PerMaViss

Release v0.0.2

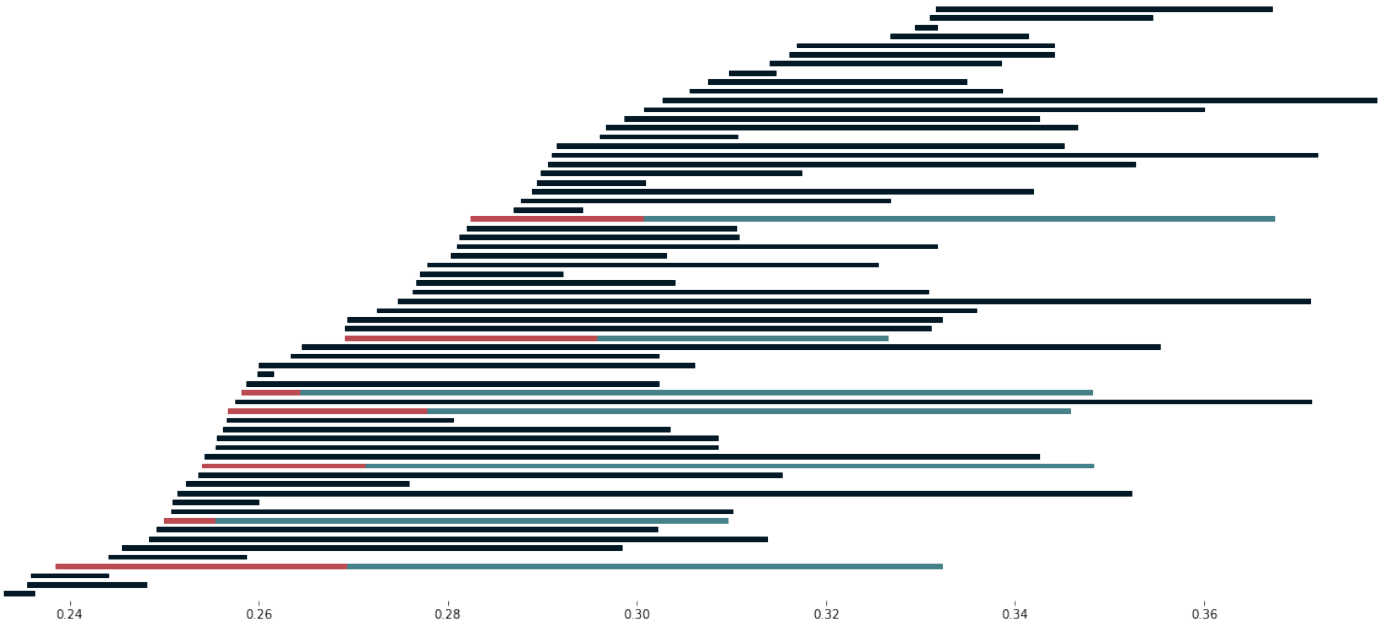
Feb 08, 2021

Contents

Welcome to PerMaViss! This is a Python3 implementation of the Persistence Mayer Vietoris spectral sequence. For a mathematical description of the procedure, see Distributing Persistent Homology via Spectral Sequences.

In a nutshell, this library is intended to be a *proof of concept* for persistence homology parallelization. That is, one can divide a point cloud into covering regions, compute persistent homology on each part, and combine all results to obtain the global persistent homology again. This is done by means of the Persistence Mayer Vietoris spectral sequence. Here we present two examples, the torus and random point clouds in three dimensions. Both of these are divided into *8* mutually overlapping regions, and the spectral sequence is computed with respect to this cover. The resulting barcodes coincide with that which would be obtained by computing persistent homology directly.

This implementation is more of a *prototype* than a finished program. As such, it still needs to be optimized. Even there might be some bugs, since there are still not enough tests for all the functionalities of the spectral sequence (if you detect any, please get in touch). Also, it would be great to have more examples for different covers. Additionally, it would be interesting to also have an implementation for cubical, alpha, and other complexes. Any collaboration or suggestion will be welcome!



