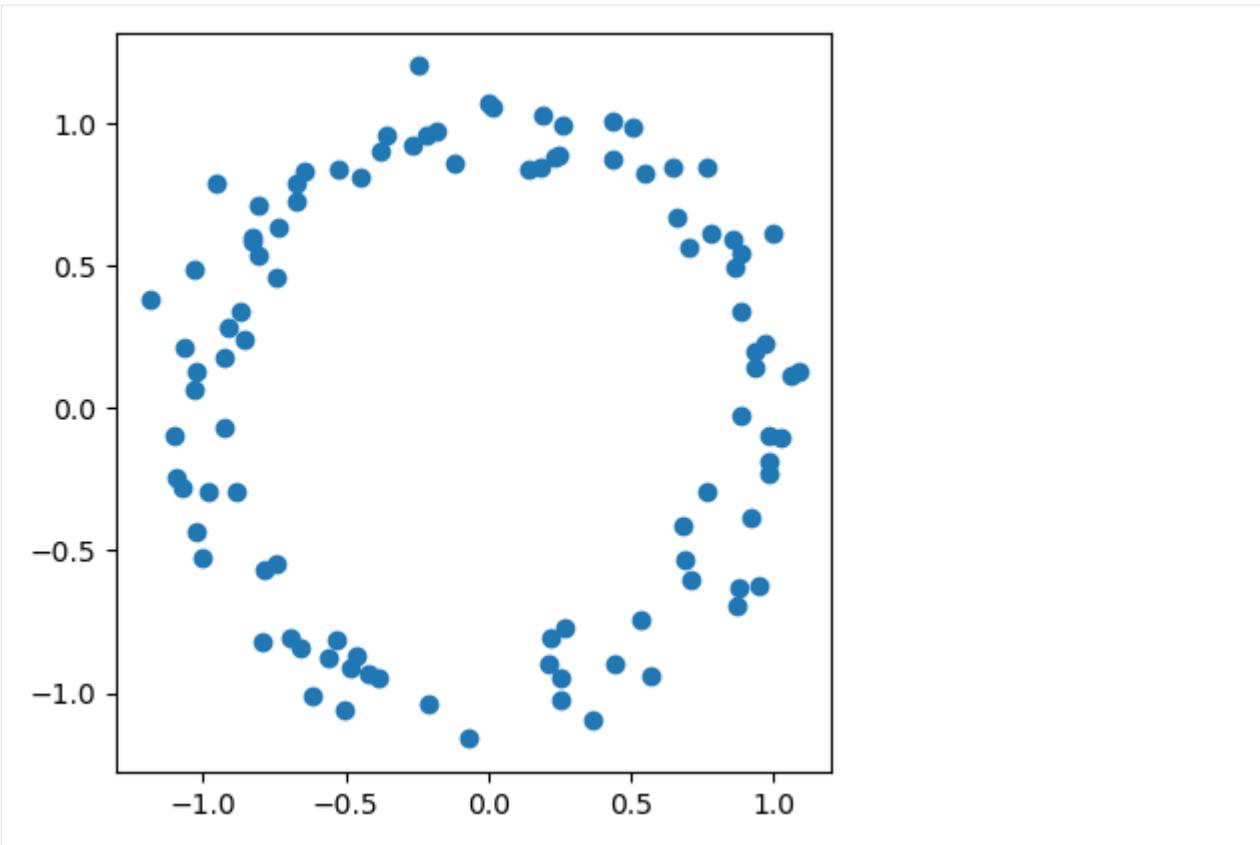
[2]: (0, 0)
```

```
[3]: a = bats.F2(1)
b = a + a
bats.get_field_ops()

[3]: 1
```

Example with Rips Reduction

```
[4]: # first, generate a circle
n = 100
X = np.random.normal(size=(n,2))
X = X / np.linalg.norm(X, axis=1).reshape(-1,1)
X = X + np.random.normal(size=(n,2), scale = 0.1 )
fig = plt.scatter(X[:,0], X[:,1])
fig.axes.set_aspect('equal')
# plt.savefig('RipsEx_data.png')
plt.show(fig)
```



```
[5]: data = bats.DataSet(bats.Matrix(X)) # put into a bats.DataSet
dist = bats.Euclidean() # distance we would like to use
F = bats.RipsFiltration(data, dist, np.inf, 2) # generate a RipsFiltration
```

```
[6]: bats.reset_field_ops(), bats.reset_column_ops()
C = bats.FilteredF2ChainComplex(F)
bats.get_field_ops(), bats.get_column_ops()
```

```
[6]: (0, 0)
```

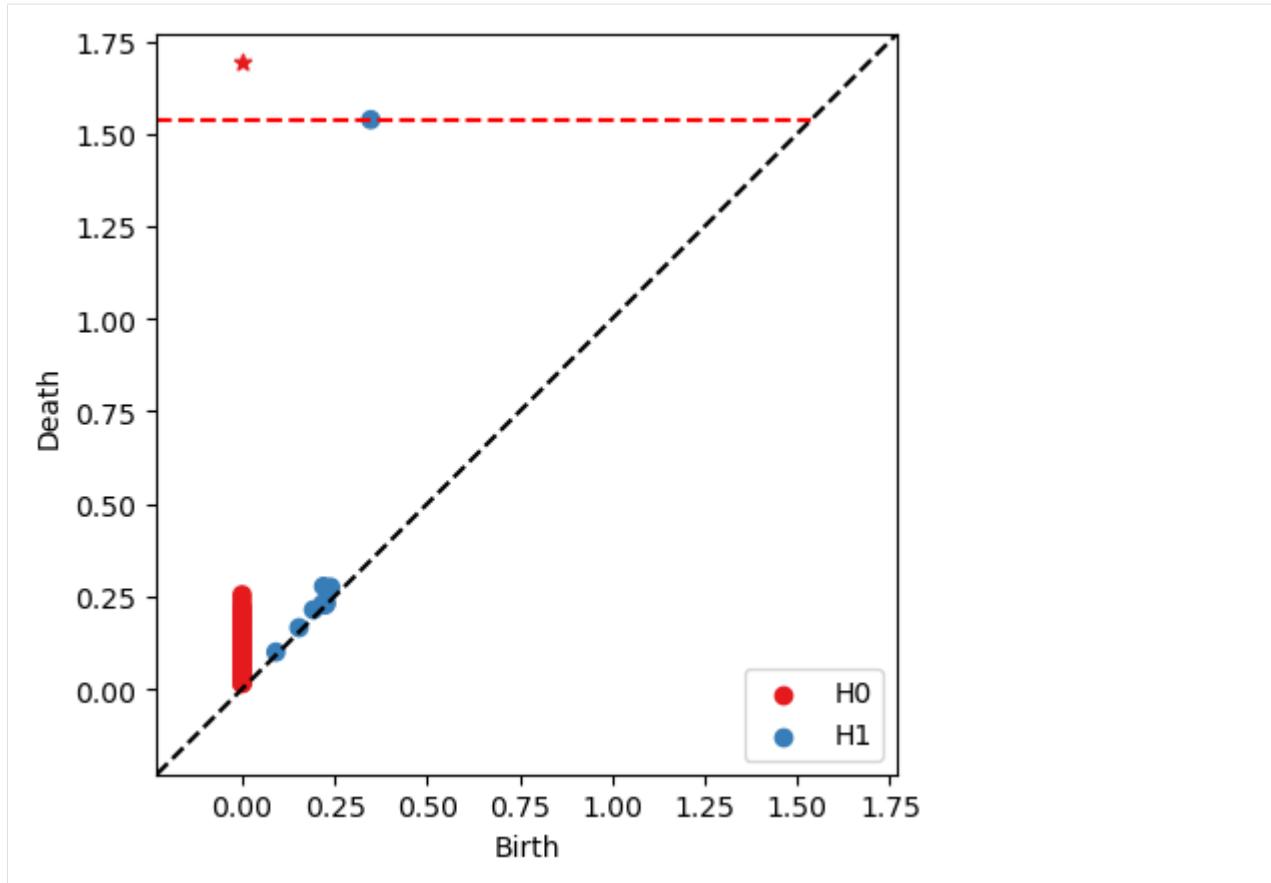
Options for the Reduction Algorithm

First, we'll try the standard reduction algorithm (with basis)

```
[7]: bats.reset_field_ops(), bats.reset_column_ops()
RC = bats.ReducedFilteredF2ChainComplex(C)
print("field operations: {}".format(bats.get_field_ops()))
print("column operations: {}".format(bats.get_column_ops()))

field operations: 19759887
column operations: 4984028
```

```
[8]: ps = RC.persistence_pairs(0) + RC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```

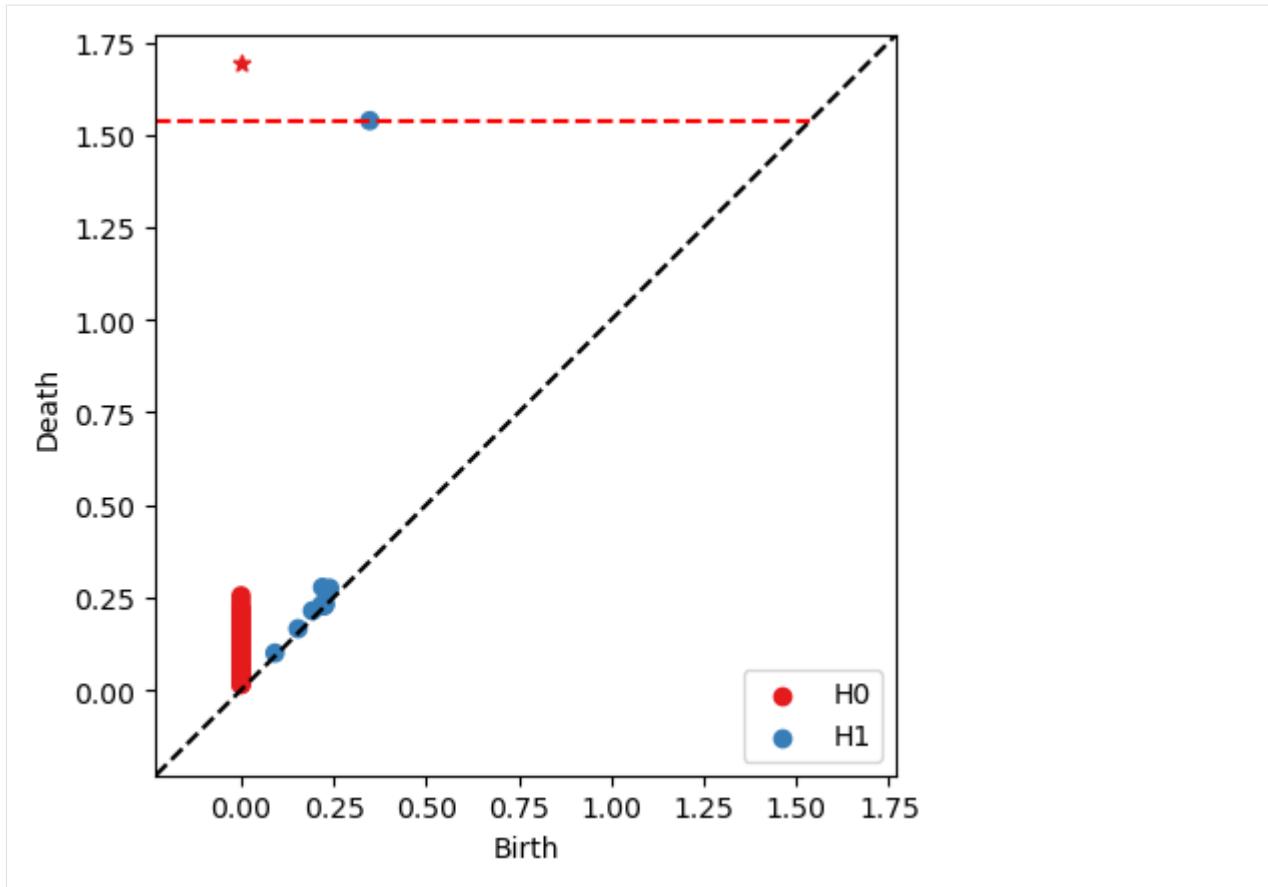


Next the standard reduction algorithm with no basis - the column operations are cut in half.

```
[9]: bats.reset_field_ops(), bats.reset_column_ops()
RC = bats.ReducedFilteredF2ChainComplex(C, bats.standard_reduction_flag())
print("field operations: {}".format(bats.get_field_ops()))
print("column operations: {}".format(bats.get_column_ops()))

field operations: 15465396
column operations: 2492014
```

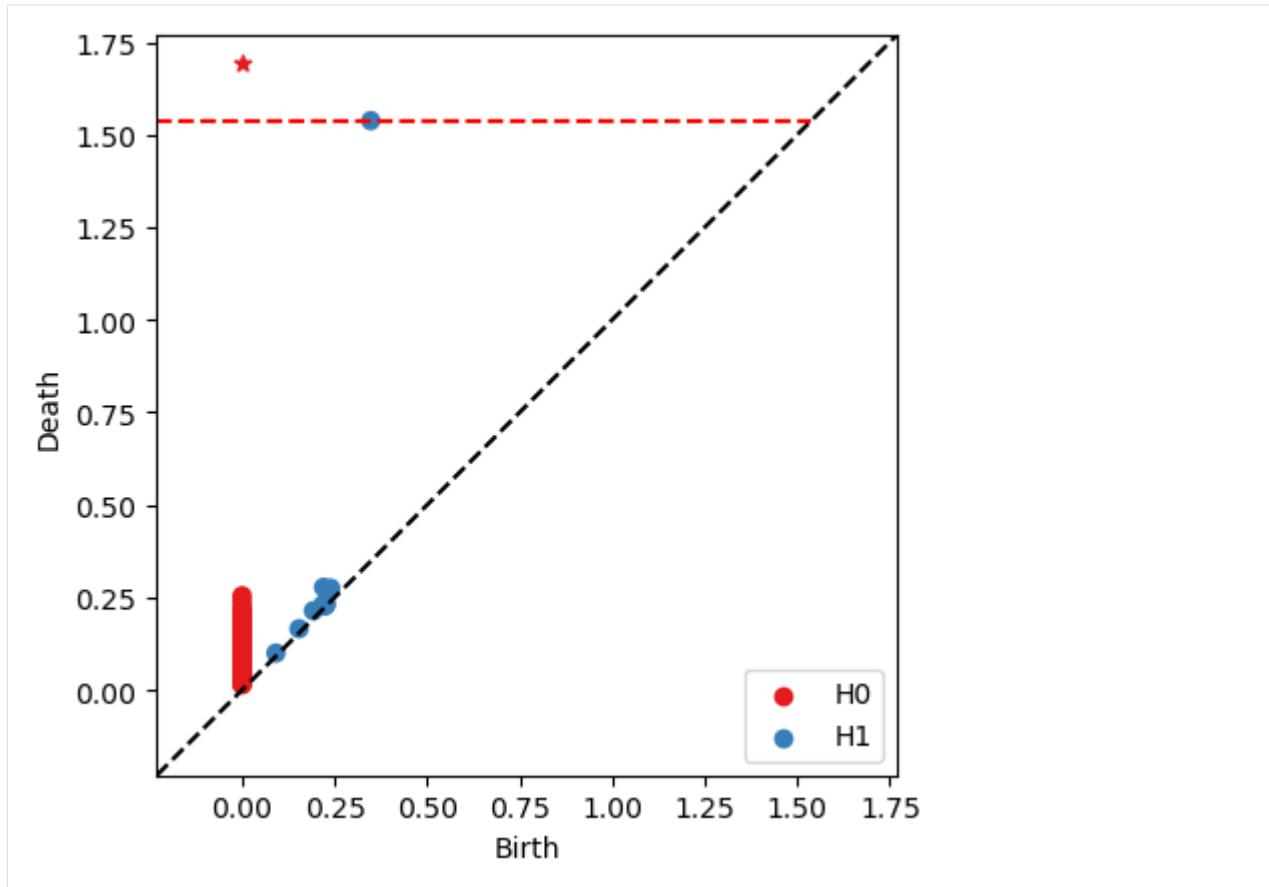
```
[10]: ps = RC.persistence_pairs(0) + RC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



Now, let's try the clearing optimization

```
[11]: bats.reset_field_ops(), bats.reset_column_ops()
RC = bats.ReducedFilteredF2ChainComplex(C, bats.standard_reduction_flag(), bats.clearing_
    _flag())
print("field operations: {}".format(bats.get_field_ops()))
print("column operations: {}".format(bats.get_column_ops()))
field operations: 15341005
column operations: 2462129
```

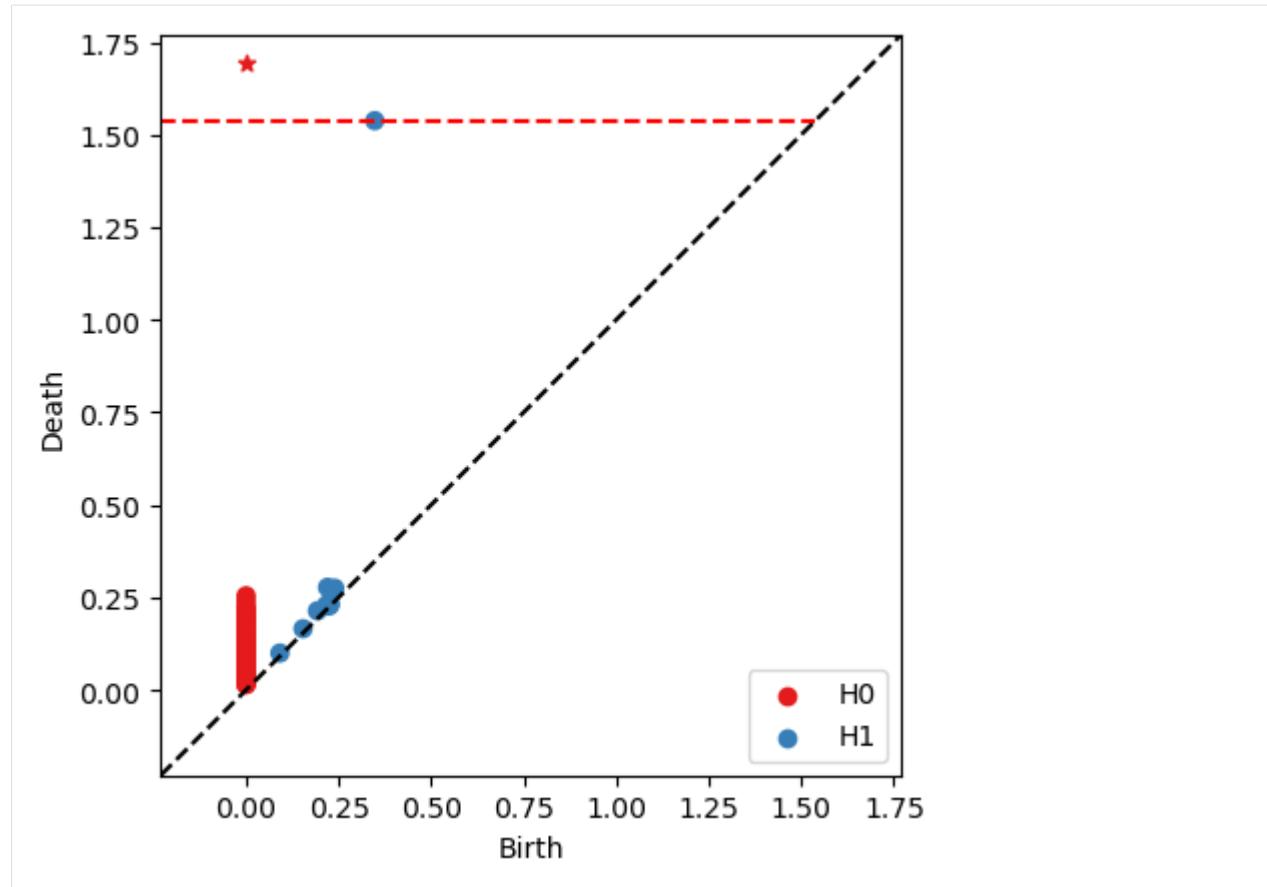
```
[12]: ps = RC.persistence_pairs(0) + RC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



And finally, the extra reduction flag with clearing really decreases the number of column operations

```
[13]: bats.reset_field_ops(), bats.reset_column_ops()
RC = bats.ReducedFilteredF2ChainComplex(C, bats.extra_reduction_flag(), bats.clearing_
    _flag())
print("field operations: {}".format(bats.get_field_ops()))
print("column operations: {}".format(bats.get_column_ops()))
field operations: 19686497
column operations: 496491
```

```
[14]: ps = RC.persistence_pairs(0) + RC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



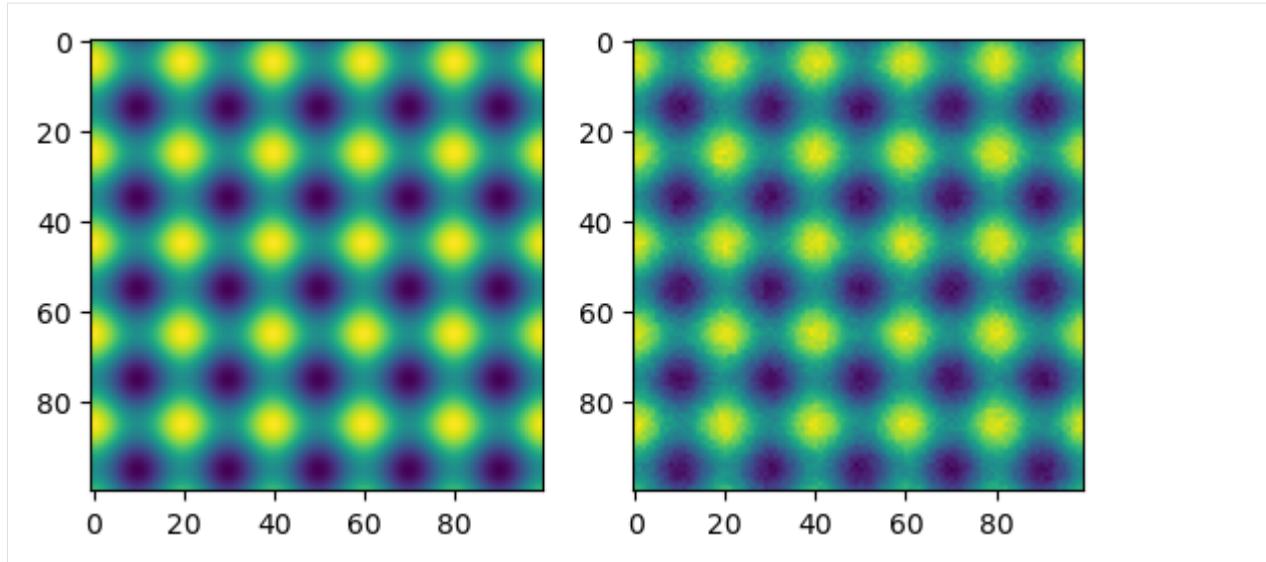
Example Updating Persistence

In this example, we update the level set persistence of an image

```
[15]: n = 100

img = np.empty((n,n), dtype=np.float64)
for i in range(n):
    for j in range(n):
        img[i,j] = np.sin(10* np.pi * i / n) + np.cos(10* np.pi * j/n)

img2 = img + 0.1 * np.random.randn(n,n)
fig, ax = plt.subplots(1,2)
ax[0].imshow(img)
ax[1].imshow(img2)
plt.show()
```



First, we compute the reduced chain complex for the original image

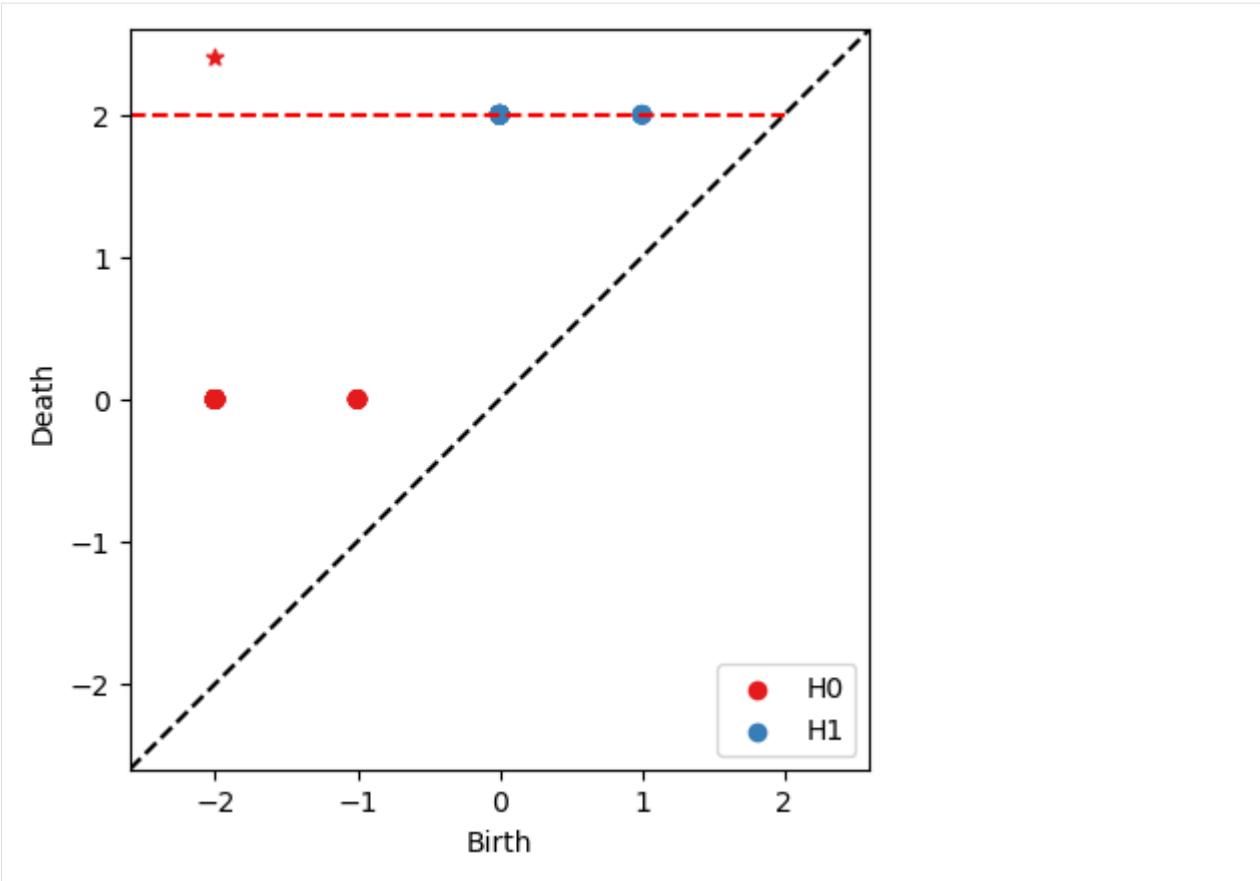
```
[16]: X = bats.Freudenthal(n,n)

# extend image filtration to Freudenthal triangulation
vals, imap = bats.lower_star_filtration(X, img.flatten())
F = bats.FilteredSimplicialComplex(X, vals)

bats.reset_field_ops(), bats.reset_column_ops()
C = bats.FilteredF2ChainComplex(F)
RC = bats.ReducedFilteredF2ChainComplex(C)
print("field operations: {}".format(bats.get_field_ops()))
print("column operations: {}".format(bats.get_column_ops()))

field operations: 1501302
column operations: 530266
```

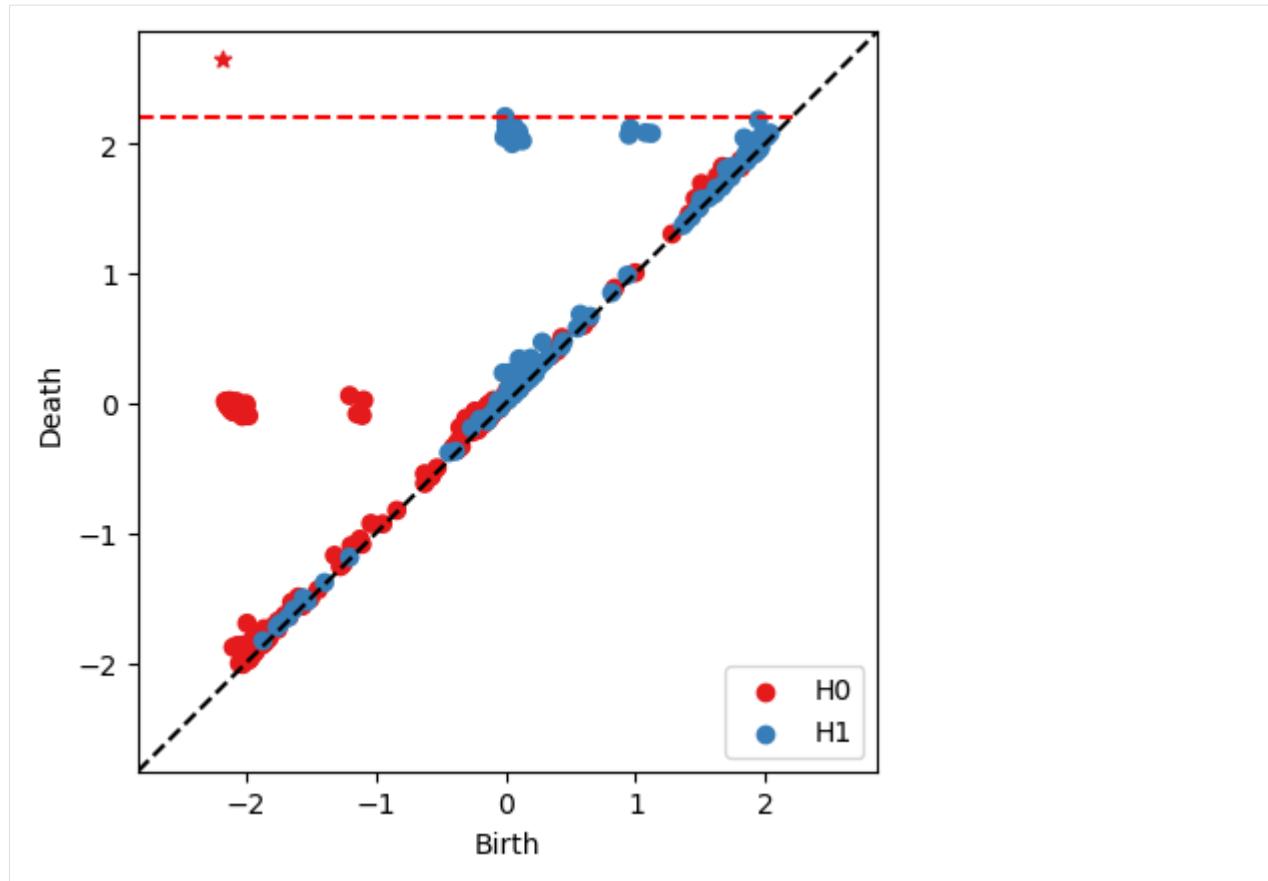
```
[17]: ps = RC.persistence_pairs(0) + RC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



Now, let's update persistence

```
[18]: vals, imap = bats.lower_star_filtration(X, img2.flatten())
bats.reset_field_ops(), bats.reset_column_ops()
RC.update_filtration(vals)
print("field operations: {}".format(bats.get_field_ops()))
print("column operations: {}".format(bats.get_column_ops()))
field operations: 325913
column operations: 40158
```

```
[19]: ps = RC.persistence_pairs(0) + RC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```

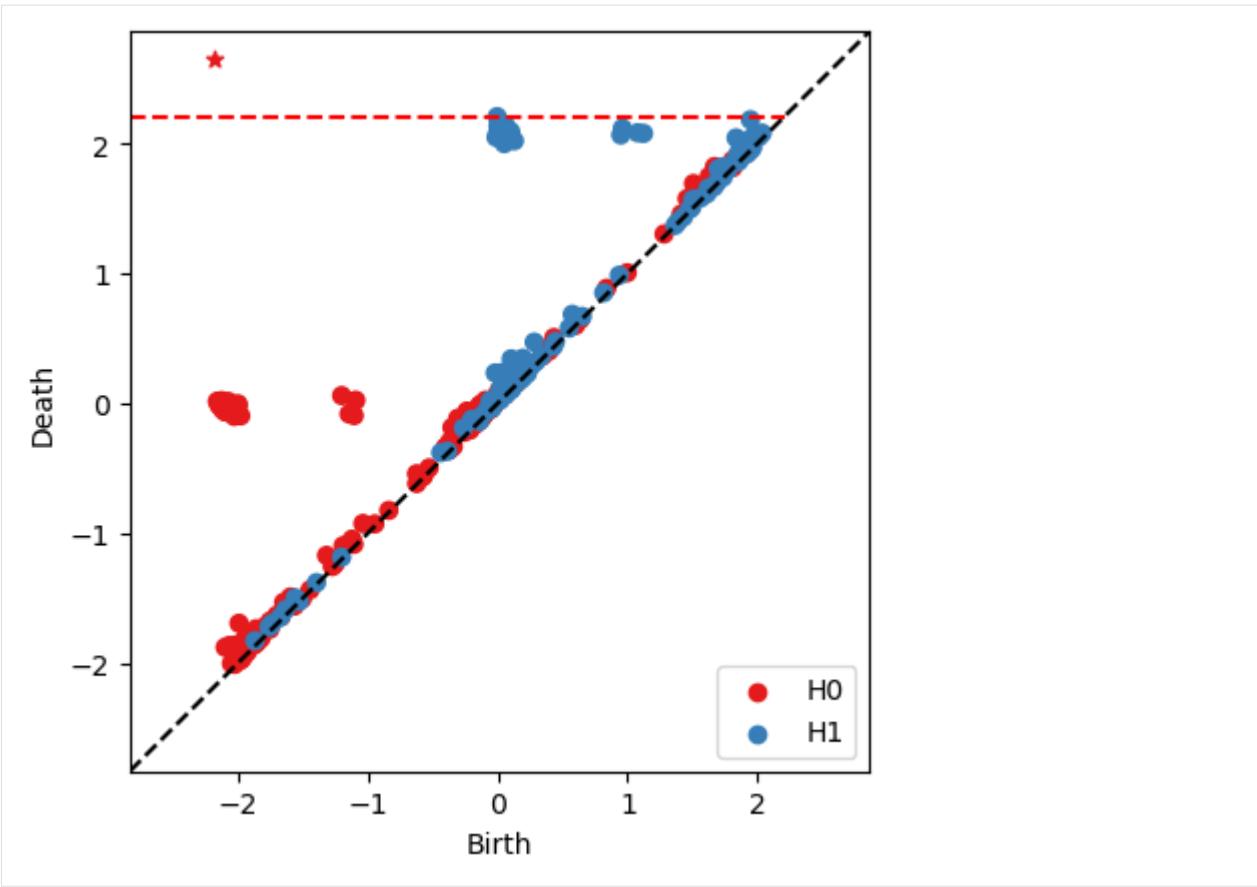


And compare to running the updated reduction from scratch

```
[20]: bats.reset_field_ops(), bats.reset_column_ops()
vals, imap = bats.lower_star_filtration(X, img2.flatten())
F = bats.FilteredSimplicialComplex(X, vals)
C = bats.FilteredF2ChainComplex(F)
RC = bats.ReducedFilteredF2ChainComplex(C)
print("field operations: {}".format(bats.get_field_ops()))
print("column operations: {}".format(bats.get_column_ops()))

field operations: 1779741
column operations: 523044
```

```
[21]: ps = RC.persistence_pairs(0) + RC.persistence_pairs(1)
bats.persistence_diagram(ps)
plt.show()
```



[]:

1.4 Examples

1.4.1 Cubical Complexes

In this example, we'll now see how to create cubical complexes using toplices, and how to compute a zigzag diagram through levelsets of an image.

```
import bats
import numpy as np
import matplotlib.pyplot as plt
```

Cubical Complexes

In BATS, cubical complexes are given a maximal dimension.

```
X = bats.CubicalComplex(3) # 3 = max dimension
```

maximum dimension cubes are defined by a list of length 2^d , where d is the dimension

```
[0,1,1,2,0,1] # cube (0,1) x (1,2) x (0,1)
```

lower dimensional cubes have degeneracies, but are still a list of length 2^d

```
[0,0,1,2,1,1] # cube (0) x (1,2) x (1)
```

Toplex

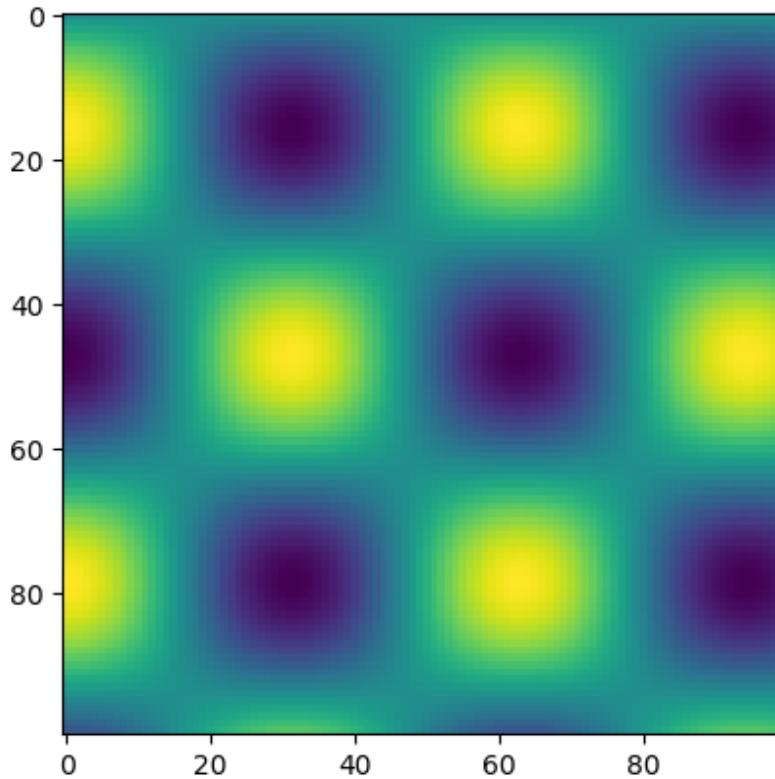
A toplex is a complex described by maximum dimension cells. All faces that must exist exist. Cubical complexes can be created by adding these top-level cells.

```
X.add_recursive([0,1,0,1,0,1]) # adds cube (0,1) x (0,1) x (0,1)
print(X.ncells()) # 27
```

Image Levelset Zigzag

Images are a common way to obtain cubical complexes. Let's generate one.

```
# generate image
m = 100
n = 100
A = np.empty((m,n))
for i in range(m):
    for j in range(n):
        A[i,j] = np.sin(i/10) * np.cos(j/10)
plt.imshow(A)
```



We'll zigzag through cubical complexes defined by level sets and their unions:

```
lsets = [[x/10, (x+2)/10] for x in range(-10,9,1)]
```

We'll operate on boolean images which indicate the support of each level set. The first thing to do is to compute top-level cubes from these images

```
def to_toplexes(A):
    """
    Create list of toplexes from boolean array A

    assume 2-dimensional for now
    """
    dims = A.shape
    toplex_list = []
    for i in range(dims[0]-1):
        for j in range(dims[1]-1):
            if (A[i,j] and A[i+1,j] and A[i,j+1] and A[i+1,j+1]):
                toplex_list.append([i,i+1,j,j+1])

    return toplex_list
```

Then we turn this list of cubes into a complex

Basic Applied Topology Subprograms

```
def image_to_complex(A):
    """
    Create cubical complex from boolean image A
    """
    toplex_list = to_toplexes(A)
    X = bats.CubicalComplex(2)
    for t in toplex_list:
        X.add_recursive(t)
    return X
```

Let's now create our diagram of complexes:

```
# create diagram of cubical complexes
D = bats.CubicalComplexDiagram()
for i in range(len(lsets)):
    lb = lsets[i][0]
    ub = lsets[i][1]
    AL = np.logical_and(lb < A, A < ub)
    ii = D.add_node(image_to_complex(AL))

    if i != 0:
        # add edge to previous union
        D.add_edge(ii, ii-1, bats.CubicalMap(D.node_data(ii), D.node_data(ii-1)))

    if i != len(lsets)-1:
        # add node for union
        ub = lsets[i+1][1]
        AL = np.logical_and(lb < A, A < ub)
        D.add_node(image_to_complex(AL))

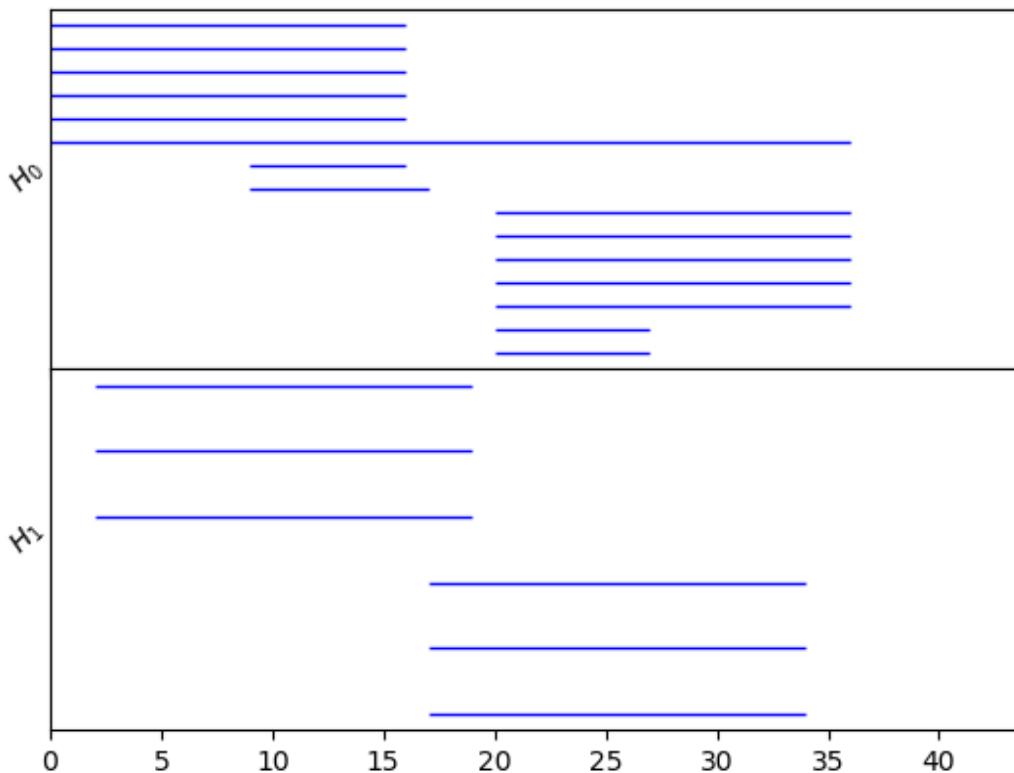
        # add edge to union
        D.add_edge(ii, ii+1, bats.CubicalMap(D.node_data(ii), D.node_data(ii+1)))
```

And we can compute the zigzag barcode:

```
FD = bats.Chain(D, bats.F2()) # F2 coefficients

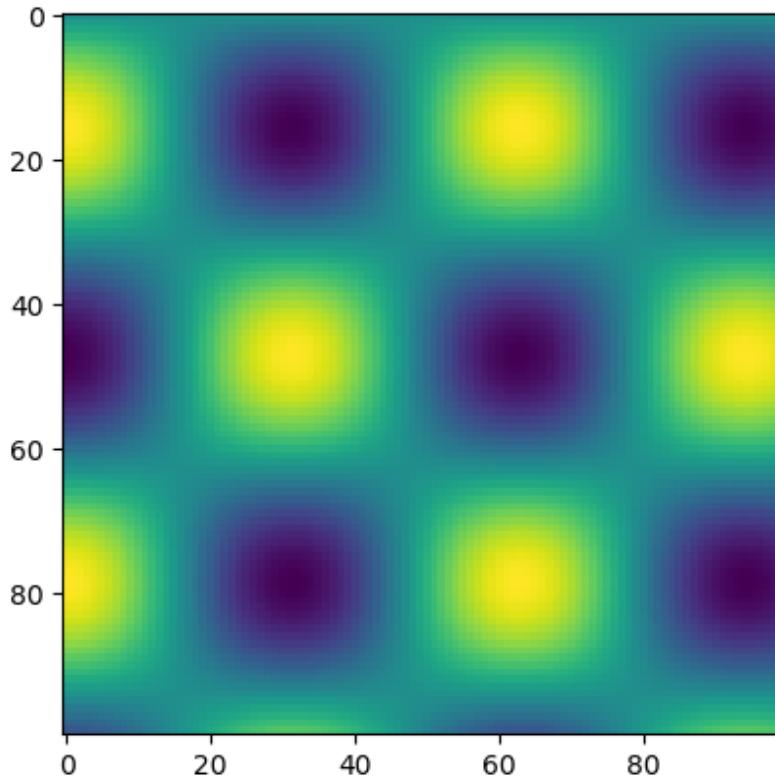
ps = []
for hdim in range(2):
    RD = bats.Hom(FD, hdim)
    ps.extend(bats.barcode(RD, hdim))

fig, ax = bats.visualization.persistence_barcode(ps)
```



Filtered Cubical Complexes

We'll now look at computing persistent homology on the image from before



The idea is to filter toplexes by the largest pixel value

```
def to_filtered_toplexes(A):
    """
    Create list of toplexes from array A
    assume 2-dimensional for now
    """
    dims = A.shape
    toplex_list = []
    for i in range(dims[0]-1):
        for j in range(dims[1]-1):
            t = max(A[i,j], A[i+1,j], A[i,j+1], A[i+1,j+1])
            toplex_list.append((t, [i,i+1,j,j+1]))

    return toplex_list
```

We can then create a filtered cubical complex

```
def image_to_filtration(A):
    """
    Create cubical complex from boolean image A
    """
    toplex_list = to_filtered_toplexes(A)
```

(continues on next page)

(continued from previous page)

```
toplex_list = sorted(toplex_list)
X = bats.FilteredCubicalComplex(2)
for t, s in toplex_list:
    X.add_recursive(t, s)
return X
```

To put everything together:

```
X = image_to_filtration(A)
C = bats.FilteredF2ChainComplex(X)
R = bats.reduce(C)
ps = R.persistence_pairs(0) + R.persistence_pairs(1)

for p in ps:
    if p.length() > 0.1:
        print(p)
```

yields the following output

```
0 : (-0.996841,inf) <0,-1>
0 : (-0.995794,0.0029413) <4,9802>
0 : (-0.99572,0.00345711) <8,9812>
0 : (-0.994674,0.00268533) <12,9796>
0 : (-0.99272,0.0014712) <16,9770>
0 : (-0.991188,0.00242617) <20,9789>
0 : (-0.365212,0.00216782) <2463,9779>
0 : (-0.364648,0.00219438) <2470,9783>
1 : (0.00363827,0.999058) <9817,9790>
1 : (0.00382229,0.9988) <9821,9784>
1 : (0.0998193,0.999432) <11671,9794>
1 : (0.930355,0.999616) <19275,9800>
```

1.4.2 Covers and Nerves

Covers

In BATS, a set is a python set, and a cover is a list of sets.

```
cover0 = [ {1,2,3}, {3,4,5}, {5,6,1} ]
```

You can also generate covers from data using landmarks

Nerves

The nerve of a cover is a simplicial complex with a vertex for every set in a cover, and a k-simplex for every non-empty intersection of k+1 sets.

```
from bats import Nerve
N = Nerve(cover0, 2) # second argument is maximum dimension of simplices
```

We can then compute homology:

```
RN = bats.reduce(N, bats.F2()) # second argument is field to use for reduction
# print the betti numbers
for d in range(RN.maxdim() + 1):
    print("betti_{}: {}".format(d, RN.hdim(d)))
```

You should see

```
betti_0: 1
betti_1: 1
betti_2: 0
```

1.4.3 Rips Filtrations

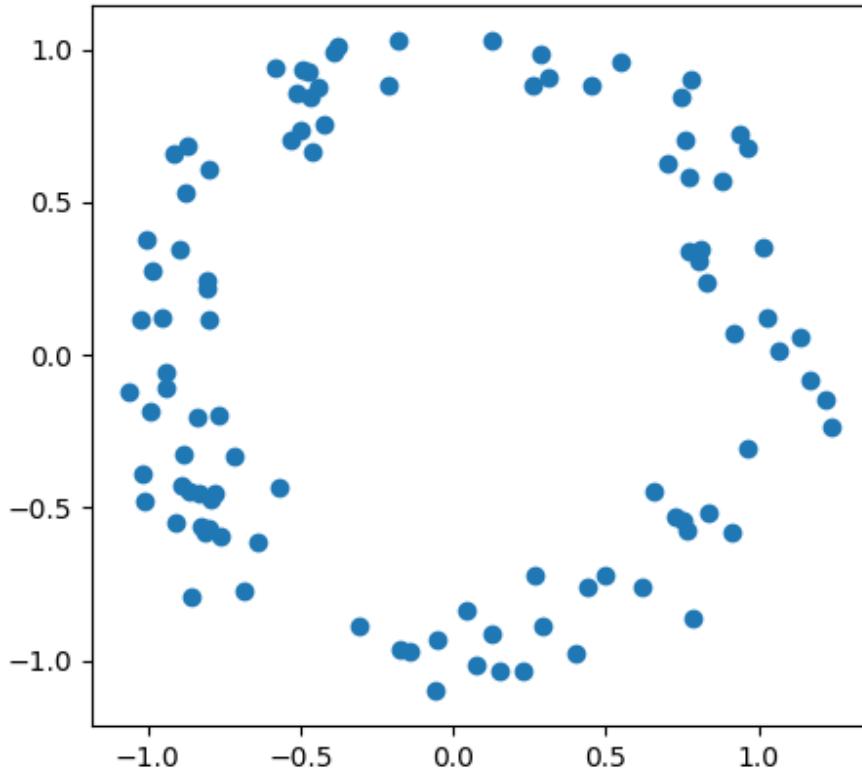
In this example, we'll cover a couple of ways to construct Rips filtrations, compute persistent homology, and subsample data.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.spatial.distance
import bats
```

1.4.4 Generate data

We'll just generate a noisy circle for demonstration purposes.

```
# first, generate a circle
n = 100
X = np.random.normal(size=(n,2))
X = X / np.linalg.norm(X, axis=1).reshape(-1,1)
X = X + np.random.normal(size=(n,2), scale = 0.1 )
fig = plt.scatter(X[:,0], X[:,1])
fig.axes.set_aspect('equal')
plt.savefig('figures/RipsEx_data.png')
```



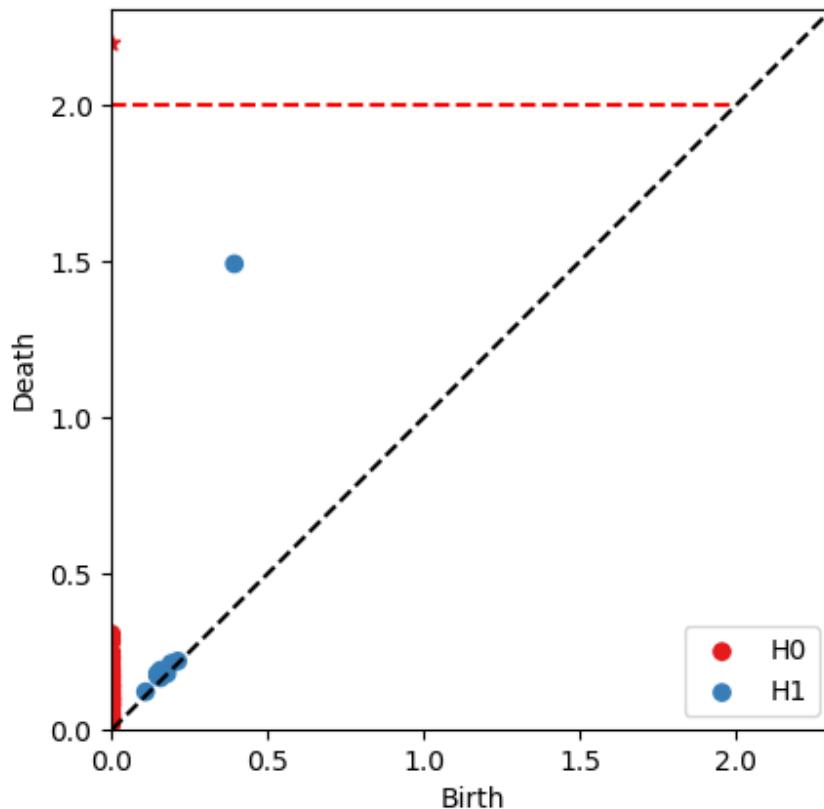
1.4.5 Use a BATS metric

You can use a distance in BATS to form a Rips filtration

```
# use bats to compute distances internally
data = bats.DataSet(bats.Matrix(X)) # put into a bats.DataSet
dist = bats.Euclidean() # distance we would like to use
F = bats.RipsFiltration(data, dist, np.inf, 2) # generate a RipsFiltration

R = bats.reduce(F, bats.F2()) # reduce with F2 coefficients
ps = []
for d in range(R.maxdim()):
    ps.extend(R.persistence_pairs(d))

fig, ax = bats.persistence_diagram(ps, tmax = 2.0)
plt.savefig('figures/RipsEx_pd_euc.png')
```



We see a robust H1 class because we sampled near a circle.

1.4.6 Pairwise distances

You can also construct Rips filtrations from pairwise distances.

you can generate a matrix of pairwise distances from a BATS metric

```
# method 1: use bats to get pairwise distances
data = bats.DataSet(bats.Matrix(X)) # put into a bats.DataSet
dist = bats.Euclidean() # distance we would like to use
pdist = dist(data, data) # returns a bats.Matrix of pairwise distances
```

or, you can generate the pairwise distances some other way

```
# method 2: use scipy to get pairwise distances
pdist_sp = scipy.spatial.distance.squareform(scipy.spatial.distance.pdist(X, 'euclidean'))
pdist = bats.Matrix(pdist_sp)
```

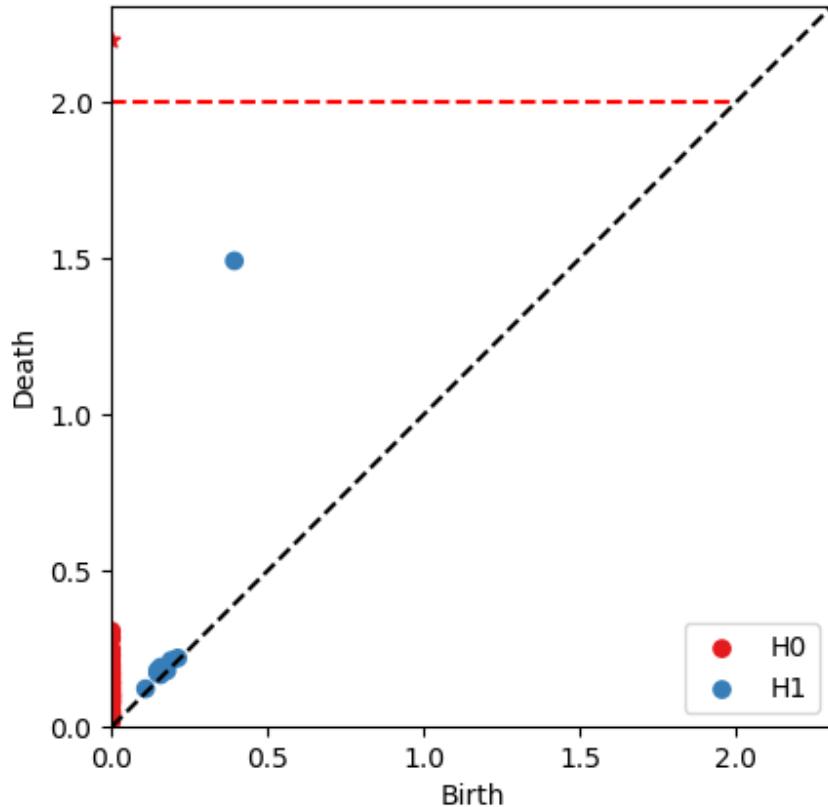
either way, you can construct a Rips filtration in a very similar way

```

F = bats.RipsFiltration(pdist, np.inf, 2) # generate a filtration on pairwise distances
R = bats.reduce(F, bats.F2()) # reduce with F2 coefficients
ps = []
for d in range(R.maxdim()):
    ps.extend(R.persistence_pairs(d))

fig, ax = bats.persistence_diagram(ps, tmax = 2.0)
plt.savefig('figures/RipsEx_pd.png')

```



1.4.7 Greedy Subsampling

BATS has a function provided to greedily subsample data. This can be used to reduce the number of points used to construct a Rips filtration, and speed up computations.

```

# inds is sequence of indices selected by greedy landmarking
# dists[k] is hausdorff distance from X[inds[:k]] to X
inds, dists = bats.greedy_landmarks_hausdorff(pdist, 0) # 0 is first index

```

We can look at the first k greedy samples, and obtain the hausdorff distance to the full data set.

```

k = 40 # we'll landmark 40 points
# print('hausdorff distance is {}'.format(dists[k]))

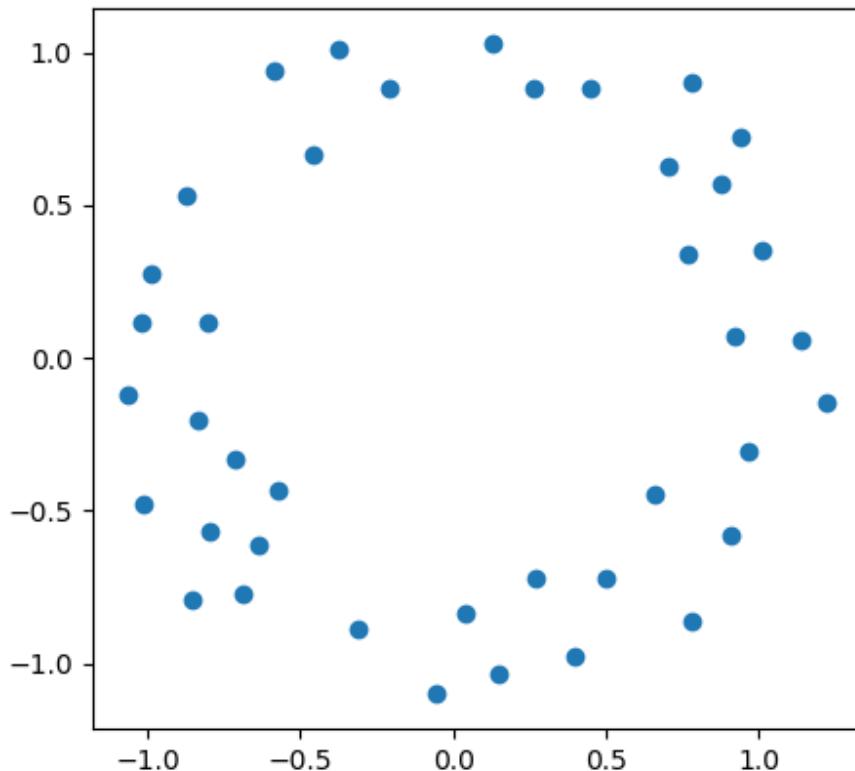
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure()
ret = plt.scatter(X[indxs[:k],0], X[indxs[:k],1])
fig.suptitle('hausdorff distance: {:.3f}'.format(dists[k]))
ret.axes.set_aspect('equal')
plt.savefig('figures/RipsEx_data_landmark.png')
```

hausdorff distance: 0.154

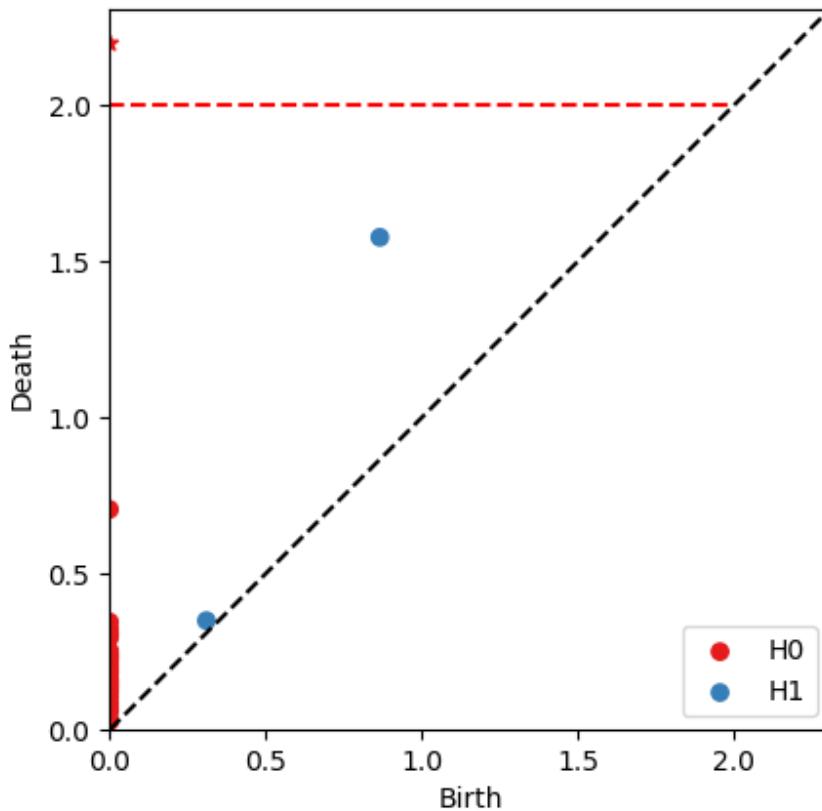


Now, we can compute persistent homology of a Rips filtration in the standard way.

```
# use scipy to get pairwise distances
pdist_sp = scipy.spatial.distance.squareform(scipy.spatial.distance.pdist(X[:k],
˓→'euclidean'))
pdist = bats.Matrix(pdist_sp)

F = bats.RipsFiltration(pdist, np.inf, 2) # generate a filtration on pairwise distances
R = bats.reduce(F, bats.F2()) # reduce with F2 coefficients
ps = []
for d in range(R.maxdim()):
    ps.extend(R.persistence_pairs(d))

fig, ax = bats.persistence_diagram(ps, tmax = 2.0)
plt.savefig('figures/RipsEx_pd_landmark.png')
```



We still see a robust H1 class, but the birth is a bit later now.

1.4.8 Visualization of H1 generator

Here's an example of how to visualize the longest-length H1 generator with plotly:

```
import plotly
import plotly.graph_objects as go

# use bats to get pairwise distances
data = bats.DataSet(bats.Matrix(X)) # put into a bats.DataSet
dist = bats.L1Dist() # distance we would like to use
pdist = dist(data, data) # returns a bats.Matrix of pairwise distances

pdist_np = np.array(pdist) # numpy array of pairwise distances

F = bats.RipsFiltration(pdist, np.inf, 2) # generate a filtration on pairwise distances
R = bats.reduce(F, bats.F2()) # reduce with F2 coefficients

# get longest H1 pair
ps1 = R.persistence_pairs(1)
lens = [p.death() - p.birth() for p in ps1] # find longest length pair
```

(continues on next page)

(continued from previous page)

```

ind = np.argmax(lens)
pair = ps1[ind]

def plot_representative_2D(X, F, R, pair, D, thresh=None, **kwargs):
    """
    Plot H1 representative on 2D scatter plot

    plot representative
    X: 2-dimensional locations of points
    F: bats FilteredSimplicialComplex
    R: bats ReducedFilteredChainComplex
    pair: bats PersistencePair
    D: N x N distance matrix
    thresh: threshold parameter
    kwargs: passed onto figure layout
    """

    if thresh is None:
        thresh = pair.birth()

    fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=X[:,0], y=X[:,1],
        mode='markers',
    ))
    edge_x = []
    edge_y = []
    N = X.shape[0]
    for i in range(N):
        for j in range(N):
            if D[i, j] <= thresh:
                edge_x.extend([X[i,0], X[j,0], None])
                edge_y.extend([X[i,1], X[j,1], None])

    fig.add_trace(go.Scatter(
        x=edge_x, y=edge_y,
        line=dict(width=0.5, color="#888"),
        hoverinfo='none',
        mode='lines')
    )

    edge_x = []
    edge_y = []
    r = R.representative(pair)
    nzind = r.nzinds()
    cpx = F.complex()
    for k in nzind:
        [i, j] = cpx.get_simplex(1, k)
        if D[i, j] <= thresh:
            edge_x.extend([X[i,0], X[j,0], None])
            edge_y.extend([X[i,1], X[j,1], None])
    fig.add_trace(go.Scatter(
        x=edge_x, y=edge_y,

```

(continues on next page)

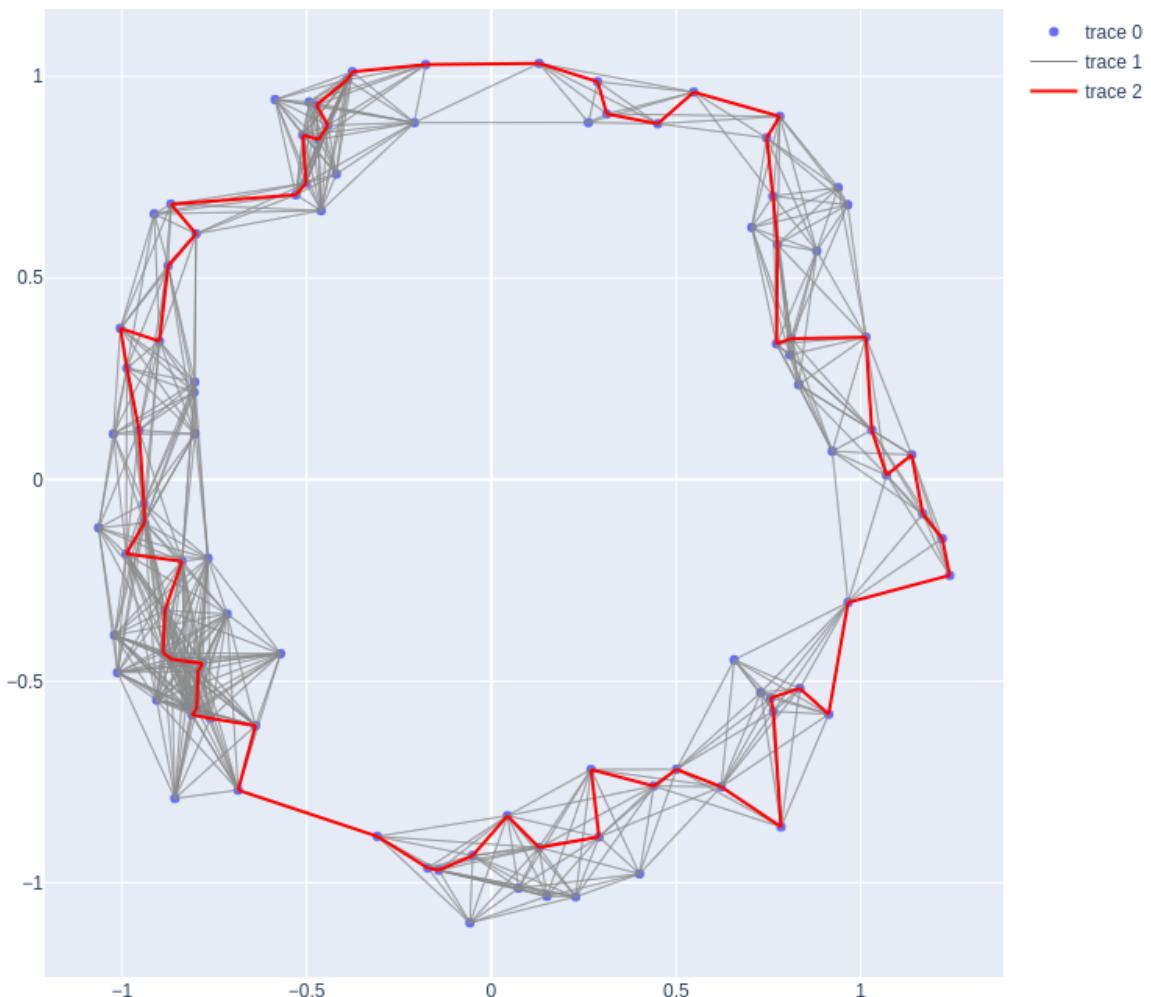
(continued from previous page)

```

        line=dict(width=2, color='red'),
        hoverinfo='none',
        mode='lines')
    )
fig.update_layout(**kwargs)
return fig

fig = plot_representative_2D(X, F, R, pair, pdist_np, width=800, height=800)
fig.write_image('figures/H1_rep.png')

```



1.5 API Reference

```
class bats.AngleDist
    Bases: pybind11_builtins.pybind11_object

class bats.CSCMatrix
    Bases: pybind11_builtins.pybind11_object
    ncol(self: bats.linalg.CSCMatrix) → int
        number of columns.
    nrow(self: bats.linalg.CSCMatrix) → int
        number of rows.
    print(self: bats.linalg.CSCMatrix) → None

class bats.CellComplex
    Bases: pybind11_builtins.pybind11_object
    add(self: bats.topology.CellComplex, arg0: List[int], arg1: List[int], arg2: int) → int
        add cell in dimension k by specifying boundary and coefficients.
    add_vertex(self: bats.topology.CellComplex) → int
        add vertex to cell complex
    add_vertices(self: bats.topology.CellComplex, arg0: int) → int
        add vertices to cell complex
    boundary(self: bats.topology.CellComplex, arg0: int) → CSCMatrix<int, unsigned long>
    maxdim(self: bats.topology.CellComplex) → int
        maximum dimension cell
    ncells(*args, **kwargs)
        Overloaded function.
            1. ncells(self: bats.topology.CellComplex) -> int
                number of cells
            2. ncells(self: bats.topology.CellComplex, arg0: int) -> int
                number of cells in given dimension

class bats.CellComplexDiagram
    Bases: pybind11_builtins.pybind11_object
    add_edge(self: bats.topology.CellComplexDiagram, arg0: int, arg1: int, arg2: bats.topology.CellularMap)
        → int
    add_node(self: bats.topology.CellComplexDiagram, arg0: bats.topology.CellComplex) → int
    edge_data(self: bats.topology.CellComplexDiagram, arg0: int) → bats.topology.CellularMap
    edge_source(self: bats.topology.CellComplexDiagram, arg0: int) → int
    edge_target(self: bats.topology.CellComplexDiagram, arg0: int) → int
    nedge(self: bats.topology.CellComplexDiagram) → int
    nnodes(self: bats.topology.CellComplexDiagram) → int
    node_data(self: bats.topology.CellComplexDiagram, arg0: int) → bats.topology.CellComplex
    set_edge(self: bats.topology.CellComplexDiagram, arg0: int, arg1: int, arg2: int, arg3:
        bats.topology.CellularMap) → None
```

```

set_node(self: bats.topology.CellComplexDiagram, arg0: int, arg1: bats.topology.CellComplex) → None
class bats.CellularMap
    Bases: pybind11_builtins.pybind11_object

bats.Chain(*args, **kwargs)
    Overloaded function.

    1. Chain(arg0: bats.topology.SimplicialComplexDiagram, arg1: bats.linalg_f2.F2) ->
       bats.linalg_f2.F2ChainDiagram
    2. Chain(arg0: bats.topology.CubicalComplexDiagram, arg1: bats.linalg_f2.F2) ->
       bats.linalg_f2.F2ChainDiagram
    3. Chain(arg0: bats.topology.CellComplexDiagram, arg1: bats.linalg_f2.F2) ->
       bats.linalg_f2.F2ChainDiagram
    4. Chain(arg0: bats.topology.CellularMap, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.F2ChainMap
    5. Chain(arg0: bats.topology.CellularMap, arg1: bats.topology.SimplicialComplex,
       arg2: bats.topology.SimplicialComplex, arg3: bats.topology.SimplicialComplex, arg4:
       bats.topology.SimplicialComplex, arg5: bats.linalg_f2.F2) -> bats.linalg_f2.F2ChainMap
    6. Chain(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f2.F2) ->
       bats.linalg_f2.F2ChainComplex
    7. Chain(arg0: bats.topology.SimplicialComplex, arg1: bats.topology.SimplicialComplex, arg2:
       bats.linalg_f2.F2) -> bats.linalg_f2.F2ChainComplex
    8. Chain(arg0: bats.topology.CubicalComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.F2ChainComplex
    9. Chain(arg0: bats.topology.CellComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.F2ChainComplex
    10. Chain(arg0: bats.topology.SimplicialComplexDiagram, arg1: bats.linalg_f3.F3) ->
       bats.linalg_f3.F3ChainDiagram
    11. Chain(arg0: bats.topology.CubicalComplexDiagram, arg1: bats.linalg_f3.F3) ->
       bats.linalg_f3.F3ChainDiagram
    12. Chain(arg0: bats.topology.CellComplexDiagram, arg1: bats.linalg_f3.F3) ->
       bats.linalg_f3.F3ChainDiagram
    13. Chain(arg0: bats.topology.CellularMap, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.F3ChainMap
    14. Chain(arg0: bats.topology.CellularMap, arg1: bats.topology.SimplicialComplex,
       arg2: bats.topology.SimplicialComplex, arg3: bats.topology.SimplicialComplex, arg4:
       bats.topology.SimplicialComplex, arg5: bats.linalg_f3.F3) -> bats.linalg_f3.F3ChainMap
    15. Chain(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3) ->
       bats.linalg_f3.F3ChainComplex
    16. Chain(arg0: bats.topology.SimplicialComplex, arg1: bats.topology.SimplicialComplex, arg2:
       bats.linalg_f3.F3) -> bats.linalg_f3.F3ChainComplex
    17. Chain(arg0: bats.topology.CubicalComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.F3ChainComplex
    18. Chain(arg0: bats.topology.CellComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.F3ChainComplex

bats.ChainFunctor(*args, **kwargs)
    Overloaded function.

    1. ChainFunctor(arg0: bats.topology.SimplicialComplexDiagram, arg1: bats.linalg_f2.F2) ->
       bats.linalg_f2.F2ChainDiagram

```

Basic Applied Topology Subprograms

```
2. ChainFunctor(arg0:      bats.topology.SimplicialComplexDiagram,    arg1:      bats.linalg_f3.F3)  ->
   bats.linalg_f3.F3ChainDiagram

class bats.CosineDist
  Bases: pybind11_builtins.pybind11_object

class bats.CoverDiagram
  Bases: pybind11_builtins.pybind11_object

  add_edge(self: bats.topology.CoverDiagram, arg0: int, arg1: int, arg2: List[int]) → int
  add_node(self: bats.topology.CoverDiagram, arg0: List[Set[int]]) → int
  edge_data(self: bats.topology.CoverDiagram, arg0: int) → List[int]
  edge_source(self: bats.topology.CoverDiagram, arg0: int) → int
  edge_target(self: bats.topology.CoverDiagram, arg0: int) → int
  nedge(self: bats.topology.CoverDiagram) → int
  nnode(self: bats.topology.CoverDiagram) → int
  node_data(self: bats.topology.CoverDiagram, arg0: int) → List[Set[int]]
  set_edge(self: bats.topology.CoverDiagram, arg0: int, arg1: int, arg2: int, arg3: List[int]) → None
  set_node(self: bats.topology.CoverDiagram, arg0: int, arg1: List[Set[int]]) → None

bats.Cube(*args, **kwargs)
  Overloaded function.

  1. Cube(arg0: int, arg1: int) -> bats.topology.CubicalComplex
  2. Cube(arg0: int, arg1: int, arg2: int) -> bats.topology.CubicalComplex
  3. Cube(arg0: int, arg1: int, arg2: int, arg3: int, arg4: int, arg5: int, arg6: int, arg7: int, arg8: int) ->
     bats.topology.CubicalComplex

class bats.CubicalComplex
  Bases: pybind11_builtins.pybind11_object

  add(self: bats.topology.CubicalComplex, arg0: List[int]) → bats.topology.cell_ind
    add cube

  add_recursive(self: bats.topology.CubicalComplex, arg0: List[int]) → List[bats.topology.cell_ind]
    add cube as well as faces

  boundary(self: bats.topology.CubicalComplex, arg0: int) → CSCMatrix<int, unsigned long>
    integer boundary matrix

  find_idx(self: bats.topology.CubicalComplex, arg0: List[int]) → int

  get_cube(self: bats.topology.CubicalComplex, arg0: int, arg1: int) → List[int]
    get cube in given dimension

  get_cubes(*args, **kwargs)
    Overloaded function.

    1. get_cubes(self: bats.topology.CubicalComplex, arg0: int) -> List[List[int]]
       Returns a list of all cubes in given dimension.

    2. get_cubes(self: bats.topology.CubicalComplex) -> List[List[int]]
       Returns a list of all cubes.
```

```

load_cubes(self: bats.topology.CubicalComplex, arg0: str) → None
    load cubes from a csv file.

maxdim(self: bats.topology.CubicalComplex) → int
    maximum dimension cube

ncells(*args, **kwargs)
    Overloaded function.
        1. ncells(self: bats.topology.CubicalComplex) -> int
            number of cells
        2. ncells(self: bats.topology.CubicalComplex, arg0: int) -> int
            number of cells in given dimension

skeleton(self: bats.topology.CubicalComplex, arg0: int) → bats.topology.CubicalComplex
    k-skeleton of complex

class bats.CubicalComplexDiagram
    Bases: pybind11_builtins.pybind11_object
        add_edge(self: bats.topology.CubicalComplexDiagram, arg0: int, arg1: int, arg2: bats.topology.CellularMap) → int
        add_node(self: bats.topology.CubicalComplexDiagram, arg0: bats.topology.CubicalComplex) → int
        edge_data(self: bats.topology.CubicalComplexDiagram, arg0: int) → bats.topology.CellularMap
        edge_source(self: bats.topology.CubicalComplexDiagram, arg0: int) → int
        edge_target(self: bats.topology.CubicalComplexDiagram, arg0: int) → int
        nedge(self: bats.topology.CubicalComplexDiagram) → int
        nnode(self: bats.topology.CubicalComplexDiagram) → int
        node_data(self: bats.topology.CubicalComplexDiagram, arg0: int) → bats.topology.CubicalComplex
        set_edge(self: bats.topology.CubicalComplexDiagram, arg0: int, arg1: int, arg2: int, arg3: bats.topology.CellularMap) → None
        set_node(self: bats.topology.CubicalComplexDiagram, arg0: int, arg1: bats.topology.CubicalComplex) → None
bats.CubicalMap(arg0: bats.topology.CubicalComplex, arg1: bats.topology.CubicalComplex) → bats.topology.CellularMap

class bats.DataSet
    Bases: pybind11_builtins.pybind11_object
        data(self: bats.dense.DataSet) → bats.dense.Matrix
        dim(self: bats.dense.DataSet) → int
        size(self: bats.dense.DataSet) → int

bats.DiscreteMorozovZigzag(*args, **kwargs)
    Overloaded function.
        1. DiscreteMorozovZigzag(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float, arg3: int) -> Tuple[bats::Diagram<bats::SimplicialComplex, bats::CellularMap>, List[float]]
            discrete Morozov Zigzag (dM-ZZ) construction.

```

2. DiscreteMorozovZigzag(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float, arg3: int)
-> Tuple[bats::Diagram<bats::SimplicialComplex, bats::CellularMap>, List[float]]

discrete Morozov Zigzag (dM-ZZ) construction.

3. DiscreteMorozovZigzag(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float, arg3: int)
-> Tuple[bats::Diagram<bats::SimplicialComplex, bats::CellularMap>, List[float]]

discrete Morozov Zigzag (dM-ZZ) construction.

4. DiscreteMorozovZigzag(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float, arg3: int)
-> Tuple[bats::Diagram<bats::SimplicialComplex, bats::CellularMap>, List[float]]

discrete Morozov Zigzag (dM-ZZ) construction.

5. DiscreteMorozovZigzag(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float, arg3: int)
-> Tuple[bats::Diagram<bats::SimplicialComplex, bats::CellularMap>, List[float]]

discrete Morozov Zigzag (dM-ZZ) construction.

6. DiscreteMorozovZigzag(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float, arg3: int)
-> Tuple[bats::Diagram<bats::SimplicialComplex, bats::CellularMap>, List[float]]

discrete Morozov Zigzag (dM-ZZ) construction.

7. DiscreteMorozovZigzag(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float, arg3: int)
-> Tuple[bats::Diagram<bats::SimplicialComplex, bats::CellularMap>, List[float]]

discrete Morozov Zigzag (dM-ZZ) construction.

bats.DiscreteMorozovZigzagSets(*args, **kwargs)

Overloaded function.

1. DiscreteMorozovZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float)
-> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long> >, std::vector<unsigned long, std::allocator<unsigned long> > >, List[float]]

SetDiagram for discrete Morozov Zigzag (dM-ZZ) construction.

2. DiscreteMorozovZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float)
-> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long> >, std::vector<unsigned long, std::allocator<unsigned long> > >, List[float]]

SetDiagram for discrete Morozov Zigzag (dM-ZZ) construction.

3. DiscreteMorozovZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float)
-> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long> >, std::vector<unsigned long, std::allocator<unsigned long> > >, List[float]]

SetDiagram for discrete Morozov Zigzag (dM-ZZ) construction.

4. DiscreteMorozovZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float)
-> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long> >, std::vector<unsigned long, std::allocator<unsigned long> > >, List[float]]

SetDiagram for discrete Morozov Zigzag (dM-ZZ) construction.

5. DiscreteMorozovZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float)
-> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long> >, std::vector<unsigned long, std::allocator<unsigned long> > >, List[float]]

SetDiagram for discrete Morozov Zigzag (dM-ZZ) construction.

6. DiscreteMorozovZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]]

SetDiagram for discrete Morozov Zigzag (dM-ZZ) construction.

7. DiscreteMorozovZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]]

SetDiagram for discrete Morozov Zigzag (dM-ZZ) construction.

bats.DowkerCoverFiltration(*args, **kwargs)

Overloaded function.

1. DowkerCoverFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.Euclidean, arg3: List[Set[int]], arg4: float, arg5: int) -> bats.topology.FilteredSimplicialComplex
2. DowkerCoverFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.L1Dist, arg3: List[Set[int]], arg4: float, arg5: int) -> bats.topology.FilteredSimplicialComplex
3. DowkerCoverFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.LInfDist, arg3: List[Set[int]], arg4: float, arg5: int) -> bats.topology.FilteredSimplicialComplex
4. DowkerCoverFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.CosineDist, arg3: List[Set[int]], arg4: float, arg5: int) -> bats.topology.FilteredSimplicialComplex
5. DowkerCoverFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.RPCosineDist, arg3: List[Set[int]], arg4: float, arg5: int) -> bats.topology.FilteredSimplicialComplex
6. DowkerCoverFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.AngleDist, arg3: List[Set[int]], arg4: float, arg5: int) -> bats.topology.FilteredSimplicialComplex
7. DowkerCoverFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.RPAngleDist, arg3: List[Set[int]], arg4: float, arg5: int) -> bats.topology.FilteredSimplicialComplex
8. DowkerCoverFiltration(arg0: A<Dense<double, RowMaj>>, arg1: List[Set[int]], arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex

bats.DowkerFiltration(*args, **kwargs)

Overloaded function.

1. DowkerFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.Euclidean, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
2. DowkerFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.L1Dist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
3. DowkerFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.LInfDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
4. DowkerFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.CosineDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
5. DowkerFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.RPCosineDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Basic Applied Topology Subprograms

6. DowkerFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.AngleDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
7. DowkerFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.RPAngleDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
8. DowkerFiltration(arg0: A<Dense<double, RowMaj>>, arg1: float, arg2: int) -> bats.topology.FilteredSimplicialComplex

bats.**EL_L_commute**(*args, **kwargs)

Overloaded function.

1. EL_L_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>, arg1: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>) -> ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>

E_L, L commutation

2. EL_L_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long>>, arg1: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long>>) -> ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long>>

E_L, L commutation

bats.**EU_U_commute**(*args, **kwargs)

Overloaded function.

1. EU_U_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>, arg1: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>) -> ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>

E_U, U commutation

2. EU_U_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long>>, arg1: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long>>) -> ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long>>

E_U, U commutation

class bats.Euclidean

Bases: pybind11_builtins.pybind11_object

class bats.F2

Bases: pybind11_builtins.pybind11_object

to_int(self: bats.linalg_f2.F2) → int
convert to integer.

bats.**F2Chain**(arg0: bats.topology.SimplicialComplexDiagram) → bats.linalg_f2.F2ChainDiagram

class bats.F2ChainComplex

Bases: pybind11_builtins.pybind11_object

clear_compress_apparent_pairs(self: bats.linalg_f2.F2ChainComplex) → None

dim(*args, **kwargs)
Overloaded function.

1. dim(self: bats.linalg_f2.F2ChainComplex, arg0: int) -> int
2. dim(self: bats.linalg_f2.F2ChainComplex) -> int

maxdim(self: bats.linalg_f2.F2ChainComplex) → int

```
class bats.F2ChainDiagram
    Bases: pybind11_builtins.pybind11_object
        add_edge(self: bats.linalg_f2.F2ChainDiagram, arg0: int, arg1: int, arg2: bats.linalg_f2.F2ChainMap) →
            int
        add_node(self: bats.linalg_f2.F2ChainDiagram, arg0: bats.linalg_f2.F2ChainComplex) → int
        edge_data(self: bats.linalg_f2.F2ChainDiagram, arg0: int) → bats.linalg_f2.F2ChainMap
        edge_source(self: bats.linalg_f2.F2ChainDiagram, arg0: int) → int
        edge_target(self: bats.linalg_f2.F2ChainDiagram, arg0: int) → int
        nedge(self: bats.linalg_f2.F2ChainDiagram) → int
        nnode(self: bats.linalg_f2.F2ChainDiagram) → int
        node_data(self: bats.linalg_f2.F2ChainDiagram, arg0: int) → bats.linalg_f2.F2ChainComplex
        set_edge(self: bats.linalg_f2.F2ChainDiagram, arg0: int, arg1: int, arg2: int, arg3:
            bats.linalg_f2.F2ChainMap) → None
        set_node(self: bats.linalg_f2.F2ChainDiagram, arg0: int, arg1: bats.linalg_f2.F2ChainComplex) → None
class bats.F2ChainMap
    Bases: pybind11_builtins.pybind11_object
class bats.F2DGHomDiagram
    Bases: pybind11_builtins.pybind11_object
        add_edge(self: bats.linalg_f2.F2DGHomDiagram, arg0: int, arg1: int, arg2: bats.linalg_f2.F2Mat) → int
        add_node(self: bats.linalg_f2.F2DGHomDiagram, arg0: bats.linalg_f2.ReducedF2DGVectorSpace) → int
        edge_data(self: bats.linalg_f2.F2DGHomDiagram, arg0: int) → bats.linalg_f2.F2Mat
        edge_source(self: bats.linalg_f2.F2DGHomDiagram, arg0: int) → int
        edge_target(self: bats.linalg_f2.F2DGHomDiagram, arg0: int) → int
        nedge(self: bats.linalg_f2.F2DGHomDiagram) → int
        nnode(self: bats.linalg_f2.F2DGHomDiagram) → int
        node_data(self: bats.linalg_f2.F2DGHomDiagram, arg0: int) → bats.linalg_f2.ReducedF2DGVectorSpace
        set_edge(self: bats.linalg_f2.F2DGHomDiagram, arg0: int, arg1: int, arg2: int, arg3:
            bats.linalg_f2.F2Mat) → None
        set_node(self: bats.linalg_f2.F2DGHomDiagram, arg0: int, arg1:
            bats.linalg_f2.ReducedF2DGVectorSpace) → None
class bats.F2DGHomDiagramAll
    Bases: pybind11_builtins.pybind11_object
        add_edge(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: int, arg1: int, arg2: List[bats.linalg_f2.F2Mat]) → int
        add_node(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: bats.linalg_f2.ReducedF2DGVectorSpace) → int
        edge_data(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: int) → List[bats.linalg_f2.F2Mat]
        edge_source(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: int) → int
        edge_target(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: int) → int
```

```
nedge(self: bats.linalg_f2.F2DGHomDiagramAll) → int
nnode(self: bats.linalg_f2.F2DGHomDiagramAll) → int
node_data(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: int) →
    bats.linalg_f2.ReducedF2DGVectorSpace
set_edge(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: int, arg1: int, arg2: int, arg3:
    List[bats.linalg_f2.F2Mat]) → None
set_node(self: bats.linalg_f2.F2DGHomDiagramAll, arg0: int, arg1:
    bats.linalg_f2.ReducedF2DGVectorSpace) → None

class bats.F2DGLinearDiagram
    Bases: pybind11_builtins.pybind11_object
        add_edge(self: bats.linalg_f2.F2DGLinearDiagram, arg0: int, arg1: int, arg2:
            bats.linalg_f2.F2DGLinearMap) → int
        add_node(self: bats.linalg_f2.F2DGLinearDiagram, arg0: bats.linalg_f2.F2DGVectorSpace) → int
        edge_data(self: bats.linalg_f2.F2DGLinearDiagram, arg0: int) → bats.linalg_f2.F2DGLinearMap
        edge_source(self: bats.linalg_f2.F2DGLinearDiagram, arg0: int) → int
        edge_target(self: bats.linalg_f2.F2DGLinearDiagram, arg0: int) → int
        nedge(self: bats.linalg_f2.F2DGLinearDiagram) → int
        nnode(self: bats.linalg_f2.F2DGLinearDiagram) → int
        node_data(self: bats.linalg_f2.F2DGLinearDiagram, arg0: int) → bats.linalg_f2.F2DGVectorSpace
        set_edge(self: bats.linalg_f2.F2DGLinearDiagram, arg0: int, arg1: int, arg2: int, arg3:
            bats.linalg_f2.F2DGLinearMap) → None
        set_node(self: bats.linalg_f2.F2DGLinearDiagram, arg0: int, arg1: bats.linalg_f2.F2DGVectorSpace) →
            None

bats.F2DGLinearFunctor(arg0: bats.topology.SimplicialComplexDiagram, arg1: int) →
    bats.linalg_f2.F2DGLinearDiagram

class bats.F2DGLinearMap
    Bases: pybind11_builtins.pybind11_object

class bats.F2DGVectorSpace
    Bases: pybind11_builtins.pybind11_object
        property degree
            maxdim(self: bats.linalg_f2.F2DGVectorSpace) → int

class bats.F2Diagram
    Bases: pybind11_builtins.pybind11_object
        add_edge(self: bats.linalg_f2.F2Diagram, arg0: int, arg1: int, arg2: bats.linalg_f2.F2Mat) → int
        add_node(self: bats.linalg_f2.F2Diagram, arg0: int) → int
        edge_data(self: bats.linalg_f2.F2Diagram, arg0: int) → bats.linalg_f2.F2Mat
        edge_source(self: bats.linalg_f2.F2Diagram, arg0: int) → int
        edge_target(self: bats.linalg_f2.F2Diagram, arg0: int) → int
        nedge(self: bats.linalg_f2.F2Diagram) → int
        nnode(self: bats.linalg_f2.F2Diagram) → int
```

```
node_data(self: bats.linalg_f2.F2Diagram, arg0: int) → int
set_edge(self: bats.linalg_f2.F2Diagram, arg0: int, arg1: int, arg2: int, arg3: bats.linalg_f2.F2Mat) →
    None
set_node(self: bats.linalg_f2.F2Diagram, arg0: int, arg1: int) → None

class bats.F2HomDiagram
Bases: pybind11_builtins.pybind11_object

add_edge(self: bats.linalg_f2.F2HomDiagram, arg0: int, arg1: int, arg2: bats.linalg_f2.F2Mat) → int
add_node(self: bats.linalg_f2.F2HomDiagram, arg0: bats.linalg_f2.ReducedF2ChainComplex) → int
edge_data(self: bats.linalg_f2.F2HomDiagram, arg0: int) → bats.linalg_f2.F2Mat
edge_source(self: bats.linalg_f2.F2HomDiagram, arg0: int) → int
edge_target(self: bats.linalg_f2.F2HomDiagram, arg0: int) → int
nedge(self: bats.linalg_f2.F2HomDiagram) → int
nnode(self: bats.linalg_f2.F2HomDiagram) → int
node_data(self: bats.linalg_f2.F2HomDiagram, arg0: int) → bats.linalg_f2.ReducedF2ChainComplex
set_edge(self: bats.linalg_f2.F2HomDiagram, arg0: int, arg1: int, arg2: int, arg3: bats.linalg_f2.F2Mat) →
    None
set_node(self: bats.linalg_f2.F2HomDiagram, arg0: int, arg1: bats.linalg_f2.ReducedF2ChainComplex) →
    None

class bats.F2HomDiagramAll
Bases: pybind11_builtins.pybind11_object

add_edge(self: bats.linalg_f2.F2HomDiagramAll, arg0: int, arg1: int, arg2: List[bats.linalg_f2.F2Mat]) →
    int
add_node(self: bats.linalg_f2.F2HomDiagramAll, arg0: bats.linalg_f2.ReducedF2ChainComplex) → int
edge_data(self: bats.linalg_f2.F2HomDiagramAll, arg0: int) → List[bats.linalg_f2.F2Mat]
edge_source(self: bats.linalg_f2.F2HomDiagramAll, arg0: int) → int
edge_target(self: bats.linalg_f2.F2HomDiagramAll, arg0: int) → int
nedge(self: bats.linalg_f2.F2HomDiagramAll) → int
nnode(self: bats.linalg_f2.F2HomDiagramAll) → int
node_data(self: bats.linalg_f2.F2HomDiagramAll, arg0: int) → bats.linalg_f2.ReducedF2ChainComplex
set_edge(self: bats.linalg_f2.F2HomDiagramAll, arg0: int, arg1: int, arg2: int, arg3:
    List[bats.linalg_f2.F2Mat]) → None
set_node(self: bats.linalg_f2.F2HomDiagramAll, arg0: int, arg1: bats.linalg_f2.ReducedF2ChainComplex)
    → None

class bats.F2Mat
Bases: pybind11_builtins.pybind11_object

T(self: bats.linalg_f2.F2Mat) → bats.linalg_f2.F2Mat
    transpose
append_column(self: bats.linalg_f2.F2Mat, arg0: bats.linalg_f2.F2Vector) → None
    appends column
```

Basic Applied Topology Subprograms

```
ncol(self: bats.linalg_f2.F2Mat) → int
    number of columns.

nnz(self: bats.linalg_f2.F2Mat) → int
    number of non-zeros

nrow(self: bats.linalg_f2.F2Mat) → int
    number of rows.

permute_cols(self: bats.linalg_f2.F2Mat, arg0: List[int]) → None
    permute columns

permute_rows(self: bats.linalg_f2.F2Mat, arg0: List[int]) → None
    permute rows

tolist(self: bats.linalg_f2.F2Mat) → List[List[bats.linalg_f2.F2]]
    return as C-style array

class bats.F2Vector
    Bases: pybind11_builtins.pybind11_object

    axpy(self: bats.linalg_f2.F2Vector, arg0: bats.linalg_f2.F2Vector) → None
        add vector y

    nnz(self: bats.linalg_f2.F2Vector) → int
        number of non-zeros

    nzinds(self: bats.linalg_f2.F2Vector) → List[int]

    nzs(self: bats.linalg_f2.F2Vector) → Tuple[List[int], List[bats.linalg_f2.F2]]
        tuple of lists: non-zero indices, and non-zero values

    nzvals(self: bats.linalg_f2.F2Vector) → List[bats.linalg_f2.F2]

    permute(self: bats.linalg_f2.F2Vector, arg0: List[int]) → None
        permute the indices

    sort(self: bats.linalg_f2.F2Vector) → None
        put non-zeros in sorted order

class bats.F3
    Bases: pybind11_builtins.pybind11_object

    to_int(self: bats.linalg_f3.F3) → int
        convert to integer.

bats.F3Chain(arg0: bats.topology.SimplicialComplexDiagram) → bats.linalg_f3.F3ChainDiagram

class bats.F3ChainComplex
    Bases: pybind11_builtins.pybind11_object

    clear_compress_apparent_pairs(self: bats.linalg_f3.F3ChainComplex) → None

    dim(*args, **kwargs)
        Overloaded function.

        1. dim(self: bats.linalg_f3.F3ChainComplex, arg0: int) -> int
        2. dim(self: bats.linalg_f3.F3ChainComplex) -> int

    maxdim(self: bats.linalg_f3.F3ChainComplex) → int

class bats.F3ChainDiagram
    Bases: pybind11_builtins.pybind11_object
```

```

add_edge(self: bats.linalg_f3.F3ChainDiagram, arg0: int, arg1: int, arg2: bats.linalg_f3.F3ChainMap) →
    int

add_node(self: bats.linalg_f3.F3ChainDiagram, arg0: bats.linalg_f3.F3ChainComplex) → int

edge_data(self: bats.linalg_f3.F3ChainDiagram, arg0: int) → bats.linalg_f3.F3ChainMap

edge_source(self: bats.linalg_f3.F3ChainDiagram, arg0: int) → int

edge_target(self: bats.linalg_f3.F3ChainDiagram, arg0: int) → int

nedge(self: bats.linalg_f3.F3ChainDiagram) → int

nnode(self: bats.linalg_f3.F3ChainDiagram) → int

node_data(self: bats.linalg_f3.F3ChainDiagram, arg0: int) → bats.linalg_f3.F3ChainComplex

set_edge(self: bats.linalg_f3.F3ChainDiagram, arg0: int, arg1: int, arg2: int, arg3:
    bats.linalg_f3.F3ChainMap) → None

set_node(self: bats.linalg_f3.F3ChainDiagram, arg0: int, arg1: bats.linalg_f3.F3ChainComplex) → None

class bats.F3ChainMap
    Bases: pybind11_builtins.pybind11_object

class bats.F3DGHomDiagram
    Bases: pybind11_builtins.pybind11_object

add_edge(self: bats.linalg_f3.F3DGHomDiagram, arg0: int, arg1: int, arg2: bats.linalg_f3.F3Mat) → int

add_node(self: bats.linalg_f3.F3DGHomDiagram, arg0: bats.linalg_f3.ReducedF3DGVectorSpace) → int

edge_data(self: bats.linalg_f3.F3DGHomDiagram, arg0: int) → bats.linalg_f3.F3Mat

edge_source(self: bats.linalg_f3.F3DGHomDiagram, arg0: int) → int

edge_target(self: bats.linalg_f3.F3DGHomDiagram, arg0: int) → int

nedge(self: bats.linalg_f3.F3DGHomDiagram) → int

nnode(self: bats.linalg_f3.F3DGHomDiagram) → int

node_data(self: bats.linalg_f3.F3DGHomDiagram, arg0: int) → bats.linalg_f3.ReducedF3DGVectorSpace

set_edge(self: bats.linalg_f3.F3DGHomDiagram, arg0: int, arg1: int, arg2: int, arg3:
    bats.linalg_f3.F3Mat) → None

set_node(self: bats.linalg_f3.F3DGHomDiagram, arg0: int, arg1:
    bats.linalg_f3.ReducedF3DGVectorSpace) → None

class bats.F3DGHomDiagramAll
    Bases: pybind11_builtins.pybind11_object

add_edge(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: int, arg1: int, arg2: List[bats.linalg_f3.F3Mat]) →
    int

add_node(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: bats.linalg_f3.ReducedF3DGVectorSpace) →
    int

edge_data(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: int) → List[bats.linalg_f3.F3Mat]

edge_source(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: int) → int

edge_target(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: int) → int

nedge(self: bats.linalg_f3.F3DGHomDiagramAll) → int

nnode(self: bats.linalg_f3.F3DGHomDiagramAll) → int

```

```
node_data(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: int) →
    bats.linalg_f3.ReducedF3DGVectorSpace

set_edge(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: int, arg1: int, arg2: int, arg3:
    List[bats.linalg_f3.F3Mat]) → None

set_node(self: bats.linalg_f3.F3DGHomDiagramAll, arg0: int, arg1:
    bats.linalg_f3.ReducedF3DGVectorSpace) → None

class bats.F3DGLinearDiagram
    Bases: pybind11_builtins.pybind11_object

    add_edge(self: bats.linalg_f3.F3DGLinearDiagram, arg0: int, arg1: int, arg2:
        bats.linalg_f3.F3DGLinearMap) → int

    add_node(self: bats.linalg_f3.F3DGLinearDiagram, arg0: bats.linalg_f3.F3DGVectorSpace) → int
    edge_data(self: bats.linalg_f3.F3DGLinearDiagram, arg0: int) → bats.linalg_f3.F3DGLinearMap
    edge_source(self: bats.linalg_f3.F3DGLinearDiagram, arg0: int) → int
    edge_target(self: bats.linalg_f3.F3DGLinearDiagram, arg0: int) → int
    nedge(self: bats.linalg_f3.F3DGLinearDiagram) → int
    nnnode(self: bats.linalg_f3.F3DGLinearDiagram) → int
    node_data(self: bats.linalg_f3.F3DGLinearDiagram, arg0: int) → bats.linalg_f3.F3DGVectorSpace
    set_edge(self: bats.linalg_f3.F3DGLinearDiagram, arg0: int, arg1: int, arg2: int, arg3:
        bats.linalg_f3.F3DGLinearMap) → None
    set_node(self: bats.linalg_f3.F3DGLinearDiagram, arg0: int, arg1:
        bats.linalg_f3.F3DGVectorSpace) → None

bats.F3DGLinearFunctor(arg0: bats.topology.CubicalComplexDiagram, arg1: int) →
    bats.linalg_f3.F3DGLinearDiagram

class bats.F3DGLinearMap
    Bases: pybind11_builtins.pybind11_object

class bats.F3DGVectorSpace
    Bases: pybind11_builtins.pybind11_object

    property degree

    maxdim(self: bats.linalg_f3.F3DGVectorSpace) → int

class bats.F3Diagram
    Bases: pybind11_builtins.pybind11_object

    add_edge(self: bats.linalg_f3.F3Diagram, arg0: int, arg1: int, arg2: bats.linalg_f3.F3Mat) → int
    add_node(self: bats.linalg_f3.F3Diagram, arg0: int) → int
    edge_data(self: bats.linalg_f3.F3Diagram, arg0: int) → bats.linalg_f3.F3Mat
    edge_source(self: bats.linalg_f3.F3Diagram, arg0: int) → int
    edge_target(self: bats.linalg_f3.F3Diagram, arg0: int) → int
    nedge(self: bats.linalg_f3.F3Diagram) → int
    nnnode(self: bats.linalg_f3.F3Diagram) → int
    node_data(self: bats.linalg_f3.F3Diagram, arg0: int) → int
```

```

set_edge(self: bats.linalg_f3.F3Diagram, arg0: int, arg1: int, arg2: int, arg3: bats.linalg_f3.F3Mat) →
    None

set_node(self: bats.linalg_f3.F3Diagram, arg0: int, arg1: int) → None

class bats.F3HomDiagram
    Bases: pybind11_builtins.pybind11_object

        add_edge(self: bats.linalg_f3.F3HomDiagram, arg0: int, arg1: int, arg2: bats.linalg_f3.F3Mat) → int
        add_node(self: bats.linalg_f3.F3HomDiagram, arg0: bats.linalg_f3.ReducedF3ChainComplex) → int
        edge_data(self: bats.linalg_f3.F3HomDiagram, arg0: int) → bats.linalg_f3.F3Mat
        edge_source(self: bats.linalg_f3.F3HomDiagram, arg0: int) → int
        edge_target(self: bats.linalg_f3.F3HomDiagram, arg0: int) → int
        nedge(self: bats.linalg_f3.F3HomDiagram) → int
        nnode(self: bats.linalg_f3.F3HomDiagram) → int
        node_data(self: bats.linalg_f3.F3HomDiagram, arg0: int) → bats.linalg_f3.ReducedF3ChainComplex
        set_edge(self: bats.linalg_f3.F3HomDiagram, arg0: int, arg1: int, arg2: int, arg3: bats.linalg_f3.F3Mat) →
            None
        set_node(self: bats.linalg_f3.F3HomDiagram, arg0: int, arg1: bats.linalg_f3.ReducedF3ChainComplex) →
            None

class bats.F3HomDiagramAll
    Bases: pybind11_builtins.pybind11_object

        add_edge(self: bats.linalg_f3.F3HomDiagramAll, arg0: int, arg1: int, arg2: List[bats.linalg_f3.F3Mat]) →
            int
        add_node(self: bats.linalg_f3.F3HomDiagramAll, arg0: bats.linalg_f3.ReducedF3ChainComplex) → int
        edge_data(self: bats.linalg_f3.F3HomDiagramAll, arg0: int) → List[bats.linalg_f3.F3Mat]
        edge_source(self: bats.linalg_f3.F3HomDiagramAll, arg0: int) → int
        edge_target(self: bats.linalg_f3.F3HomDiagramAll, arg0: int) → int
        nedge(self: bats.linalg_f3.F3HomDiagramAll) → int
        nnode(self: bats.linalg_f3.F3HomDiagramAll) → int
        node_data(self: bats.linalg_f3.F3HomDiagramAll, arg0: int) → bats.linalg_f3.ReducedF3ChainComplex
        set_edge(self: bats.linalg_f3.F3HomDiagramAll, arg0: int, arg1: int, arg2: int, arg3:
            List[bats.linalg_f3.F3Mat]) → None
        set_node(self: bats.linalg_f3.F3HomDiagramAll, arg0: int, arg1: bats.linalg_f3.ReducedF3ChainComplex) →
            None

class bats.F3Mat
    Bases: pybind11_builtins.pybind11_object

        T(self: bats.linalg_f3.F3Mat) → bats.linalg_f3.F3Mat
            transpose

        append_column(self: bats.linalg_f3.F3Mat, arg0: bats.linalg_f3.F3Vector) → None
            appends column

        ncol(self: bats.linalg_f3.F3Mat) → int
            number of columns.

```

Basic Applied Topology Subprograms

```
nnz(self: bats.linalg_f3.F3Mat) → int
    number of non-zeros

nrow(self: bats.linalg_f3.F3Mat) → int
    number of rows.

permute_cols(self: bats.linalg_f3.F3Mat, arg0: List[int]) → None
    permute columns

permute_rows(self: bats.linalg_f3.F3Mat, arg0: List[int]) → None
    permute rows

tolist(self: bats.linalg_f3.F3Mat) → List[List[bats.linalg_f3.F3]]
    return as C-style array

class bats.F3Vector
    Bases: pybind11_builtins.pybind11_object

    axpy(self: bats.linalg_f3.F3Vector, arg0: bats.linalg_f3.F3Vector) → None
        add vector y

    nnz(self: bats.linalg_f3.F3Vector) → int
        number of non-zeros

    nzinds(self: bats.linalg_f3.F3Vector) → List[int]

    nzs(self: bats.linalg_f3.F3Vector) → Tuple[List[int], List[bats.linalg_f3.F3]]
        tuple of lists: non-zero indices, and non-zero values

    nzvals(self: bats.linalg_f3.F3Vector) → List[bats.linalg_f3.F3]

    permute(self: bats.linalg_f3.F3Vector, arg0: List[int]) → None
        permute the indices

    sort(self: bats.linalg_f3.F3Vector) → None
        put non-zeros in sorted order

class bats.FilteredCubicalComplex
    Bases: pybind11_builtins.pybind11_object

    add(self: bats.topology.FilteredCubicalComplex, arg0: float, arg1: List[int]) → bats.topology.cell_ind

    add_recursive(self: bats.topology.FilteredCubicalComplex, arg0: float, arg1: List[int]) →
        List[bats.topology.cell_ind]

    all_vals(self: bats.topology.FilteredCubicalComplex) → List[List[float]]

    complex(self: bats.topology.FilteredCubicalComplex) → bats.topology.CubicalComplex

    maxdim(self: bats.topology.FilteredCubicalComplex) → int

    ncells(self: bats.topology.FilteredCubicalComplex, arg0: int) → int

    sublevelset(self: bats.topology.FilteredCubicalComplex, arg0: float) → bats.topology.CubicalComplex

    update_filtration(self: bats.topology.FilteredCubicalComplex, arg0: List[List[float]]) → None

    vals(self: bats.topology.FilteredCubicalComplex, arg0: int) → List[float]

class bats.FilteredEdge
    Bases: pybind11_builtins.pybind11_object

class bats.FilteredF2ChainComplex
    Bases: pybind11_builtins.pybind11_object
```

```
perm(self: bats.linalg_f2.FilteredF2ChainComplex) → List[List[int]]  
    permutation from original order  
update_filtration(self: bats.linalg_f2.FilteredF2ChainComplex, arg0: List[List[float]]) → None  
    update filtration with new values  
update_filtration_general(*args, **kwargs)  
    Overloaded function.  
        1. update_filtration_general(self: bats.linalg_f2.FilteredF2ChainComplex, arg0:  
            bats.topology.UpdateInfoFiltration) -> None  
                general update Filtered Chain Complex with updating information  
        2. update_filtration_general(self: bats.linalg_f2.FilteredF2ChainComplex, arg0:  
            bats.topology.UpdateInfoLightFiltration) -> None  
                general update Filtered Chain Complex with updating information  
val(self: bats.linalg_f2.FilteredF2ChainComplex) → List[List[float]]  
    filtration values.  
class bats.FilteredF2DGVectorSpace  
    Bases: pybind11_builtins.pybind11_object  
perm(self: bats.linalg_f2.FilteredF2DGVectorSpace) → List[List[int]]  
    permutation from original order  
update_filtration(self: bats.linalg_f2.FilteredF2DGVectorSpace, arg0: List[List[float]]) → None  
    update filtration with new values  
val(self: bats.linalg_f2.FilteredF2DGVectorSpace) → List[List[float]]  
    filtration values.  
class bats.FilteredF3ChainComplex  
    Bases: pybind11_builtins.pybind11_object  
perm(self: bats.linalg_f3.FilteredF3ChainComplex) → List[List[int]]  
    permutation from original order  
update_filtration(self: bats.linalg_f3.FilteredF3ChainComplex, arg0: List[List[float]]) → None  
    update filtration with new values  
update_filtration_general(*args, **kwargs)  
    Overloaded function.  
        1. update_filtration_general(self: bats.linalg_f3.FilteredF3ChainComplex, arg0:  
            bats.topology.UpdateInfoFiltration) -> None  
                general update Filtered Chain Complex with updating information  
        2. update_filtration_general(self: bats.linalg_f3.FilteredF3ChainComplex, arg0:  
            bats.topology.UpdateInfoLightFiltration) -> None  
                general update Filtered Chain Complex with updating information  
val(self: bats.linalg_f3.FilteredF3ChainComplex) → List[List[float]]  
    filtration values.  
class bats.FilteredF3DGVectorSpace  
    Bases: pybind11_builtins.pybind11_object  
perm(self: bats.linalg_f3.FilteredF3DGVectorSpace) → List[List[int]]  
    permutation from original order
```

Basic Applied Topology Subprograms

```
update_filtration(self: bats.linalg_f3.FilteredF3DGVectorSpace, arg0: List[List[float]]) → None
    update filtration with new values

val(self: bats.linalg_f3.FilteredF3DGVectorSpace) → List[List[float]]
    filtration values.

class bats.FilteredLightSimplicialComplex
    Bases: pybind11_builtins.pybind11_object

    add(self: bats.topology.FilteredLightSimplicialComplex, arg0: float, arg1: List[int]) →
        bats.topology.cell_ind

    add_recursive(self: bats.topology.FilteredLightSimplicialComplex, arg0: float, arg1: List[int]) →
        List[bats.topology.cell_ind]

    all_vals(self: bats.topology.FilteredLightSimplicialComplex) → List[List[float]]

    complex(self: bats.topology.FilteredLightSimplicialComplex) → bats.topology.LightSimplicialComplex

    maxdim(self: bats.topology.FilteredLightSimplicialComplex) → int

    ncells(self: bats.topology.FilteredLightSimplicialComplex, arg0: int) → int

    sublevelset(self: bats.topology.FilteredLightSimplicialComplex, arg0: float) →
        bats.topology.LightSimplicialComplex

    update_filtration(self: bats.topology.FilteredLightSimplicialComplex, arg0: List[List[float]]) → None

    vals(self: bats.topology.FilteredLightSimplicialComplex, arg0: int) → List[float]

class bats.FilteredSimplicialComplex
    Bases: pybind11_builtins.pybind11_object

    add(self: bats.topology.FilteredSimplicialComplex, arg0: float, arg1: List[int]) → bats.topology.cell_ind

    add_directed(self: bats.topology.FilteredSimplicialComplex, arg0: float, arg1: List[int]) →
        bats.topology.cell_ind

    add_recursive(self: bats.topology.FilteredSimplicialComplex, arg0: float, arg1: List[int]) →
        List[bats.topology.cell_ind]

    all_vals(self: bats.topology.FilteredSimplicialComplex) → List[List[float]]

    complex(self: bats.topology.FilteredSimplicialComplex) → bats.topology.SimplicialComplex

    maxdim(self: bats.topology.FilteredSimplicialComplex) → int

    ncells(self: bats.topology.FilteredSimplicialComplex, arg0: int) → int

    sublevelset(self: bats.topology.FilteredSimplicialComplex, arg0: float) →
        bats.topology.SimplicialComplex

    update_filtration(self: bats.topology.FilteredSimplicialComplex, arg0: List[List[float]]) → None

    vals(self: bats.topology.FilteredSimplicialComplex, arg0: int) → List[float]

bats.Filtration(*args, **kwargs)
    Overloaded function.

    1. Filtration(arg0: bats.topology.SimplicialComplex, arg1: List[List[float]]) -> bats::Filtration<double,
       bats::SimplicialComplex>

    2. Filtration(arg0: bats.topology.LightSimplicialComplex, arg1: List[List[float]]) -> bats::Filtration<double,
       bats::LightSimplicialComplex<unsigned long, std::unordered_map<unsigned long, unsigned long,
       std::hash<unsigned long>, std::equal_to<unsigned long>, std::allocator<std::pair<unsigned long const, unsigned long>>>>>
```

3. Filtration(arg0: bats.topology.CubicalComplex, arg1: List[List[float]]) -> bats::Filtration<double, bats::CubicalComplex>

bats.FlagComplex(arg0: List[int], arg1: int, arg2: int) → *bats.topology.SimplicialComplex*

Flag complex constructed from a (flattened) list of edges.

bats.Freudenthal(*args, **kwargs)

Overloaded function.

1. Freudenthal(arg0: int, arg1: int) -> bats.topology.SimplicialComplex
2. Freudenthal(arg0: int, arg1: int, arg2: int) -> bats.topology.SimplicialComplex
3. Freudenthal(arg0: int, arg1: int, arg2: int, arg3: int, arg4: int, arg5: int, arg6: int, arg7: int, arg8: int) -> bats.topology.SimplicialComplex
4. Freudenthal(arg0: bats.topology.CubicalComplex, arg1: int, arg2: int, arg3: int) -> bats.topology.SimplicialComplex

bats.Hom(*args, **kwargs)

Overloaded function.

1. Hom(arg0: bats.linalg_f2.F2ChainDiagram, arg1: int) -> bats.linalg_f2.F2HomDiagram
2. Hom(arg0: bats.linalg_f2.F2ChainDiagram) -> bats.linalg_f2.F2HomDiagramAll
3. Hom(arg0: bats.linalg_f2.F2ChainDiagram, arg1: bool) -> bats.linalg_f2.F2HomDiagramAll
4. Hom(arg0: bats.linalg_f2.F2DGLinearDiagram, arg1: int) -> bats.linalg_f2.F2DGHomDiagram
5. Hom(arg0: bats.linalg_f2.F2DGLinearDiagram) -> bats.linalg_f2.F2DGHomDiagramAll
6. Hom(arg0: bats.linalg_f2.F2DGLinearDiagram, arg1: bool) -> bats.linalg_f2.F2DGHomDiagramAll
7. Hom(arg0: bats.linalg_f3.F3ChainDiagram, arg1: int) -> bats.linalg_f3.F3HomDiagram
8. Hom(arg0: bats.linalg_f3.F3ChainDiagram) -> bats.linalg_f3.F3HomDiagramAll
9. Hom(arg0: bats.linalg_f3.F3ChainDiagram, arg1: bool) -> bats.linalg_f3.F3HomDiagramAll
10. Hom(arg0: bats.linalg_f3.F3DGLinearDiagram, arg1: int) -> bats.linalg_f3.F3DGHomDiagram
11. Hom(arg0: bats.linalg_f3.F3DGLinearDiagram) -> bats.linalg_f3.F3DGHomDiagramAll
12. Hom(arg0: bats.linalg_f3.F3DGLinearDiagram, arg1: bool) -> bats.linalg_f3.F3DGHomDiagramAll

bats.Identity(*args, **kwargs)

Overloaded function.

1. Identity(arg0: int, arg1: ModP<int, 2u>) -> ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>
2. Identity(arg0: int, arg1: ModP<int, 3u>) -> ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long>>

bats.IdentityMap(arg0: *bats.topology.SimplicialComplex*) → *bats.topology.CellularMap*

bats.InducedMap(*args, **kwargs)

Overloaded function.

1. InducedMap(arg0: bats.linalg_f2.F2ChainMap, arg1: bats.linalg_f2.ReducedF2ChainComplex, arg2: bats.linalg_f2.ReducedF2ChainComplex, arg3: int) -> bats.linalg_f2.F2Mat

Induced map on homology.

2. InducedMap(arg0: bats.linalg_f2.F2DGLinearMap, arg1: bats.linalg_f2.ReducedF2DGVectorSpace, arg2: bats.linalg_f2.ReducedF2DGVectorSpace, arg3: int) -> bats.linalg_f2.F2Mat

Basic Applied Topology Subprograms

Induced map on homology.

3. InducedMap(arg0: bats.linalg_f3.F3ChainMap, arg1: bats.linalg_f3.ReducedF3ChainComplex, arg2: bats.linalg_f3.ReducedF3ChainComplex, arg3: int) -> bats.linalg_f3.F3Mat

Induced map on homology.

4. InducedMap(arg0: bats.linalg_f3.F3DGLinearMap, arg1: bats.linalg_f3.ReducedF3DGVectorSpace, arg2: bats.linalg_f3.ReducedF3DGVectorSpace, arg3: int) -> bats.linalg_f3.F3Mat

Induced map on homology.

class bats.IntMat

Bases: pybind11_builtins.pybind11_object

T(*self*: bats.linalg.IntMat) → *bats.linalg.IntMat*

transpose

append_column(*self*: bats.linalg.IntMat, *arg0*: bats.linalg.IntVector) → None

appends column

ncol(*self*: bats.linalg.IntMat) → int

number of columns.

nnz(*self*: bats.linalg.IntMat) → int

number of non-zeros

nrow(*self*: bats.linalg.IntMat) → int

number of rows.

permute_cols(*self*: bats.linalg.IntMat, *arg0*: List[int]) → None

permute columns

permute_rows(*self*: bats.linalg.IntMat, *arg0*: List[int]) → None

permute rows

tolist(*self*: bats.linalg.IntMat) → List[List[int]]

return as C-style array

class bats.IntVector

Bases: pybind11_builtins.pybind11_object

axpy(*self*: bats.linalg.IntVector, *arg0*: bats.linalg.IntVector) → None

add vector y

nnz(*self*: bats.linalg.IntVector) → int

number of non-zeros

nzinds(*self*: bats.linalg.IntVector) → List[int]

nzs(*self*: bats.linalg.IntVector) → Tuple[List[int], List[int]]

tuple of lists: non-zero indices, and non-zero values

nzvals(*self*: bats.linalg.IntVector) → List[int]

permute(*self*: bats.linalg.IntVector, *arg0*: List[int]) → None

permute the indices

sort(*self*: bats.linalg.IntVector) → None

put non-zeros in sorted order

class bats.L1Dist

Bases: pybind11_builtins.pybind11_object

`bats.LEUP(*args, **kwargs)`

Overloaded function.

1. LEUP(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >]

LEUP factorization

2. LEUP(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >]

LEUP factorization

`class bats.LInfDist`

Bases: pybind11_builtins.pybind11_object

`bats.LQU(*args, **kwargs)`

Overloaded function.

1. LQU(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >]

LQU factorization

2. LQU(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >]

LQU factorization

`bats.L_EL_commute(*args, **kwargs)`

Overloaded function.

1. L_EL_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >

L, E_L commutation

2. L_EL_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >

L, E_L commutation

`bats.LightFlagComplex(arg0: List[int], arg1: int, arg2: int) → bats.topology.LightSimplicialComplex`

Flag complex constructed from a (flattened) list of edges.

`bats.LightFreudenthal(*args, **kwargs)`

Overloaded function.

1. LightFreudenthal(arg0: int, arg1: int) -> bats.topology.LightSimplicialComplex
2. LightFreudenthal(arg0: int, arg1: int, arg2: int) -> bats.topology.LightSimplicialComplex
3. LightFreudenthal(arg0: int, arg1: int, arg2: int, arg3: int, arg4: int, arg5: int, arg6: int, arg7: int, arg8: int) -> bats.topology.LightSimplicialComplex

Basic Applied Topology Subprograms

4. LightFreudenthal(arg0: bats.topology.CubicalComplex, arg1: int, arg2: int, arg3: int) -> bats.topology.LightSimplicialComplex

bats.**LightRipsComplex**(arg0: A<Dense<double, RowMaj>>, arg1: float, arg2: int) →
bats.topology.LightSimplicialComplex

Rips Complex constructed from pairwise distances.

bats.**LightRipsFiltration**(*args, **kwargs)

Overloaded function.

1. LightRipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float, arg3: int) -> bats.topology.FilteredLightSimplicialComplex
2. LightRipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float, arg3: int) -> bats.topology.FilteredLightSimplicialComplex
3. LightRipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float, arg3: int) -> bats.topology.FilteredLightSimplicialComplex
4. LightRipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float, arg3: int) -> bats.topology.FilteredLightSimplicialComplex
5. LightRipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float, arg3: int) -> bats.topology.FilteredLightSimplicialComplex
6. LightRipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float, arg3: int) -> bats.topology.FilteredLightSimplicialComplex
7. LightRipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float, arg3: int) -> bats.topology.FilteredLightSimplicialComplex
8. LightRipsFiltration(arg0: A<Dense<double, RowMaj>>, arg1: float, arg2: int) -> bats.topology.FilteredLightSimplicialComplex

Rips Filtration using built using pairwise distances.

bats.**LightRipsFiltration_extension**(*args, **kwargs)

Overloaded function.

1. LightRipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

2. LightRipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

3. LightRipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

4. LightRipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

5. LightRipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

6. LightRipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

7. LightRipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

8. LightRipsFiltration_extension(arg0: A<Dense<double, RowMaj> >, arg1: float, arg2: int) -> Tuple[bats.topology.FilteredLightSimplicialComplex, List[List[int]]]

Rips Filtration built using pairwise distances with inverse map returned.

```
class bats.LightSimplicialComplex
    Bases: pybind11_builtins.pybind11_object

    add(self: bats.topology.LightSimplicialComplex, arg0: List[int]) → bats.topology.cell_ind
        add simplex

    add_recursive(self: bats.topology.LightSimplicialComplex, arg0: List[int]) → List[bats.topology.cell_ind]
        add simplex and missing faces

    boundary(self: bats.topology.LightSimplicialComplex, arg0: int) → CSCMatrix<int, unsigned long>
    find_idx(self: bats.topology.LightSimplicialComplex, arg0: List[int]) → int
    get_simplex(self: bats.topology.LightSimplicialComplex, arg0: int, arg1: int) → List[int]
    get_simplices(*args, **kwargs)
        Overloaded function.
        1. get_simplices(self: bats.topology.LightSimplicialComplex, arg0: int) -> List[List[int]]
            Returns a list of all simplices in given dimension.
        2. get_simplices(self: bats.topology.LightSimplicialComplex) -> List[List[int]]
            Returns a list of all simplices.

    maxdim(self: bats.topology.LightSimplicialComplex) → int
        maximum dimension simplex

    ncells(*args, **kwargs)
        Overloaded function.
        1. ncells(self: bats.topology.LightSimplicialComplex) -> int
            number of cells
        2. ncells(self: bats.topology.LightSimplicialComplex, arg0: int) -> int
            number of cells in given dimension

bats.Mat(*args, **kwargs)
    Overloaded function.
    1. Mat(arg0: CSCMatrix<int, unsigned long>) -> bats.linalg.IntMat
    2. Mat(arg0: bats.linalg.CSCMatrix, arg1: ModP<int, 2u>) -> ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >
    3. Mat(arg0: bats.linalg.CSCMatrix, arg1: ModP<int, 3u>) -> ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >
```

Basic Applied Topology Subprograms

```
class bats.Matrix
    Bases: pybind11_builtin.pybind11_object

    ncol(self: bats.dense.Matrix) → int
    nrow(self: bats.dense.Matrix) → int
    print(self: bats.dense.Matrix) → None

bats.Nerve(arg0: List[Set[int]], arg1: int) → bats.topology.SimplicialComplex
bats.NerveDiagram(arg0: bats.topology.CoverDiagram, arg1: int) → bats.topology.SimplicialComplexDiagram
bats.OscillatingRipsZigzagSets(*args, **kwargs)
    Overloaded function.

    1. OscillatingRipsZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float, arg3: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]
        SetDiagram for Oscillating Rips Zigzag constrution.

    2. OscillatingRipsZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float, arg3: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]
        SetDiagram for Oscillating Rips Zigzag constrution.

    3. OscillatingRipsZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float, arg3: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]
        SetDiagram for Oscillating Rips Zigzag constrution.

    4. OscillatingRipsZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float, arg3: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]
        SetDiagram for Oscillating Rips Zigzag constrution.

    5. OscillatingRipsZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float, arg3: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]
        SetDiagram for Oscillating Rips Zigzag constrution.

    6. OscillatingRipsZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float, arg3: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]
        SetDiagram for Oscillating Rips Zigzag constrution.

    7. OscillatingRipsZigzagSets(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float, arg3: float) -> Tuple[bats::Diagram<std::set<unsigned long, std::less<unsigned long>, std::allocator<unsigned long>>, std::vector<unsigned long, std::allocator<unsigned long>>>, List[float]]
        SetDiagram for Oscillating Rips Zigzag constrution.

bats.PLEU(*args, **kwargs)
    Overloaded function.

    1. PLEU(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long>>]

```

PLEU factorization

2. PLEU(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >]

PLEU factorization

bats.PUEL(*args, **kwargs)

Overloaded function.

1. PUEL(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >]

PUEL factorization

2. PUEL(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >]

PUEL factorization

class bats.PersistencePair

Bases: pybind11_builtins.pybind11_object

birth(self: bats.linalg.PersistencePair) → float

birth_ind(self: bats.linalg.PersistencePair) → int

death(self: bats.linalg.PersistencePair) → float

death_ind(self: bats.linalg.PersistencePair) → int

dim(self: bats.linalg.PersistencePair) → int

length(self: bats.linalg.PersistencePair) → float

mid(self: bats.linalg.PersistencePair) → float

class bats.PersistencePair_int

Bases: pybind11_builtins.pybind11_object

birth(self: bats.linalg.PersistencePair_int) → int

birth_ind(self: bats.linalg.PersistencePair_int) → int

death(self: bats.linalg.PersistencePair_int) → int

death_ind(self: bats.linalg.PersistencePair_int) → int

dim(self: bats.linalg.PersistencePair_int) → int

length(self: bats.linalg.PersistencePair_int) → int

mid(self: bats.linalg.PersistencePair_int) → int

class bats.RPAngleDist

Bases: pybind11_builtins.pybind11_object

class bats.RPCosineDist

Bases: pybind11_builtins.pybind11_object

Basic Applied Topology Subprograms

bats.ReducedChainComplex(*args, **kwargs)

Overloaded function.

1. ReducedChainComplex(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.ReducedF2ChainComplex
2. ReducedChainComplex(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.ReducedF2ChainComplex
3. ReducedChainComplex(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedF3ChainComplex
4. ReducedChainComplex(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedF3ChainComplex

class bats.ReducedF2ChainComplex

Bases: pybind11_builtins.pybind11_object

R(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: int) → bats.linalg_f2.F2Mat
reduced matrix in specified dimension

U(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: int) → bats.linalg_f2.F2Mat
basis matrix in specified dimension

chain_preferredRepresentative(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: bats.linalg_f2.F2Vector, arg1: int) → bats.linalg_f2.F2Vector
return the preferred representative of a chain

find_preferredRepresentative(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: bats.linalg_f2.F2Vector, arg1: int) → None

fromHomBasis(*args, **kwargs)

Overloaded function.

1. fromHomBasis(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: bats.linalg_f2.F2Mat, arg1: int) -> bats.linalg_f2.F2Mat
2. fromHomBasis(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: bats.linalg_f2.F2Vector, arg1: int) -> bats.linalg_f2.F2Vector

getPreferredRepresentative(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: int, arg1: int) → bats.linalg_f2.F2Vector
get the preferred representative for homology class

hdim(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: int) → int

maxdim(self: bats.linalg_f2.ReducedF2ChainComplex) → int

toHomBasis(*args, **kwargs)

Overloaded function.

1. toHomBasis(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: bats.linalg_f2.F2Mat, arg1: int) -> bats.linalg_f2.F2Mat
2. toHomBasis(self: bats.linalg_f2.ReducedF2ChainComplex, arg0: bats.linalg_f2.F2Vector, arg1: int) -> bats.linalg_f2.F2Vector

class bats.ReducedF2DGVectorSpace

Bases: pybind11_builtins.pybind11_object

hdim(self: bats.linalg_f2.ReducedF2DGVectorSpace, arg0: int) → int

maxdim(self: bats.linalg_f2.ReducedF2DGVectorSpace) → int

```

class bats.ReducedF3ChainComplex
Bases: pybind11_builtins.pybind11_object

R(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: int) → bats.linalg_f3.F3Mat
    reduced matrix in specified dimension

U(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: int) → bats.linalg_f3.F3Mat
    basis matrix in specified dimension

chain_preferredRepresentative(self: bats.linalg_f3.ReducedF3ChainComplex, arg0:
    bats.linalg_f3.F3Vector, arg1: int) → bats.linalg_f3.F3Vector
    return the preferred representative of a chain

find_preferredRepresentative(self: bats.linalg_f3.ReducedF3ChainComplex, arg0:
    bats.linalg_f3.F3Vector, arg1: int) → None

fromHomBasis(*args, **kwargs)
Overloaded function.

1. fromHomBasis(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: bats.linalg_f3.F3Mat, arg1:
    int) -> bats.linalg_f3.F3Mat
2. fromHomBasis(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: bats.linalg_f3.F3Vector, arg1:
    int) -> bats.linalg_f3.F3Vector

getPreferredRepresentative(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: int, arg1: int) →
    bats.linalg_f3.F3Vector
get the preferred representative for homology class

hdim(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: int) → int

maxdim(self: bats.linalg_f3.ReducedF3ChainComplex) → int

toHomBasis(*args, **kwargs)
Overloaded function.

1. toHomBasis(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: bats.linalg_f3.F3Mat, arg1: int)
    -> bats.linalg_f3.F3Mat
2. toHomBasis(self: bats.linalg_f3.ReducedF3ChainComplex, arg0: bats.linalg_f3.F3Vector, arg1:
    int) -> bats.linalg_f3.F3Vector

class bats.ReducedF3DGVectorSpace
Bases: pybind11_builtins.pybind11_object

hdim(self: bats.linalg_f3.ReducedF3DGVectorSpace, arg0: int) → int

maxdim(self: bats.linalg_f3.ReducedF3DGVectorSpace) → int

bats.ReducedFilteredChainComplex(*args, **kwargs)
Overloaded function.

1. ReducedFilteredChainComplex(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2)
    -> bats.linalg_f2.ReducedFilteredF2ChainComplex
2. ReducedFilteredChainComplex(arg0: bats.topology.FilteredLightSimplicialComplex, arg1:
    bats.linalg_f2.F2) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
3. ReducedFilteredChainComplex(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f2.F2) ->
    bats.linalg_f2.ReducedFilteredF2ChainComplex
4. ReducedFilteredChainComplex(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3)
    -> bats.linalg_f3.ReducedFilteredF3ChainComplex

```

5. ReducedFilteredChainComplex(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
6. ReducedFilteredChainComplex(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedFilteredF3ChainComplex

class bats.ReducedFilteredF2ChainComplex

Bases: pybind11_builtins.pybind11_object

dim(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: int) → int

maxdim(self: bats.linalg_f2.ReducedFilteredF2ChainComplex) → int

nnz_R(self: bats.linalg_f2.ReducedFilteredF2ChainComplex) → List[int]
get the number of non-zeros in R

nnz_U(self: bats.linalg_f2.ReducedFilteredF2ChainComplex) → List[int]
get the number of non-zeros in U

perm(self: bats.linalg_f2.ReducedFilteredF2ChainComplex) → List[List[int]]
permutation from original order

persistence_pairs(*args, **kwargs)

Overloaded function.

1. persistence_pairs(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: int) -> List[bats.linalg.PersistencePair]
2. persistence_pairs(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: int, arg1: bool) -> List[bats.linalg.PersistencePair]

persistence_pairs_vec(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: int) → Tuple[List[float], List[int]]

reduced_complex(self: bats.linalg_f2.ReducedFilteredF2ChainComplex) →
bats.linalg_f2.ReducedF2ChainComplex

underlying reduced complex

representative(*args, **kwargs)

Overloaded function.

1. representative(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: bats.linalg.PersistencePair) -> bats.linalg_f2.F2Vector
2. representative(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: bats.linalg.PersistencePair, arg1: bool) -> bats.linalg_f2.F2Vector

update_filtration(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: List[List[float]]) → None
update filtration with new values

update_filtration_general(*args, **kwargs)

Overloaded function.

1. update_filtration_general(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: bats.topology.UpdateInfoFiltration) -> None

generally update filtration with updating information

2. update_filtration_general(self: bats.linalg_f2.ReducedFilteredF2ChainComplex, arg0: bats.topology.UpdateInfoLightFiltration) -> None

generally update filtration with updating information

val(self: bats.linalg_f2.ReducedFilteredF2ChainComplex) → List[List[float]]

filtration values

```

class bats.ReducedFilteredF2DGVectorSpace
Bases: pybind11_builtins.pybind11_object

dim(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace, arg0: int) → int
hdim(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace, arg0: int) → int
maxdim(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace) → int
persistence_pairs(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace, arg0: int) →
    List[bats.linalg.PersistencePair]
persistence_pairs_vec(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace, arg0: int) →
    Tuple[List[float], List[int]]
update(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace, arg0: bats.topology.UpdateInfo2) → None
    update filtration with permutations, additions, deletions
update_filtration(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace, arg0: List[List[float]]) →
    None
    update filtration with new values
update_filtration_general(*args, **kwargs)
    Overloaded function.
    1. update_filtration_general(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace,      arg0:
        bats.topology.UpdateInfoFiltration) -> None
        generally update filtration with updating information
    2. update_filtration_general(self: bats.linalg_f2.ReducedFilteredF2DGVectorSpace,      arg0:
        bats.topology.UpdateInfoLightFiltration) -> None
        generally update filtration with updating information
class bats.ReducedFilteredF3ChainComplex
Bases: pybind11_builtins.pybind11_object

dim(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: int) → int
maxdim(self: bats.linalg_f3.ReducedFilteredF3ChainComplex) → int
nnz_R(self: bats.linalg_f3.ReducedFilteredF3ChainComplex) → List[int]
    get the number of non-zeros in R
nnz_U(self: bats.linalg_f3.ReducedFilteredF3ChainComplex) → List[int]
    get the number of non-zeros in U
perm(self: bats.linalg_f3.ReducedFilteredF3ChainComplex) → List[List[int]]
    permutation from original order
persistence_pairs(*args, **kwargs)
    Overloaded function.
    1. persistence_pairs(self: bats.linalg_f3.ReducedFilteredF3ChainComplex,      arg0:      int) ->
        List[bats.linalg.PersistencePair]
    2. persistence_pairs(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: int, arg1: bool) ->
        List[bats.linalg.PersistencePair]
persistence_pairs_vec(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: int) →
    Tuple[List[float], List[int]]

```

```
reduced_complex(self: bats.linalg_f3.ReducedFilteredF3ChainComplex) →
    bats.linalg_f3.ReducedF3ChainComplex
    underlying reduced complex

representative(*args, **kwargs)
    Overloaded function.
    1. representative(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: bats.linalg.PersistencePair) -> bats.linalg_f3.F3Vector
    2. representative(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: bats.linalg.PersistencePair, arg1: bool) -> bats.linalg_f3.F3Vector

update_filtration(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: List[List[float]]) → None
    update filtration with new values

update_filtration_general(*args, **kwargs)
    Overloaded function.
    1. update_filtration_general(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: bats.topology.UpdateInfoFiltration) -> None
        generally update filtration with updating information
    2. update_filtration_general(self: bats.linalg_f3.ReducedFilteredF3ChainComplex, arg0: bats.topology.UpdateInfoLightFiltration) -> None
        generally update filtration with updating information

val(self: bats.linalg_f3.ReducedFilteredF3ChainComplex) → List[List[float]]
    filtration values

class bats.ReducedFilteredF3DGVectorSpace
    Bases: pybind11_builtins.pybind11_object

dim(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: int) → int
hdim(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: int) → int
maxdim(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace) → int
persistence_pairs(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: int) →
    List[bats.linalg.PersistencePair]
persistence_pairs_vec(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: int) →
    Tuple[List[float], List[int]]
update(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: bats.topology.UpdateInfo2) → None
    update filtration with permutations, additions, deletions
update_filtration(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: List[List[float]]) →
    None
    update filtration with new values

update_filtration_general(*args, **kwargs)
    Overloaded function.
    1. update_filtration_general(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: bats.topology.UpdateInfoFiltration) -> None
        generally update filtration with updating information
    2. update_filtration_general(self: bats.linalg_f3.ReducedFilteredF3DGVectorSpace, arg0: bats.topology.UpdateInfoLightFiltration) -> None
```

generally update filtration with updating information

bats.Rips(*args, **kwargs)

Overloaded function.

1. Rips(arg0: bats.topology.SetDiagram, arg1: bats::DataSet<double>, arg2: bats.topology.Euclidean, arg3: float, arg4: int) -> bats.topology.SimplicialComplexDiagram

Construct a diagram of Rips complexes from a SetDiagram.

2. Rips(arg0: bats.topology.SetDiagram, arg1: bats::DataSet<double>, arg2: bats.topology.Euclidean, arg3: List[float], arg4: int) -> bats.topology.SimplicialComplexDiagram

Construct a diagram of Rips complexes from a SetDiagram.

bats.RipsComplex(*args, **kwargs)

Overloaded function.

1. RipsComplex(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float, arg3: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from data set and metric.

2. RipsComplex(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float, arg3: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from data set and metric.

3. RipsComplex(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float, arg3: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from data set and metric.

4. RipsComplex(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float, arg3: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from data set and metric.

5. RipsComplex(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float, arg3: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from data set and metric.

6. RipsComplex(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float, arg3: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from data set and metric.

7. RipsComplex(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float, arg3: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from data set and metric.

8. RipsComplex(arg0: A<Dense<double, RowMaj> >, arg1: float, arg2: int) -> bats.topology.SimplicialComplex

Rips Complex constructed from pairwise distances.

bats.RipsCoverFiltration(*args, **kwargs)

Overloaded function.

1. RipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.Euclidean, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

2. RipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.L1Dist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Basic Applied Topology Subprograms

3. RipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.LInfDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
4. RipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.CosineDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
5. RipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.RPCosineDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
6. RipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.AngleDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex
7. RipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.RPAngleDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

bats.RipsFiltration(*args, **kwargs)

Overloaded function.

1. RipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex
2. RipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex
3. RipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex
4. RipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex
5. RipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex
6. RipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex
7. RipsFiltration(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float, arg3: int) -> bats.topology.FilteredSimplicialComplex
8. RipsFiltration(arg0: A<Dense<double>, RowMaj> >, arg1: float, arg2: int) -> bats.topology.FilteredSimplicialComplex

Rips Filtration using built using pairwise distances.

bats.RipsFiltration_extension(*args, **kwargs)

Overloaded function.

1. RipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

2. RipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

3. RipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

4. RipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

5. RipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

6. RipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

7. RipsFiltration_extension(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: float, arg3: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration with inverse map returned

8. RipsFiltration_extension(arg0: A<Dense<double, RowMaj> >, arg1: float, arg2: int) -> Tuple[bats.topology.FilteredSimplicialComplex, List[List[int]]]

Rips Filtration built using pairwise distances Rips with inverse map returned.

```
bats.RipsHom(arg0: bats.topology.SetDiagram, arg1: bats.dense.DataSet, arg2: bats.topology.Euclidean, arg3: List[float], arg4: int, arg5: bats.linalg_f2.F2) → bats.linalg_f2.F2Diagram
```

class bats.SetDiagram

Bases: pybind11_builtins.pybind11_object

add_edge(self: bats.topology.SetDiagram, arg0: int, arg1: int, arg2: List[int]) → int

add_node(self: bats.topology.SetDiagram, arg0: Set[int]) → int

edge_data(self: bats.topology.SetDiagram, arg0: int) → List[int]

edge_source(self: bats.topology.SetDiagram, arg0: int) → int

edge_target(self: bats.topology.SetDiagram, arg0: int) → int

nedge(self: bats.topology.SetDiagram) → int

nnode(self: bats.topology.SetDiagram) → int

node_data(self: bats.topology.SetDiagram, arg0: int) → Set[int]

set_edge(self: bats.topology.SetDiagram, arg0: int, arg1: int, arg2: int, arg3: List[int]) → None

set_node(self: bats.topology.SetDiagram, arg0: int, arg1: Set[int]) → None

bats.SierpinskiDiagram(arg0: int) → bats::Diagram<bats::CellComplex, bats::CellularMap>

Diagram of Sierpinski triangle iterations.

class bats.SimplicialComplex

Bases: pybind11_builtins.pybind11_object

add(self: bats.topology.SimplicialComplex, arg0: List[int]) → bats.topology.cell_ind
add simplex

add_directed(self: bats.topology.SimplicialComplex, arg0: List[int]) → bats.topology.cell_ind
add directed or degenerate simplex

add_directed_recursive(self: bats.topology.SimplicialComplex, arg0: List[int]) → List[bats.topology.cell_ind]
add directed or degenerate simplex recursively

add_recursive(self: bats.topology.SimplicialComplex, arg0: List[int]) → List[bats.topology.cell_ind]
add simplex and missing faces

Basic Applied Topology Subprograms

```
boundary(self: bats.topology.SimplicialComplex, arg0: int) → CSCMatrix<int, unsigned long>
find_idx(self: bats.topology.SimplicialComplex, arg0: List[int]) → int
get_simplex(self: bats.topology.SimplicialComplex, arg0: int, arg1: int) → List[int]
get_simplices(*args, **kwargs)
    Overloaded function.
        1. get_simplices(self: bats.topology.SimplicialComplex, arg0: int) -> List[List[int]]
            Returns a list of all simplices in given dimension.
        2. get_simplices(self: bats.topology.SimplicialComplex) -> List[List[int]]
            Returns a list of all simplices.
maxdim(self: bats.topology.SimplicialComplex) → int
    maximum dimension simplex
ncells(*args, **kwargs)
    Overloaded function.
        1. ncells(self: bats.topology.SimplicialComplex) -> int
            number of cells
        2. ncells(self: bats.topology.SimplicialComplex, arg0: int) -> int
            number of cells in given dimension
class bats.SimplicialComplexDiagram
    Bases: pybind11_builtins.pybind11_object
add_edge(self: bats.topology.SimplicialComplexDiagram, arg0: int, arg1: int, arg2: bats.topology.CellularMap) → int
add_node(self: bats.topology.SimplicialComplexDiagram, arg0: bats.topology.SimplicialComplex) → int
edge_data(self: bats.topology.SimplicialComplexDiagram, arg0: int) → bats.topology.CellularMap
edge_source(self: bats.topology.SimplicialComplexDiagram, arg0: int) → int
edge_target(self: bats.topology.SimplicialComplexDiagram, arg0: int) → int
nedge(self: bats.topology.SimplicialComplexDiagram) → int
nnode(self: bats.topology.SimplicialComplexDiagram) → int
node_data(self: bats.topology.SimplicialComplexDiagram, arg0: int) → bats.topology.SimplicialComplex
set_edge(self: bats.topology.SimplicialComplexDiagram, arg0: int, arg1: int, arg2: int, arg3: bats.topology.CellularMap) → None
set_node(self: bats.topology.SimplicialComplexDiagram, arg0: int, arg1: bats.topology.SimplicialComplex) → None
bats.SimplicialMap(*args, **kwargs)
    Overloaded function.
        1. SimplicialMap(arg0: bats.topology.SimplicialComplex, arg1: bats.topology.SimplicialComplex) -> bats.topology.CellularMap
            Inclusion map of simplicial complexes
        2. SimplicialMap(arg0: bats.topology.SimplicialComplex, arg1: bats.topology.SimplicialComplex, arg2: List[int]) -> bats.topology.CellularMap
```

simplicial map extended from function on vertices

bats.StrictRipsCoverFiltration(*args, **kwargs)

Overloaded function.

1. StrictRipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.Euclidean, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Rips Cover Filtration

2. StrictRipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.L1Dist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Rips Cover Filtration

3. StrictRipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.LInfDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Rips Cover Filtration

4. StrictRipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.CosineDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Rips Cover Filtration

5. StrictRipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.RPCosineDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Rips Cover Filtration

6. StrictRipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.AngleDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Rips Cover Filtration

7. StrictRipsCoverFiltration(arg0: bats::DataSet<double>, arg1: List[Set[int]], arg2: bats.topology.RPAngleDist, arg3: float, arg4: int) -> bats.topology.FilteredSimplicialComplex

Rips Cover Filtration

bats.TriangulatedProduct(arg0: bats.topology.SimplicialComplex, arg1: bats.topology.SimplicialComplex)
 \rightarrow bats.topology.SimplicialComplex

bats.UELP(*args, **kwargs)

Overloaded function.

1. UELP(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >]

UELP factorization

2. UELP(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> Tuple[ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >]

UELP factorization

bats.U_EU_commute(*args, **kwargs)

Overloaded function.

1. U_EU_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >

U, E_U commutation

2. U_EU_commute(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >

U, E_U commutation

class bats.UpdateInfo2

Bases: pybind11_builtins.pybind11_object

property insertion_cols

property insertion_indices

property ndeletions

property perm

reverse_for_cohomology(self: bats.topology.UpdateInfo2) → None

reverse for cohomology

class bats.UpdateInfoFiltration

Bases: pybind11_builtins.pybind11_object

property addition_indices

property deletion_indices

filtered_info(self: bats.topology.UpdateInfoFiltration, arg0: List[List[int]], arg1: int) → None

if the cells in filtration are not sorted by their filtration values, we find filtered updating information

property kendall_tau_dists

property permutations

class bats.UpdateInfoLightFiltration

Bases: pybind11_builtins.pybind11_object

property addition_indices

property deletion_indices

filtered_info(self: bats.topology.UpdateInfoLightFiltration, arg0: List[List[int]], arg1: int) → None

if the cells in filtration are not sorted by their filtration values, we find filtered updating information

property kendall_tau_dists

property permutations

bats.WitnessFiltration(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2:

bats.topology.Euclidean, arg3: float, arg4: int) →

bats.topology.FilteredSimplicialComplex

bats.ZigzagBarcode(*args, **kwargs)

Overloaded function.

1. ZigzagBarcode(arg0: bats.topology.ZigzagCubicalComplex, arg1: int, arg2: ModP<int, 2u>) -> List[List[bats::zigzag::ZigzagPair<double>]]

2. ZigzagBarcode(arg0: bats.topology.ZigzagCubicalComplex, arg1: int, arg2: ModP<int, 2u>, arg3: bats::extra_reduction_flag) -> List[List[bats::zigzag::ZigzagPair<double>]]

```

3. ZigzagBarcode(arg0: bats.topology.ZigzagSimplicialComplex, arg1: int, arg2: ModP<int, 2u>) ->
List[List[bats::zigzag::ZigzagPair<double>]]

4. ZigzagBarcode(arg0: bats.topology.ZigzagSimplicialComplex, arg1: int, arg2: ModP<int, 2u>, arg3:
bats::extra_reduction_flag) -> List[List[bats::zigzag::ZigzagPair<double>]]
```

class bats.ZigzagCubicalComplex

Bases: pybind11_builtins.pybind11_object

- add(self: bats.topology.ZigzagCubicalComplex, arg0: float, arg1: float, arg2: List[int]) ->**
bats.topology.cell_ind
- add_recursive(self: bats.topology.ZigzagCubicalComplex, arg0: float, arg1: float, arg2: List[int]) ->**
List[bats.topology.cell_ind]
- complex(self: bats.topology.ZigzagCubicalComplex) -> bats.topology.CubicalComplex**
- levelset(self: bats.topology.ZigzagCubicalComplex, arg0: float, arg1: float) ->**
bats.topology.CubicalComplex
- maxdim(self: bats.topology.ZigzagCubicalComplex) -> int**
- ncells(self: bats.topology.ZigzagCubicalComplex, arg0: int) -> int**
- vals(*args, **kwargs)**
Overloaded function.
 1. vals(self: bats.topology.ZigzagCubicalComplex) -> List[List[List[Tuple[float, float]]]]
 2. vals(self: bats.topology.ZigzagCubicalComplex, arg0: int) -> List[List[Tuple[float, float]]]

class bats.ZigzagPair

Bases: pybind11_builtins.pybind11_object

- birth(self: bats.linalg.ZigzagPair) -> float**
- birth_ind(self: bats.linalg.ZigzagPair) -> int**
- death(self: bats.linalg.ZigzagPair) -> float**
- death_ind(self: bats.linalg.ZigzagPair) -> int**
- dim(self: bats.linalg.ZigzagPair) -> int**
- length(self: bats.linalg.ZigzagPair) -> float**
- mid(self: bats.linalg.ZigzagPair) -> float**

bats.ZigzagSetDiagram(arg0: List[Set[int]]) -> bats.topology.SetDiagram
Create a zigzag diagram of sets and pairwise unions.

class bats.ZigzagSimplicialComplex

Bases: pybind11_builtins.pybind11_object

- add(self: bats.topology.ZigzagSimplicialComplex, arg0: float, arg1: float, arg2: List[int]) ->**
bats.topology.cell_ind
- add_recursive(self: bats.topology.ZigzagSimplicialComplex, arg0: float, arg1: float, arg2: List[int]) ->**
List[bats.topology.cell_ind]
- complex(self: bats.topology.ZigzagSimplicialComplex) -> bats.topology.SimplicialComplex**
- levelset(self: bats.topology.ZigzagSimplicialComplex, arg0: float, arg1: float) ->**
bats.topology.SimplicialComplex
- maxdim(self: bats.topology.ZigzagSimplicialComplex) -> int**

Basic Applied Topology Subprograms

ncells(*self*: bats.topology.ZigzagSimplicialComplex, *arg0*: int) → int

vals(**args*, ***kargs*)

Overloaded function.

1. vals(*self*: bats.topology.ZigzagSimplicialComplex) -> List[List[List[Tuple[float, float]]]]

2. vals(*self*: bats.topology.ZigzagSimplicialComplex, *arg0*: int) -> List[List[Tuple[float, float]]]

bats.**approx_center**(**args*, ***kargs*)

Overloaded function.

1. approx_center(*arg0*: bats::DataSet<double>, *arg1*: bats.topology.Euclidean, *arg2*: int, *arg3*: int) -> int

2. approx_center(*arg0*: bats::DataSet<double>, *arg1*: bats.topology.L1Dist, *arg2*: int, *arg3*: int) -> int

3. approx_center(*arg0*: bats::DataSet<double>, *arg1*: bats.topology.LInfDist, *arg2*: int, *arg3*: int) -> int

4. approx_center(*arg0*: bats::DataSet<double>, *arg1*: bats.topology.CosineDist, *arg2*: int, *arg3*: int) -> int

5. approx_center(*arg0*: bats::DataSet<double>, *arg1*: bats.topology.RPCosineDist, *arg2*: int, *arg3*: int) -> int

6. approx_center(*arg0*: bats::DataSet<double>, *arg1*: bats.topology.AngleDist, *arg2*: int, *arg3*: int) -> int

7. approx_center(*arg0*: bats::DataSet<double>, *arg1*: bats.topology.RPAngleDist, *arg2*: int, *arg3*: int) -> int

bats.**barcode**(**args*, ***kargs*)

Overloaded function.

1. barcode(*arg0*: bats.linalg_f2.F2HomDiagram, *arg1*: int) -> List[bats.linalg.PersistencePair_int]

2. barcode(*arg0*: bats.linalg_f2.F2HomDiagram, *arg1*: int, *arg2*: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]

3. barcode(*arg0*: bats.linalg_f2.F2HomDiagram, *arg1*: int, *arg2*: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]

4. barcode(*arg0*: bats.linalg_f2.F2HomDiagram, *arg1*: int, *arg2*: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]

5. barcode(*arg0*: bats.linalg_f2.F2HomDiagramAll) -> List[bats.linalg.PersistencePair_int]

6. barcode(*arg0*: bats.linalg_f2.F2HomDiagramAll, *arg1*: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]

7. barcode(*arg0*: bats.linalg_f2.F2HomDiagramAll, *arg1*: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]

8. barcode(*arg0*: bats.linalg_f2.F2HomDiagramAll, *arg1*: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]

9. barcode(*arg0*: bats.linalg_f2.F2DGHomDiagram, *arg1*: int) -> List[bats.linalg.PersistencePair_int]

10. barcode(*arg0*: bats.linalg_f2.F2DGHomDiagram, *arg1*: int, *arg2*: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]

11. barcode(*arg0*: bats.linalg_f2.F2DGHomDiagram, *arg1*: int, *arg2*: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]

12. barcode(*arg0*: bats.linalg_f2.F2DGHomDiagram, *arg1*: int, *arg2*: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]

13. barcode(*arg0*: bats.linalg_f2.F2DGHomDiagramAll) -> List[bats.linalg.PersistencePair_int]

14. barcode(*arg0*: bats.linalg_f2.F2DGHomDiagramAll, *arg1*: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]

15. barcode(arg0: bats.linalg_f2.F2DGHomDiagramAll, arg1: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]
16. barcode(arg0: bats.linalg_f2.F2DGHomDiagramAll, arg1: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]
17. barcode(arg0: bats.linalg_f2.F2Diagram, arg1: int) -> List[bats.linalg.PersistencePair_int]
18. barcode(arg0: bats.linalg_f2.F2Diagram, arg1: int, arg2: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]
19. barcode(arg0: bats.linalg_f2.F2Diagram, arg1: int, arg2: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]
20. barcode(arg0: bats.linalg_f2.F2Diagram, arg1: int, arg2: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]
21. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >>>) -> List[bats.linalg.PersistencePair_int]
22. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >>>, arg1: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]
23. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >>>, arg1: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]
24. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >>>, arg1: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]
25. barcode(arg0: bats.linalg_f3.F3HomDiagram, arg1: int) -> List[bats.linalg.PersistencePair_int]
26. barcode(arg0: bats.linalg_f3.F3HomDiagram, arg1: int, arg2: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]
27. barcode(arg0: bats.linalg_f3.F3HomDiagram, arg1: int, arg2: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]
28. barcode(arg0: bats.linalg_f3.F3HomDiagram, arg1: int, arg2: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]
29. barcode(arg0: bats.linalg_f3.F3HomDiagramAll) -> List[bats.linalg.PersistencePair_int]
30. barcode(arg0: bats.linalg_f3.F3HomDiagramAll, arg1: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]
31. barcode(arg0: bats.linalg_f3.F3HomDiagramAll, arg1: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]
32. barcode(arg0: bats.linalg_f3.F3HomDiagramAll, arg1: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]
33. barcode(arg0: bats.linalg_f3.F3DGHomDiagram, arg1: int) -> List[bats.linalg.PersistencePair_int]
34. barcode(arg0: bats.linalg_f3.F3DGHomDiagram, arg1: int, arg2: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]
35. barcode(arg0: bats.linalg_f3.F3DGHomDiagram, arg1: int, arg2: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]

```

36. barcode(arg0: bats.linalg_f3.F3DGHomDiagram, arg1: int, arg2: bats.linalg.leftward) ->
    List[bats.linalg.PersistencePair_int]

37. barcode(arg0: bats.linalg_f3.F3DGHomDiagramAll) -> List[bats.linalg.PersistencePair_int]

38. barcode(arg0: bats.linalg_f3.F3DGHomDiagramAll, arg1: bats.linalg.divide_conquer) ->
    List[bats.linalg.PersistencePair_int]

39. barcode(arg0: bats.linalg_f3.F3DGHomDiagramAll, arg1: bats.linalg.rightward) ->
    List[bats.linalg.PersistencePair_int]

40. barcode(arg0: bats.linalg_f3.F3DGHomDiagramAll, arg1: bats.linalg.leftward) ->
    List[bats.linalg.PersistencePair_int]

41. barcode(arg0: bats.linalg_f3.F3Diagram, arg1: int) -> List[bats.linalg.PersistencePair_int]

42. barcode(arg0: bats.linalg_f3.F3Diagram, arg1: int, arg2: bats.linalg.divide_conquer) ->
    List[bats.linalg.PersistencePair_int]

43. barcode(arg0: bats.linalg_f3.F3Diagram, arg1: int, arg2: bats.linalg.rightward) ->
    List[bats.linalg.PersistencePair_int]

44. barcode(arg0: bats.linalg_f3.F3Diagram, arg1: int, arg2: bats.linalg.leftward) ->
    List[bats.linalg.PersistencePair_int]

45. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >>>) -> List[bats.linalg.PersistencePair_int]

46. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >>>, arg1: bats.linalg.divide_conquer) -> List[bats.linalg.PersistencePair_int]

47. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >>>, arg1: bats.linalg.rightward) -> List[bats.linalg.PersistencePair_int]

48. barcode(arg0: bats::Diagram<unsigned long, std::vector<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, std::allocator<ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >>>, arg1: bats.linalg.leftward) -> List[bats.linalg.PersistencePair_int]

```

bats.bivariate_cover(arg0: List[Set[int]], arg1: List[Set[int]]) → Tuple[List[Set[int]], List[int], List[int]]

class bats.cell_ind

Bases: pybind11_builtins.pybind11_object

class bats.clearing_flag

Bases: pybind11_builtins.pybind11_object

class bats.compression_flag

Bases: pybind11_builtins.pybind11_object

class bats.compute_basis_flag

Bases: pybind11_builtins.pybind11_object

class bats.divide_conquer

Bases: pybind11_builtins.pybind11_object

bats.enclosing_radius(arg0: A<Dense<double, RowMaj> >) → float

Enclosing radius from matrix of pairwise distances

bats.extend_zigzag_filtration(arg0: List[float], arg1: bats.topology.SimplicialComplex, arg2: float) →
bats.topology.ZigzagSimplicialComplex

```
class bats.extra_reduction_flag
    Bases: pybind11_builtins.pybind11_object

bats.extract_pairs(arg0: bats.linalg_f3.F3Mat, arg1: List[int], arg2: List[float], arg3: List[float], arg4: int)
    → List[bats.linalg.PersistencePair]
    extract persistence pairs

bats.force_repel_rp(arg0: bats::DataSet<double>, arg1: float) → None

bats.greedy_landmarks(*args, **kwargs)
    Overloaded function.
    1. greedy_landmarks(arg0: bats::DataSet<double>, arg1: int, arg2: bats.topology.Euclidean, arg3: int) ->
        bats::DataSet<double>
    2. greedy_landmarks(arg0: bats::DataSet<double>, arg1: int, arg2: bats.topology.L1Dist, arg3: int) ->
        bats::DataSet<double>
    3. greedy_landmarks(arg0: bats::DataSet<double>, arg1: int, arg2: bats.topology.LInfDist, arg3: int) ->
        bats::DataSet<double>
    4. greedy_landmarks(arg0: bats::DataSet<double>, arg1: int, arg2: bats.topology.CosineDist, arg3: int) ->
        bats::DataSet<double>
    5. greedy_landmarks(arg0: bats::DataSet<double>, arg1: int, arg2: bats.topology.RPCosineDist, arg3: int)
        -> bats::DataSet<double>
    6. greedy_landmarks(arg0: bats::DataSet<double>, arg1: int, arg2: bats.topology.AngleDist, arg3: int) ->
        bats::DataSet<double>
    7. greedy_landmarks(arg0: bats::DataSet<double>, arg1: int, arg2: bats.topology.RPAngleDist, arg3: int) ->
        bats::DataSet<double>

bats.greedy_landmarks_hausdorff(*args, **kwargs)
    Overloaded function.
    1. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: int) ->
        Tuple[List[int], List[float]]
    2. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.Euclidean, arg2: int, arg3: int)
        -> Tuple[List[int], List[float]]
    3. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: int) ->
        Tuple[List[int], List[float]]
    4. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.L1Dist, arg2: int, arg3: int)
        -> Tuple[List[int], List[float]]
    5. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: int) ->
        Tuple[List[int], List[float]]
    6. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.LInfDist, arg2: int, arg3: int)
        -> Tuple[List[int], List[float]]
    7. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: int) ->
        Tuple[List[int], List[float]]
    8. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.CosineDist, arg2: int, arg3: int)
        -> Tuple[List[int], List[float]]
    9. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: int)
        -> Tuple[List[int], List[float]]
```

Basic Applied Topology Subprograms

10. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.RPCosineDist, arg2: int, arg3: int) -> Tuple[List[int], List[float]]
11. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: int) -> Tuple[List[int], List[float]]
12. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.AngleDist, arg2: int, arg3: int) -> Tuple[List[int], List[float]]
13. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: int) -> Tuple[List[int], List[float]]
14. greedy_landmarks_hausdorff(arg0: bats::DataSet<double>, arg1: bats.topology.RPAngleDist, arg2: int, arg3: int) -> Tuple[List[int], List[float]]
15. greedy_landmarks_hausdorff(arg0: A<Dense<double, RowMaj> >, arg1: int) -> Tuple[List[int], List[float]]

bats.hausdorff_landmarks(*args, **kwargs)

Overloaded function.

1. hausdorff_landmarks(arg0: bats::DataSet<double>, arg1: float, arg2: bats.topology.Euclidean, arg3: int) -> bats::DataSet<double>
2. hausdorff_landmarks(arg0: bats::DataSet<double>, arg1: float, arg2: bats.topology.L1Dist, arg3: int) -> bats::DataSet<double>
3. hausdorff_landmarks(arg0: bats::DataSet<double>, arg1: float, arg2: bats.topology.LInfDist, arg3: int) -> bats::DataSet<double>
4. hausdorff_landmarks(arg0: bats::DataSet<double>, arg1: float, arg2: bats.topology.CosineDist, arg3: int) -> bats::DataSet<double>
5. hausdorff_landmarks(arg0: bats::DataSet<double>, arg1: float, arg2: bats.topology.RPCosineDist, arg3: int) -> bats::DataSet<double>
6. hausdorff_landmarks(arg0: bats::DataSet<double>, arg1: float, arg2: bats.topology.AngleDist, arg3: int) -> bats::DataSet<double>
7. hausdorff_landmarks(arg0: bats::DataSet<double>, arg1: float, arg2: bats.topology.RPAngleDist, arg3: int) -> bats::DataSet<double>

bats.kendall_tau(*args, **kwargs)

Overloaded function.

1. kendall_tau(arg0: List[int]) -> int

Kendall-Tau distance between permutation and identity.

2. kendall_tau(arg0: List[int], arg1: List[int]) -> int

Kendall-Tau distance between two permutations.

bats.landmark_cover(*args, **kwargs)

Overloaded function.

1. landmark_cover(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.Euclidean, arg3: int) -> List[Set[int]]
2. landmark_cover(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.L1Dist, arg3: int) -> List[Set[int]]
3. landmark_cover(arg0: bats::DataSet<double>, arg1: bats::DataSet<double>, arg2: bats.topology.LInfDist, arg3: int) -> List[Set[int]]

4. landmark_cover(arg0: bats::DataSet<double>, bats.topology.CosineDist, arg3: int) -> List[Set[int]]
5. landmark_cover(arg0: bats::DataSet<double>, bats.topology.RPCosineDist, arg3: int) -> List[Set[int]]
6. landmark_cover(arg0: bats::DataSet<double>, bats.topology.AngleDist, arg3: int) -> List[Set[int]]
7. landmark_cover(arg0: bats::DataSet<double>, bats.topology.RPAngleDist, arg3: int) -> List[Set[int]]

bats.landmark_eps_cover(*args, **kwargs)

Overloaded function.

1. landmark_eps_cover(arg0: bats::DataSet<double>, bats.topology.Euclidean, arg3: float) -> List[Set[int]]
2. landmark_eps_cover(arg0: bats::DataSet<double>, bats.topology.L1Dist, arg3: float) -> List[Set[int]]
3. landmark_eps_cover(arg0: bats::DataSet<double>, bats.topology.LInfDist, arg3: float) -> List[Set[int]]
4. landmark_eps_cover(arg0: bats::DataSet<double>, bats.topology.CosineDist, arg3: float) -> List[Set[int]]
5. landmark_eps_cover(arg0: bats::DataSet<double>, bats.topology.RPCosineDist, arg3: float) -> List[Set[int]]
6. landmark_eps_cover(arg0: bats::DataSet<double>, bats.topology.AngleDist, arg3: float) -> List[Set[int]]
7. landmark_eps_cover(arg0: bats::DataSet<double>, bats.topology.RPAngleDist, arg3: float) -> List[Set[int]]

class bats.leftward

Bases: pybind11_builtins.pybind11_object

bats.lower_star_backwards(arg0: List[List[float]], arg1: List[List[int]], arg2: List[List[int]]) -> List[float]
compute gradient on function from gradient on lower star filtration diagram

bats.lower_star_filtration(*args, **kwargs)

Overloaded function.

1. lower_star_filtration(arg0: bats.topology.SimplicialComplex, arg1: List[float]) -> Tuple[List[List[float]], List[List[int]]]

extend function on 0-cells to filtration

2. lower_star_filtration(arg0: bats.topology.LightSimplicialComplex, arg1: List[float]) -> Tuple[List[List[float]], List[List[int]]]

extend function on 0-cells to filtration

3. lower_star_filtration(arg0: bats.topology.CubicalComplex, arg1: List[List[float]]) -> List[List[float]]

extend function on 0-cells to filtration

4. lower_star_filtration(arg0: bats.topology.CubicalComplex, arg1: List[List[List[float]]]) -> List[List[float]]

extend function on 0-cells to filtration

class bats.no_optimization_flag

Bases: pybind11_builtins.pybind11_object

Basic Applied Topology Subprograms

bats.persistence_barcode(*ps*, *remove_zeros*=True, *tlb*=0.0, *tub*=inf, *essential_color*=None, *figargs*={}, *lineargs*={'*linewidth*': 1})

plot a persistence barcode for pairs in *ps*. Each barcode is a pyplot axis. Barcodes are stacked horizontally in figure.

Parameters:

ps: List of PersistencePair
 remove_zeros: flag to remove zero-length bars (default: True)
 tlb: lower bound on filtration parameter to display. (default: 0.0)
 tub: upper bound on filtration parameter to display. (default: inf)
 essential_color: color for essential pairs (defualt: ‘r’)
 figargs - passed onto pyplot subplots construction (default { })
 lineargs - passed onto hlines construction (default {“*linewidth*”:1})

Returns: *fig*, *ax* - pyplot figure and axes

bats.persistence_diagram(*ps*, *remove_zeros*=True, *show_legend*=True, *tmax*=0.0, *tmin*=0.0, *ax*=None, ***kwargs*)

Plot persistence diagram of PersistencePair

bats.random_landmarks(*arg0*: bats::DataSet<double>, *arg1*: int) → bats::DataSet<double>

bats.reduce(**args*, ***kwargs*)

Overloaded function.

1. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2) ->
 bats.linalg_f2.ReducedF2ChainComplex
2. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2, *arg2*:
 bats.linalg.standard_reduction_flag) -> bats.linalg_f2.ReducedF2ChainComplex
3. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2, *arg2*:
 bats.linalg.standard_reduction_flag, *arg3*: bats.linalg.compute_basis_flag) ->
 bats.linalg_f2.ReducedF2ChainComplex
4. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2,
 arg2: bats.linalg.standard_reduction_flag, *arg3*: bats.linalg.clearing_flag) ->
 bats.linalg_f2.ReducedF2ChainComplex
5. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2, *arg2*:
 bats.linalg.standard_reduction_flag, *arg3*: bats.linalg.compression_flag) ->
 bats.linalg_f2.ReducedF2ChainComplex
6. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2, *arg2*:
 bats.linalg.standard_reduction_flag, *arg3*: bats.linalg.compression_flag,
 bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex
7. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2, *arg2*:
 bats.linalg.extra_reduction_flag) -> bats.linalg_f2.ReducedF2ChainComplex
8. reduce(*arg0*: bats.topology.SimplicialComplex, *arg1*: bats.linalg_f2.F2, *arg2*:
 bats.linalg.extra_reduction_flag, *arg3*: bats.linalg.compute_basis_flag) ->
 bats.linalg_f2.ReducedF2ChainComplex

9. reduce(arg0: bats.topology.SimplicialComplex, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.clearing_flag) ->
10. reduce(arg0: bats.topology.SimplicialComplex, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.compression_flag) ->
11. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex
12. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.ReducedF2ChainComplex
13. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f2.ReducedF2ChainComplex
14. reduce(arg0: bats.topology.LightSimplicialComplex, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.compute_basis_flag) ->
15. reduce(arg0: bats.topology.LightSimplicialComplex, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.clearing_flag) ->
16. reduce(arg0: bats.topology.LightSimplicialComplex, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.compression_flag) ->
17. reduce(arg0: bats.topology.LightSimplicialComplex, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex
18. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag) -> bats.linalg_f2.ReducedF2ChainComplex
19. reduce(arg0: bats.topology.LightSimplicialComplex, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.compute_basis_flag) ->
20. reduce(arg0: bats.topology.LightSimplicialComplex, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.clearing_flag) ->
21. reduce(arg0: bats.topology.LightSimplicialComplex, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg_f2.ReducedF2ChainComplex, arg1: bats.linalg_f2.F2, bats.linalg.compression_flag) ->
22. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex
23. reduce(arg0: bats.linalg_f2.F2ChainComplex) -> bats.linalg_f2.ReducedF2ChainComplex
24. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.standard_reduction_flag) -> bats.linalg_f2.ReducedF2ChainComplex
25. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex

26. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedF2ChainComplex
27. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedF2ChainComplex
28. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex
29. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.extra_reduction_flag) -> bats.linalg_f2.ReducedF2ChainComplex
30. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex
31. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedF2ChainComplex
32. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedF2ChainComplex
33. reduce(arg0: bats.linalg_f2.F2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedF2ChainComplex
34. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedF3ChainComplex
35. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f3.ReducedF3ChainComplex
36. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
37. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedF3ChainComplex
38. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedF3ChainComplex
39. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
40. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag) -> bats.linalg_f3.ReducedF3ChainComplex
41. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
42. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedF3ChainComplex
43. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedF3ChainComplex

44. reduce(arg0: bats.topology.SimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
45. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedF3ChainComplex
46. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f3.ReducedF3ChainComplex
47. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
48. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedF3ChainComplex
49. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedF3ChainComplex
50. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
51. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag) -> bats.linalg_f3.ReducedF3ChainComplex
52. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
53. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedF3ChainComplex
54. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedF3ChainComplex
55. reduce(arg0: bats.topology.LightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
56. reduce(arg0: bats.linalg_f3.F3ChainComplex) -> bats.linalg_f3.ReducedF3ChainComplex
57. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.standard_reduction_flag) -> bats.linalg_f3.ReducedF3ChainComplex
58. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
59. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedF3ChainComplex
60. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedF3ChainComplex
61. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex

62. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.extra_reduction_flag) -> bats.linalg_f3.ReducedF3ChainComplex
63. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
64. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedF3ChainComplex
65. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedF3ChainComplex
66. reduce(arg0: bats.linalg_f3.F3ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedF3ChainComplex
67. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
68. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
69. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
70. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
71. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
72. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
73. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
74. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
75. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
76. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
77. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
78. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex

79. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
80. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
81. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
82. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
83. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
84. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
85. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
86. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
87. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
88. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
89. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
90. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
91. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
92. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f2.F2) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
93. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
94. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
95. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex

96. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
97. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f2.F2, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
98. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
99. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.standard_reduction_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
100. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
101. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
102. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
103. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
104. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.extra_reduction_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
105. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
106. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
107. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
108. reduce(arg0: bats.linalg_f2.FilteredF2ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f2.ReducedFilteredF2ChainComplex
109. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
110. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
111. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
112. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
113. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
114. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex

115. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
116. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
117. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
118. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
119. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
120. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
121. reduce(arg0: bats.topology.FilteredSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
122. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
123. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
124. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
125. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
126. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
127. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
128. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
129. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
130. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex

131. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
132. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
133. reduce(arg0: bats.topology.FilteredLightSimplicialComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.extra_reduction_flag, arg3: bats.linalg.compression_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
134. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f3.F3) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
135. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
136. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
137. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
138. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.clearing_flag, arg4: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
139. reduce(arg0: bats.topology.FilteredCubicalComplex, arg1: bats.linalg_f3.F3, arg2: bats.linalg.standard_reduction_flag, arg3: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
140. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
141. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.standard_reduction_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
142. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
143. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
144. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
145. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.standard_reduction_flag, arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
146. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.extra_reduction_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
147. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compute_basis_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
148. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.clearing_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex
149. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.extra_reduction_flag, arg2: bats.linalg.compression_flag) -> bats.linalg_f3.ReducedFilteredF3ChainComplex

```
150. reduce(arg0: bats.linalg_f3.FilteredF3ChainComplex, arg1: bats.linalg.extra_reduction_flag,
            arg2: bats.linalg.compression_flag, arg3: bats.linalg.compute_basis_flag) ->
            bats.linalg_f3.ReducedFilteredF3ChainComplex
```

bats.reduce_matrix(*args, **kwargs)

Overloaded function.

1. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> List[int]
2. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, arg1: bats::extra_reduction_flag) -> List[int]
3. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >) -> List[int]
4. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 2u>, unsigned long> >, arg2: bats::extra_reduction_flag) -> List[int]
5. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> List[int]
6. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, arg1: bats::extra_reduction_flag) -> List[int]
7. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >) -> List[int]
8. reduce_matrix(arg0: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, arg1: ColumnMatrix<SparseVector<ModP<int, 3u>, unsigned long> >, arg2: bats::extra_reduction_flag) -> List[int]

class bats.rightward

Bases: pybind11_builtins.pybind11_object

bats.rips_union_find_pairs(arg0: List[int], arg1: List[float]) → List[bats.linalg.PersistencePair]
find Rips pairs

bats.sample_sphere(arg0: int, arg1: int) → bats::DataSet<double>

class bats.standard_reduction_flag

Bases: pybind11_builtins.pybind11_object

bats.union_find_pairs(*args, **kwargs)

Overloaded function.

1. union_find_pairs(arg0: bats.linalg_f2.FilteredF2ChainComplex) -> List[bats.linalg.PersistencePair]
2. union_find_pairs(arg0: bats.linalg_f3.FilteredF3ChainComplex) -> List[bats.linalg.PersistencePair]

bats.zigzag_levelsets(*args, **kwargs)

Overloaded function.

1. zigzag_levelsets(arg0: bats.topology.ZigzagCubicalComplex, arg1: float, arg2: float, arg3: float) -> bats::Diagram<bats::CubicalComplex, bats::CellularMap>
2. zigzag_levelsets(arg0: bats.topology.ZigzagSimplicialComplex, arg1: float, arg2: float, arg3: float) -> bats::Diagram<bats::SimplicialComplex, bats::CellularMap>

bats.zigzag_toplex(arg0: List[List[List[float]]]) → bats.topology.ZigzagCubicalComplex

1.6 About BATS

Python bindings for BATS

People who have contributed to BATS.py:

- Brad Nelson
- Yuan Luo

1.6.1 Citing BATS

If you find this code useful, please consider citing one of the following papers, depending on your use case:

- [Persistent and Zigzag Homology: A Matrix Factorization Viewpoint](#) by Gunnar Carlsson, Anjan Dwaraknath, and Bradley J. Nelson.
- [Accelerating Iterated Persistent Homology Computations with Warm Starts](#) by Yuan Luo and Bradley J. Nelson

**CHAPTER
TWO**

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

bats, 84

INDEX

A

add() (*bats.CellComplex method*), 84
add() (*bats.CubicalComplex method*), 86
add() (*bats.FilteredCubicalComplex method*), 98
add() (*bats.FilteredLightSimplicialComplex method*), 100
add() (*bats.FilteredSimplicialComplex method*), 100
add() (*bats.LightSimplicialComplex method*), 105
add() (*bats.SimplicialComplex method*), 115
add() (*bats.ZigzagCubicalComplex method*), 119
add() (*bats.ZigzagSimplicialComplex method*), 119
add_directed() (*bats.FilteredSimplicialComplex method*), 100
add_directed() (*bats.SimplicialComplex method*), 115
add_directed_recursive() (*bats.SimplicialComplex method*), 115
add_edge() (*bats.CellComplexDiagram method*), 84
add_edge() (*bats.CoverDiagram method*), 86
add_edge() (*bats.CubicalComplexDiagram method*), 87
add_edge() (*bats.F2ChainDiagram method*), 91
add_edge() (*bats.F2DGHomDiagram method*), 91
add_edge() (*bats.F2DGHomDiagramAll method*), 91
add_edge() (*bats.F2DGLinearDiagram method*), 92
add_edge() (*bats.F2Diagram method*), 92
add_edge() (*bats.F2HomDiagram method*), 93
add_edge() (*bats.F2HomDiagramAll method*), 93
add_edge() (*bats.F3ChainDiagram method*), 94
add_edge() (*bats.F3DGHomDiagram method*), 95
add_edge() (*bats.F3DGHomDiagramAll method*), 95
add_edge() (*bats.F3DGLinearDiagram method*), 96
add_edge() (*bats.F3Diagram method*), 96
add_edge() (*bats.F3HomDiagram method*), 97
add_edge() (*bats.F3HomDiagramAll method*), 97
add_edge() (*bats.SetDiagram method*), 115
add_edge() (*bats.SimplicialComplexDiagram method*), 116
add_node() (*bats.CellComplexDiagram method*), 84
add_node() (*bats.CoverDiagram method*), 86
add_node() (*bats.CubicalComplexDiagram method*), 87
add_node() (*bats.F2ChainDiagram method*), 91
add_node() (*bats.F2DGHomDiagram method*), 91
add_node() (*bats.F2DGHomDiagramAll method*), 91
add_node() (*bats.F2DGLinearDiagram method*), 92
add_node() (*bats.F2Diagram method*), 92
add_node() (*bats.F2HomDiagram method*), 93
add_node() (*bats.F2HomDiagramAll method*), 93
add_node() (*bats.F3ChainDiagram method*), 95
add_node() (*bats.F3DGHomDiagram method*), 95
add_node() (*bats.F3DGHomDiagramAll method*), 95
add_node() (*bats.F3DGLinearDiagram method*), 96
add_node() (*bats.F3Diagram method*), 96
add_node() (*bats.F3HomDiagram method*), 97
add_node() (*bats.F3HomDiagramAll method*), 97
add_node() (*bats.SetDiagram method*), 115
add_node() (*bats.SimplicialComplexDiagram method*), 116
add_recursive() (*bats.CubicalComplex method*), 86
add_recursive() (*bats.FilteredCubicalComplex method*), 98
add_recursive() (*bats.FilteredLightSimplicialComplex method*), 100
add_recursive() (*bats.FilteredSimplicialComplex method*), 100
add_recursive() (*bats.LightSimplicialComplex method*), 105
add_recursive() (*bats.SimplicialComplex method*), 115
add_recursive() (*bats.ZigzagCubicalComplex method*), 119
add_recursive() (*bats.ZigzagSimplicialComplex method*), 119
add_vertex() (*bats.CellComplex method*), 84
add_vertices() (*bats.CellComplex method*), 84
addition_indices (*bats.UpdateInfoFiltration property*), 118
addition_indices (*bats.UpdateInfoLightFiltration property*), 118
all_vals() (*bats.FilteredCubicalComplex method*), 98
all_vals() (*bats.FilteredLightSimplicialComplex method*), 100
all_vals() (*bats.FilteredSimplicialComplex method*), 100
AngleDist (*class in bats*), 84
append_column() (*bats.F2Mat method*), 93

append_column() (*bats.F3Mat method*), 97
 append_column() (*bats.IntMat method*), 102
 approx_center() (*in module bats*), 120
 axpy() (*bats.F2Vector method*), 94
 axpy() (*bats.F3Vector method*), 98
 axpy() (*bats.IntVector method*), 102

B

barcode() (*in module bats*), 120
 bats
 module, 84
 birth() (*bats.PersistencePair method*), 107
 birth() (*bats.PersistencePair_int method*), 107
 birth() (*bats.ZigzagPair method*), 119
 birth_ind() (*bats.PersistencePair method*), 107
 birth_ind() (*bats.PersistencePair_int method*), 107
 birth_ind() (*bats.ZigzagPair method*), 119
 bivariate_cover() (*in module bats*), 122
 boundary() (*bats.CellComplex method*), 84
 boundary() (*bats.CubicalComplex method*), 86
 boundary() (*bats.LightSimplicialComplex method*), 105
 boundary() (*bats.SimplicialComplex method*), 115

C

cell_ind (*class in bats*), 122
 CellComplex (*class in bats*), 84
 CellComplexDiagram (*class in bats*), 84
 CellularMap (*class in bats*), 85
 Chain() (*in module bats*), 85
 chain_preferredRepresentative()
 (*bats.ReducedF2ChainComplex*
 108
 chain_preferredRepresentative()
 (*bats.ReducedF3ChainComplex*
 109
 ChainFunctor() (*in module bats*), 85
 clear_compress_apparent_pairs()
 (*bats.F2ChainComplex method*), 90
 clear_compress_apparent_pairs()
 (*bats.F3ChainComplex method*), 94
 clearing_flag (*class in bats*), 122
 complex() (*bats.FilteredCubicalComplex method*), 98
 complex() (*bats.FilteredLightSimplicialComplex*
 method), 100
 complex() (*bats.FilteredSimplicialComplex* method),
 100
 complex() (*bats.ZigzagCubicalComplex method*), 119
 complex() (*bats.ZigzagSimplicialComplex method*), 119
 compression_flag (*class in bats*), 122
 compute_basis_flag (*class in bats*), 122
 CosineDist (*class in bats*), 86
 CoverDiagram (*class in bats*), 86
 CSCMatrix (*class in bats*), 84
 Cube() (*in module bats*), 86

CubicalComplex (*class in bats*), 86
 CubicalComplexDiagram (*class in bats*), 87
 CubicalMap() (*in module bats*), 87

D

data() (*bats.DataSet method*), 87
 DataSet (*class in bats*), 87
 death() (*bats.PersistencePair method*), 107
 death() (*bats.PersistencePair_int method*), 107
 death() (*bats.ZigzagPair method*), 119
 death_ind() (*bats.PersistencePair method*), 107
 death_ind() (*bats.PersistencePair_int method*), 107
 death_ind() (*bats.ZigzagPair method*), 119
 degree (*bats.F2DGVectorSpace property*), 92
 degree (*bats.F3DGVectorSpace property*), 96
 deletion_indices (*bats.UpdateInfoFiltration* prop-
 erty), 118
 deletion_indices (*bats.UpdateInfoLightFiltration*
 property), 118
 dim() (*bats.DataSet method*), 87
 dim() (*bats.F2ChainComplex method*), 90
 dim() (*bats.F3ChainComplex method*), 94
 dim() (*bats.PersistencePair method*), 107
 dim() (*bats.PersistencePair_int method*), 107
 dim() (*bats.ReducedFilteredF2ChainComplex* method),
 110
 dim() (*bats.ReducedFilteredF2DGVectorSpace* method),
 111
 dim() (*bats.ReducedFilteredF3ChainComplex* method),
 111
 dim() (*bats.ReducedFilteredF3DGVectorSpace* method),
 112
 dim() (*bats.ZigzagPair method*), 119
 DiscreteMorozovZigzag() (*in module bats*), 87
 DiscreteMorozovZigzagSets() (*in module bats*), 88
 divide_conquer (*class in bats*), 122
 DowkerCoverFiltration() (*in module bats*), 89
 DowkerFiltration() (*in module bats*), 89

E

edge_data() (*bats.CellComplexDiagram method*), 84
 edge_data() (*bats.CoverDiagram method*), 86
 edge_data() (*bats.CubicalComplexDiagram* method),
 87
 edge_data() (*bats.F2ChainDiagram method*), 91
 edge_data() (*bats.F2DGHomDiagram method*), 91
 edge_data() (*bats.F2DGHomDiagramAll* method), 91
 edge_data() (*bats.F2DGLinearDiagram* method), 92
 edge_data() (*bats.F2Diagram* method), 92
 edge_data() (*bats.F2HomDiagram* method), 93
 edge_data() (*bats.F2HomDiagramAll* method), 93
 edge_data() (*bats.F3ChainDiagram* method), 95
 edge_data() (*bats.F3DGHomDiagram* method), 95
 edge_data() (*bats.F3DGHomDiagramAll* method), 95

edge_data() (*bats.F3DGLinearDiagram* method), 96
edge_data() (*bats.F3Diagram* method), 96
edge_data() (*bats.F3HomDiagram* method), 97
edge_data() (*bats.F3HomDiagramAll* method), 97
edge_data() (*bats.SetDiagram* method), 115
edge_data() (*bats.SimplicialComplexDiagram* method), 116
edge_source() (*bats.CellComplexDiagram* method), 84
edge_source() (*bats.CoverDiagram* method), 86
edge_source() (*bats.CubicalComplexDiagram* method), 87
edge_source() (*bats.F2ChainDiagram* method), 91
edge_source() (*bats.F2DGHomDiagram* method), 91
edge_source() (*bats.F2DGHomDiagramAll* method), 91
edge_source() (*bats.F2DGLinearDiagram* method), 92
edge_source() (*bats.F2Diagram* method), 92
edge_source() (*bats.F2HomDiagram* method), 93
edge_source() (*bats.F2HomDiagramAll* method), 93
edge_source() (*bats.F3ChainDiagram* method), 95
edge_source() (*bats.F3DGHomDiagram* method), 95
edge_source() (*bats.F3DGHomDiagramAll* method), 95
edge_source() (*bats.F3DGLinearDiagram* method), 96
edge_source() (*bats.F3Diagram* method), 96
edge_source() (*bats.F3HomDiagram* method), 97
edge_source() (*bats.F3HomDiagramAll* method), 97
edge_source() (*bats.SetDiagram* method), 115
edge_source() (*bats.SimplicialComplexDiagram* method), 116
edge_target() (*bats.CellComplexDiagram* method), 84
edge_target() (*bats.CoverDiagram* method), 86
edge_target() (*bats.CubicalComplexDiagram* method), 87
edge_target() (*bats.F2ChainDiagram* method), 91
edge_target() (*bats.F2DGHomDiagram* method), 91
edge_target() (*bats.F2DGHomDiagramAll* method), 91
edge_target() (*bats.F2DGLinearDiagram* method), 92
edge_target() (*bats.F2Diagram* method), 92
edge_target() (*bats.F2HomDiagram* method), 93
edge_target() (*bats.F2HomDiagramAll* method), 93
edge_target() (*bats.F3ChainDiagram* method), 95
edge_target() (*bats.F3DGHomDiagram* method), 95
edge_target() (*bats.F3DGHomDiagramAll* method), 95
edge_target() (*bats.F3DGLinearDiagram* method), 96
edge_target() (*bats.F3Diagram* method), 96
edge_target() (*bats.F3HomDiagram* method), 97
edge_target() (*bats.F3HomDiagramAll* method), 97
edge_target() (*bats.SetDiagram* method), 115
edge_target() (*bats.SimplicialComplexDiagram* method), 116
EL_L_commute() (*in module bats*), 90
enclosing_radius() (*in module bats*), 122
EU_U_commute() (*in module bats*), 90
Euclidean (*class in bats*), 90
extend_zigzag_filtration() (*in module bats*), 122
extra_reduction_flag (*class in bats*), 122
extract_pairs() (*in module bats*), 123

F

F2 (*class in bats*), 90
F2Chain() (*in module bats*), 90
F2ChainComplex (*class in bats*), 90
F2ChainDiagram (*class in bats*), 90
F2ChainMap (*class in bats*), 91
F2DGHomDiagram (*class in bats*), 91
F2DGHomDiagramAll (*class in bats*), 91
F2DGLinearDiagram (*class in bats*), 92
F2DGLinearFunctor() (*in module bats*), 92
F2DGLinearMap (*class in bats*), 92
F2DGVectorSpace (*class in bats*), 92
F2Diagram (*class in bats*), 92
F2HomDiagram (*class in bats*), 93
F2HomDiagramAll (*class in bats*), 93
F2Mat (*class in bats*), 93
F2Vector (*class in bats*), 94
F3 (*class in bats*), 94
F3Chain() (*in module bats*), 94
F3ChainComplex (*class in bats*), 94
F3ChainDiagram (*class in bats*), 94
F3ChainMap (*class in bats*), 95
F3DGHomDiagram (*class in bats*), 95
F3DGHomDiagramAll (*class in bats*), 95
F3DGLinearDiagram (*class in bats*), 96
F3DGLinearFunctor() (*in module bats*), 96
F3DGLinearMap (*class in bats*), 96
F3DGVectorSpace (*class in bats*), 96
F3Diagram (*class in bats*), 96
F3HomDiagram (*class in bats*), 97
F3HomDiagramAll (*class in bats*), 97
F3Mat (*class in bats*), 97
F3Vector (*class in bats*), 98
filtered_info() (*bats.UpdateInfoFiltration* method), 118
filtered_info() (*bats.UpdateInfoLightFiltration* method), 118
FilteredCubicalComplex (*class in bats*), 98
FilteredEdge (*class in bats*), 98
FilteredF2ChainComplex (*class in bats*), 98
FilteredF2DGVectorSpace (*class in bats*), 99
FilteredF3ChainComplex (*class in bats*), 99
FilteredF3DGVectorSpace (*class in bats*), 99
FilteredLightSimplicialComplex (*class in bats*), 100
FilteredSimplicialComplex (*class in bats*), 100
Filtration() (*in module bats*), 100

find_idx() (*bats.CubicalComplex* method), 86
find_idx() (*bats.LightSimplicialComplex* method), 105
find_idx() (*bats.SimplicialComplex* method), 116
find_preferredRepresentative()
 (bats.*ReducedF2ChainComplex*
 method), 108
find_preferredRepresentative()
 (bats.*ReducedF3ChainComplex*
 method), 109
FlagComplex() (in module bats), 101
force_repel_rp() (in module bats), 123
Freudenthal() (in module bats), 101
from_hom_basis() (bats.*ReducedF2ChainComplex*
 method), 108
from_hom_basis() (bats.*ReducedF3ChainComplex*
 method), 109

G

get_cube() (*bats.CubicalComplex* method), 86
get_cubes() (*bats.CubicalComplex* method), 86
get_preferredRepresentative()
 (bats.*ReducedF2ChainComplex*
 method), 108
get_preferredRepresentative()
 (bats.*ReducedF3ChainComplex*
 method), 109
get_simplex() (*bats.LightSimplicialComplex* method),
 105
get_simplex() (*bats.SimplicialComplex* method), 116
get_simplices() (bats.*LightSimplicialComplex*
 method), 105
get_simplices() (*bats.SimplicialComplex* method),
 116
greedy_landmarks() (in module bats), 123
greedy_landmarks_hausdorff() (in module bats),
 123

H

hausdorff_landmarks() (in module bats), 124
hdim() (bats.*ReducedF2ChainComplex* method), 108
hdim() (bats.*ReducedF2DGVectorSpace* method), 108
hdim() (bats.*ReducedF3ChainComplex* method), 109
hdim() (bats.*ReducedF3DGVectorSpace* method), 109
hdim() (bats.*ReducedFilteredF2DGVectorSpace*
 method), 111
hdim() (bats.*ReducedFilteredF3DGVectorSpace*
 method), 112
Hom() (in module bats), 101

I

Identity() (in module bats), 101
IdentityMap() (in module bats), 101
InducedMap() (in module bats), 101
insertion_cols (*bats.UpdateInfo2* property), 118

insertion_indices (*bats.UpdateInfo2* property), 118
IntMat (class in bats), 102
IntVector (class in bats), 102

K

kendall_tau() (in module bats), 124
kendall_tau_dists (*bats.UpdateInfoFiltration* prop-
 erty), 118
kendall_tau_dists (*bats.UpdateInfoLightFiltration*
 property), 118

L

L1Dist (class in bats), 102
L_EL_commute() (in module bats), 103
landmark_cover() (in module bats), 124
landmark_eps_cover() (in module bats), 125
leftward (class in bats), 125
length() (*bats.PersistencePair* method), 107
length() (*bats.PersistencePair_int* method), 107
length() (*bats.ZigzagPair* method), 119
LEUP() (in module bats), 102
levelset() (*bats.ZigzagCubicalComplex* method), 119
levelset() (*bats.ZigzagSimplicialComplex* method),
 119
LightFlagComplex() (in module bats), 103
LightFreudenthal() (in module bats), 103
LightRipsComplex() (in module bats), 104
LightRipsFiltration() (in module bats), 104
LightRipsFiltration_extension() (in module
 bats), 104
LightSimplicialComplex (class in bats), 105
LInfDist (class in bats), 103
load_cubes() (*bats.CubicalComplex* method), 86
lower_star_backwards() (in module bats), 125
lower_star_filtration() (in module bats), 125
LQU() (in module bats), 103

M

Mat() (in module bats), 105
Matrix (class in bats), 105
maxdim() (*bats.CellComplex* method), 84
maxdim() (*bats.CubicalComplex* method), 87
maxdim() (*bats.F2ChainComplex* method), 90
maxdim() (*bats.F2DGVectorSpace* method), 92
maxdim() (*bats.F3ChainComplex* method), 94
maxdim() (*bats.F3DGVectorSpace* method), 96
maxdim() (*bats.FilteredCubicalComplex* method), 98
maxdim() (bats.*FilteredLightSimplicialComplex*
 method), 100
maxdim() (*bats.FilteredSimplicialComplex* method), 100
maxdim() (*bats.LightSimplicialComplex* method), 105
maxdim() (*bats.ReducedF2ChainComplex* method), 108
maxdim() (*bats.ReducedF2DGVectorSpace* method), 108
maxdim() (*bats.ReducedF3ChainComplex* method), 109

maxdim() (*bats.ReducedF3DGVectorSpace method*), 109
maxdim() (*bats.ReducedFilteredF2ChainComplex method*), 110
maxdim() (*bats.ReducedFilteredF2DGVectorSpace method*), 111
maxdim() (*bats.ReducedFilteredF3ChainComplex method*), 111
maxdim() (*bats.ReducedFilteredF3DGVectorSpace method*), 112
maxdim() (*bats.SimplicialComplex method*), 116
maxdim() (*bats.ZigzagCubicalComplex method*), 119
maxdim() (*bats.ZigzagSimplicialComplex method*), 119
mid() (*bats.PersistencePair method*), 107
mid() (*bats.PersistencePair_int method*), 107
mid() (*bats.ZigzagPair method*), 119
module
 bats, 84

N

ncells() (*bats.CellComplex method*), 84
ncells() (*bats.CubicalComplex method*), 87
ncells() (*bats.FilteredCubicalComplex method*), 98
ncells() (*bats.FilteredLightSimplicialComplex method*), 100
ncells() (*bats.FilteredSimplicialComplex method*), 100
ncells() (*bats.LightSimplicialComplex method*), 105
ncells() (*bats.SimplicialComplex method*), 116
ncells() (*bats.ZigzagCubicalComplex method*), 119
ncells() (*bats.ZigzagSimplicialComplex method*), 119
ncol() (*bats.CSCMatrix method*), 84
ncol() (*bats.F2Mat method*), 93
ncol() (*bats.F3Mat method*), 97
ncol() (*bats.IntMat method*), 102
ncol() (*bats.Matrix method*), 106
ndeletions (*bats.UpdateInfo2 property*), 118
nedge() (*bats.CellComplexDiagram method*), 84
nedge() (*bats.CoverDiagram method*), 86
nedge() (*bats.CubicalComplexDiagram method*), 87
nedge() (*bats.F2ChainDiagram method*), 91
nedge() (*bats.F2DGHomDiagram method*), 91
nedge() (*bats.F2DGHomDiagramAll method*), 91
nedge() (*bats.F2DGLinearDiagram method*), 92
nedge() (*bats.F2Diagram method*), 92
nedge() (*bats.F2HomDiagram method*), 93
nedge() (*bats.F2HomDiagramAll method*), 93
nedge() (*bats.F3ChainDiagram method*), 95
nedge() (*bats.F3DGHomDiagram method*), 95
nedge() (*bats.F3DGHomDiagramAll method*), 95
nedge() (*bats.F3DGLinearDiagram method*), 96
nedge() (*bats.F3Diagram method*), 96
nedge() (*bats.F3HomDiagram method*), 97
nedge() (*bats.F3HomDiagramAll method*), 97
nedge() (*bats.SetDiagram method*), 115
nedge() (*bats.SimplicialComplexDiagram method*), 116
Nerve() (*in module bats*), 106
NerveDiagram() (*in module bats*), 106
nnode() (*bats.CellComplexDiagram method*), 84
nnode() (*bats.CoverDiagram method*), 86
nnode() (*bats.CubicalComplexDiagram method*), 87
nnode() (*bats.F2ChainDiagram method*), 91
nnode() (*bats.F2DGHomDiagram method*), 91
nnode() (*bats.F2DGHomDiagramAll method*), 92
nnode() (*bats.F2DGLinearDiagram method*), 92
nnode() (*bats.F2Diagram method*), 92
nnode() (*bats.F2HomDiagram method*), 93
nnode() (*bats.F2HomDiagramAll method*), 93
nnode() (*bats.F3ChainDiagram method*), 95
nnode() (*bats.F3DGHomDiagram method*), 95
nnode() (*bats.F3DGHomDiagramAll method*), 95
nnode() (*bats.F3DGLinearDiagram method*), 96
nnode() (*bats.F3Diagram method*), 96
nnode() (*bats.F3HomDiagram method*), 97
nnode() (*bats.F3HomDiagramAll method*), 97
nnode() (*bats.SetDiagram method*), 115
nnode() (*bats.SimplicialComplexDiagram method*), 116
nnz() (*bats.F2Mat method*), 94
nnz() (*bats.F2Vector method*), 94
nnz() (*bats.F3Mat method*), 97
nnz() (*bats.F3Vector method*), 98
nnz() (*bats.IntMat method*), 102
nnz() (*bats.IntVector method*), 102
nnz_R() (*bats.ReducedFilteredF2ChainComplex method*), 110
nnz_R() (*bats.ReducedFilteredF3ChainComplex method*), 111
nnz_U() (*bats.ReducedFilteredF2ChainComplex method*), 110
nnz_U() (*bats.ReducedFilteredF3ChainComplex method*), 111
no_optimization_flag (*class in bats*), 125
node_data() (*bats.CellComplexDiagram method*), 84
node_data() (*bats.CoverDiagram method*), 86
node_data() (*bats.CubicalComplexDiagram method*), 87
node_data() (*bats.F2ChainDiagram method*), 91
node_data() (*bats.F2DGHomDiagram method*), 91
node_data() (*bats.F2DGHomDiagramAll method*), 92
node_data() (*bats.F2DGLinearDiagram method*), 92
node_data() (*bats.F2Diagram method*), 92
node_data() (*bats.F2HomDiagram method*), 93
node_data() (*bats.F2HomDiagramAll method*), 93
node_data() (*bats.F3ChainDiagram method*), 95
node_data() (*bats.F3DGHomDiagram method*), 95
node_data() (*bats.F3DGHomDiagramAll method*), 95
node_data() (*bats.F3DGLinearDiagram method*), 96
node_data() (*bats.F3Diagram method*), 96
node_data() (*bats.F3HomDiagram method*), 97
node_data() (*bats.F3HomDiagramAll method*), 97

node_data() (*bats.SetDiagram method*), 115
node_data() (*bats.SimplicialComplexDiagram method*), 116
nrow() (*bats.CSCMatrix method*), 84
nrow() (*bats.F2Mat method*), 94
nrow() (*bats.F3Mat method*), 98
nrow() (*bats.IntMat method*), 102
nrow() (*bats.Matrix method*), 106
nzinds() (*bats.F2Vector method*), 94
nzinds() (*bats.F3Vector method*), 98
nzinds() (*bats.IntVector method*), 102
nzs() (*bats.F2Vector method*), 94
nzs() (*bats.F3Vector method*), 98
nzs() (*bats.IntVector method*), 102
nzvals() (*bats.F2Vector method*), 94
nzvals() (*bats.F3Vector method*), 98
nzvals() (*bats.IntVector method*), 102

O

OscillatingRipsZigzagSets() (*in module bats*), 106

P

perm (*bats.UpdateInfo2 property*), 118
perm() (*bats.FilteredF2ChainComplex method*), 98
perm() (*bats.FilteredF2DGVectorSpace method*), 99
perm() (*bats.FilteredF3ChainComplex method*), 99
perm() (*bats.FilteredF3DGVectorSpace method*), 99
perm() (*bats.ReducedFilteredF2ChainComplex method*), 110
perm() (*bats.ReducedFilteredF3ChainComplex method*), 111
permutations (*bats.UpdateInfoFiltration property*), 118
permutations (*bats.UpdateInfoLightFiltration property*), 118
permute() (*bats.F2Vector method*), 94
permute() (*bats.F3Vector method*), 98
permute() (*bats.IntVector method*), 102
permute_cols() (*bats.F2Mat method*), 94
permute_cols() (*bats.F3Mat method*), 98
permute_cols() (*bats.IntMat method*), 102
permute_rows() (*bats.F2Mat method*), 94
permute_rows() (*bats.F3Mat method*), 98
permute_rows() (*bats.IntMat method*), 102
persistence_barcode() (*in module bats*), 125
persistence_diagram() (*in module bats*), 126
persistence_pairs()
 (*bats.ReducedFilteredF2ChainComplex method*), 110
persistence_pairs()
 (*bats.ReducedFilteredF2DGVectorSpace method*), 111
persistence_pairs()
 (*bats.ReducedFilteredF3ChainComplex method*), 111

persistence_pairs()
 (*bats.ReducedFilteredF3DGVectorSpace method*), 112
persistence_pairs_vec()
 (*bats.ReducedFilteredF2ChainComplex method*), 110
persistence_pairs_vec()
 (*bats.ReducedFilteredF2DGVectorSpace method*), 111
persistence_pairs_vec()
 (*bats.ReducedFilteredF3ChainComplex method*), 111
persistence_pairs_vec()
 (*bats.ReducedFilteredF3DGVectorSpace method*), 112
PersistencePair (*class in bats*), 107
PersistencePair_int (*class in bats*), 107
PLEU() (*in module bats*), 106
print() (*bats.CSCMatrix method*), 84
print() (*bats.Matrix method*), 106
PUEL() (*in module bats*), 107

R

R() (*bats.ReducedF2ChainComplex method*), 108
R() (*bats.ReducedF3ChainComplex method*), 109
random_landmarks() (*in module bats*), 126
reduce() (*in module bats*), 126
reduce_matrix() (*in module bats*), 135
reduced_complex() (*bats.ReducedFilteredF2ChainComplex method*), 110
reduced_complex() (*bats.ReducedFilteredF3ChainComplex method*), 111
ReducedChainComplex() (*in module bats*), 107
ReducedF2ChainComplex (*class in bats*), 108
ReducedF2DGVectorSpace (*class in bats*), 108
ReducedF3ChainComplex (*class in bats*), 108
ReducedF3DGVectorSpace (*class in bats*), 109
ReducedFilteredChainComplex() (*in module bats*), 109
ReducedFilteredF2ChainComplex (*class in bats*), 110
ReducedFilteredF2DGVectorSpace (*class in bats*), 110
ReducedFilteredF3ChainComplex (*class in bats*), 111
ReducedFilteredF3DGVectorSpace (*class in bats*), 112
representative() (*bats.ReducedFilteredF2ChainComplex method*), 110
representative() (*bats.ReducedFilteredF3ChainComplex method*), 112
reverse_for_cohomology() (*bats.UpdateInfo2 method*), 118
rightward (*class in bats*), 135
Rips() (*in module bats*), 113
rips_union_find_pairs() (*in module bats*), 135

RipsComplex() (*in module bats*), 113
RipsCoverFiltration() (*in module bats*), 113
RipsFiltration() (*in module bats*), 114
RipsFiltration_extension() (*in module bats*), 114
RipsHom() (*in module bats*), 115
RPAngleDist (*class in bats*), 107
RPCosineDist (*class in bats*), 107

S

sample_sphere() (*in module bats*), 135
set_edge() (*bats.CellComplexDiagram method*), 84
set_edge() (*bats.CoverDiagram method*), 86
set_edge() (*bats.CubicalComplexDiagram method*), 87
set_edge() (*bats.F2ChainDiagram method*), 91
set_edge() (*bats.F2DGHomDiagram method*), 91
set_edge() (*bats.F2DGHomDiagramAll method*), 92
set_edge() (*bats.F2DGLinearDiagram method*), 92
set_edge() (*bats.F2Diagram method*), 93
set_edge() (*bats.F2HomDiagram method*), 93
set_edge() (*bats.F2HomDiagramAll method*), 93
set_edge() (*bats.F3ChainDiagram method*), 95
set_edge() (*bats.F3DGHomDiagram method*), 95
set_edge() (*bats.F3DGHomDiagramAll method*), 96
set_edge() (*bats.F3DGLinearDiagram method*), 96
set_edge() (*bats.F3Diagram method*), 96
set_edge() (*bats.F3HomDiagram method*), 97
set_edge() (*bats.F3HomDiagramAll method*), 97
set_edge() (*bats.SetDiagram method*), 115
set_edge() (*bats.SimplicialComplexDiagram method*), 116
set_node() (*bats.CellComplexDiagram method*), 85
set_node() (*bats.CoverDiagram method*), 86
set_node() (*bats.CubicalComplexDiagram method*), 87
set_node() (*bats.F2ChainDiagram method*), 91
set_node() (*bats.F2DGHomDiagram method*), 91
set_node() (*bats.F2DGHomDiagramAll method*), 92
set_node() (*bats.F2DGLinearDiagram method*), 92
set_node() (*bats.F2Diagram method*), 93
set_node() (*bats.F2HomDiagram method*), 93
set_node() (*bats.F2HomDiagramAll method*), 93
set_node() (*bats.F3ChainDiagram method*), 95
set_node() (*bats.F3DGHomDiagram method*), 95
set_node() (*bats.F3DGHomDiagramAll method*), 96
set_node() (*bats.F3DGLinearDiagram method*), 96
set_node() (*bats.F3Diagram method*), 97
set_node() (*bats.F3HomDiagram method*), 97
set_node() (*bats.F3HomDiagramAll method*), 97
set_node() (*bats.SetDiagram method*), 115
set_node() (*bats.SimplicialComplexDiagram method*), 116

SetDiagram (*class in bats*), 115
SierpinskiDiagram() (*in module bats*), 115
SimplicialComplex (*class in bats*), 115
SimplicialComplexDiagram (*class in bats*), 116

SimplicialMap() (*in module bats*), 116
size() (*bats.DataSet method*), 87
skeleton() (*bats.CubicalComplex method*), 87
sort() (*bats.F2Vector method*), 94
sort() (*bats.F3Vector method*), 98
sort() (*bats.IntVector method*), 102
standard_reduction_flag (*class in bats*), 135
StrictRipsCoverFiltration() (*in module bats*), 117
sublevelset() (*bats.FilteredCubicalComplex method*), 98
sublevelset() (*bats.FilteredLightSimplicialComplex method*), 100
sublevelset() (*bats.FilteredSimplicialComplex method*), 100

T

T() (*bats.F2Mat method*), 93
T() (*bats.F3Mat method*), 97
T() (*bats.IntMat method*), 102
to_hom_basis() (*bats.ReducedF2ChainComplex method*), 108
to_hom_basis() (*bats.ReducedF3ChainComplex method*), 109
to_int() (*bats.F2 method*), 90
to_int() (*bats.F3 method*), 94
tolist() (*bats.F2Mat method*), 94
tolist() (*bats.F3Mat method*), 98
tolist() (*bats.IntMat method*), 102
TriangulatedProduct() (*in module bats*), 117

U

U() (*bats.ReducedF2ChainComplex method*), 108
U() (*bats.ReducedF3ChainComplex method*), 109
U_EU_commute() (*in module bats*), 117
UEL_P() (*in module bats*), 117
union_find_pairs() (*in module bats*), 135
update() (*bats.ReducedFilteredF2DGVectorSpace method*), 111
update() (*bats.ReducedFilteredF3DGVectorSpace method*), 112
update_filtration() (*bats.FilteredCubicalComplex method*), 98
update_filtration() (*bats.FilteredF2ChainComplex method*), 99
update_filtration() (*bats.FilteredF2DGVectorSpace method*), 99
update_filtration() (*bats.FilteredF3ChainComplex method*), 99
update_filtration() (*bats.FilteredF3DGVectorSpace method*), 99
update_filtration() (*bats.FilteredLightSimplicialComplex method*),

100
update_filtration()
 (*bats.FilteredSimplicialComplex* *method*),
 100
update_filtration()
 (*bats.ReducedFilteredF2ChainComplex*
 method), 110
update_filtration()
 (*bats.ReducedFilteredF2DGVectorSpace*
 method), 111
update_filtration()
 (*bats.ReducedFilteredF3ChainComplex*
 method), 112
update_filtration()
 (*bats.ReducedFilteredF3DGVectorSpace*
 method), 112
update_filtration_general()
 (*bats.FilteredF2ChainComplex* *method*),
 99
update_filtration_general()
 (*bats.FilteredF3ChainComplex* *method*),
 99
update_filtration_general()
 (*bats.ReducedFilteredF2ChainComplex*
 method), 110
update_filtration_general()
 (*bats.ReducedFilteredF2DGVectorSpace*
 method), 111
update_filtration_general()
 (*bats.ReducedFilteredF3ChainComplex*
 method), 112
update_filtration_general()
 (*bats.ReducedFilteredF3DGVectorSpace*
 method), 112
UpdateInfo2 (*class in bats*), 118
UpdateInfoFiltration (*class in bats*), 118
UpdateInfoLightFiltration (*class in bats*), 118

V

val() (*bats.FilteredF2ChainComplex method*), 99
val() (*bats.FilteredF2DGVectorSpace method*), 99
val() (*bats.FilteredF3ChainComplex method*), 99
val() (*bats.FilteredF3DGVectorSpace method*), 100
val() (*bats.ReducedFilteredF2ChainComplex method*),
 110
val() (*bats.ReducedFilteredF3ChainComplex method*),
 112
vals() (*bats.FilteredCubicalComplex method*), 98
vals() (*bats.FilteredLightSimplicialComplex method*),
 100
vals() (*bats.FilteredSimplicialComplex method*), 100
vals() (*bats.ZigzagCubicalComplex method*), 119
vals() (*bats.ZigzagSimplicialComplex method*), 120

W

WitnessFiltration() (*in module bats*), 118

Z

zigzag_levelsets() (*in module bats*), 135
zigzag_toplex() (*in module bats*), 135
ZigzagBarcode() (*in module bats*), 118
ZigzagCubicalComplex (*class in bats*), 119
ZigzagPair (*class in bats*), 119
ZigzagSetDiagram() (*in module bats*), 119
ZigzagSimplicialComplex (*class in bats*), 119