CHAPTER 1

About

1.1 Quickstart

The main function which we use is *permaviss.spectral_sequence.MV_spectral_seq.create_MV_ss()*. We start by taking 100 points in a noisy circle of radius 1

```
>>> from permaviss.sample_point_clouds.examples import random_circle
>>> point_cloud = random_circle(100, 1, epsilon=0.2)
```

Now we set the parameters for spectral sequence. These are

- a prime number *p*,
- the maximum dimension of the Rips Complex *max_dim*,
- the maximum radius of filtration *max_r*,
- the number of divisions *max_div* along the maximum range in *point_cloud*,
- and the *overlap* between different covering regions.

In our case, we set the parameters to cover our circle with 9 covering regions. Notice that in order for the algorithm to give the correct result we need *overlap* > *max_r*.

```
>>> p = 3
>>> max_dim = 3
>>> max_r = 0.2
>>> max_div = 3
>>> overlap = max_r * 1.01
```

Then, we compute the spectral sequence, notice that the method prints the successive page ranks.

```
>>> from permaviss.spectral_sequence.MV_spectral_seq import create_MV_ss
>>> MV_ss = create_MV_ss(point_cloud, max_r, max_dim, max_div, overlap, p)
PAGE: 1
[[  0   0   0   0   0]
```

(continues on next page)

(continued from previous page)

```
 [  7   0   0   0   0]
 [133  33   0   0   0]]
PAGE: 2
[[  0   0   0   0   0]
 [  7   0   0   0   0]
 [100   0   0   0   0]]
PAGE: 3
[[  0   0   0   0   0]
 [  7   0   0   0   0]
 [100   0   0   0   0]]
PAGE: 4
[[  0   0   0   0   0]
 [  7   0   0   0   0]
 [100   0   0   0   0]]
```

We can inspect the obtained barcodes on the 1st dimension.

```
>>> MV_ss.persistent_homology[1].barcode
array([[ 0.08218822,  0.09287436],
       [ 0.0874977 ,  0.11781674],
       [ 0.10459203,  0.12520266],
       [ 0.14999507,  0.18220508],
       [ 0.15036084,  0.15760192],
       [ 0.16260913,  0.1695936 ],
       [ 0.16462541,  0.16942819]])
```

Notice that in this case, there was no need to solve the extension problem. See the examples section for nontrivial extensions.

1.2 DISCLAIMER

The main purpose of this library is to explore how the Persistent Mayer Vietoris spectral sequence can be used for computing persistent homology.

This does not pretend to be an optimal library. Thus, this is slower than most other persistent homology computations.

This library is still on development and is still highly undertested. If you notice any issues, please email TorrasCasasA@cardiff.ac.uk

This library is published under the standard MIT licence. Thus: THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.3 How to cite

Álvaro Torras Casas. (2020, January 20). PerMaViss: Persistence Mayer Vietoris spectral sequence (Version v0.0.2). Zenodo. http://doi.org/10.5281/zenodo.3613870

1.4 Reference

This module is written using the algorithm in Distributing Persistent Homology via Spectral Sequences.

CHAPTER 2

Install

2.1 Dependencies

PerMaViss requires:

- Python3
- NumPy
- Scipy

Optional for examples and notebooks:

- Matplotlib
- mpl_toolkits

2.2 Installation

Permaviss is built on Python 3, and relies only on NumPy and Scipy.

Additionally, Matplotlib and mpl_toolkits are used for the tutorials.

To install using `pip3`:

```
$ pip3 install permaviss
```

If you prefer to install from source, clone from GitHub repository:

```
$ git clone https://github.com/atorras1618/PerMaViss
$ cd PerMaViss
$ pip3 install -e .
```

CHAPTER 3

Usage and examples

In these tutorials we will see how data can be broken down into pieces and persistent homology can still be computed through the Mayer-Vietoris procedure. Check the notebooks if you prefer to work through these.

3.1 Torus

We compute persistent homology through two methods. First we compute persistent homology using the standard method. Then we compute this again using the Persistence Mayer Vietoris spectral sequence. At the end we compare both results and confirm that they coincide.

First we do all the relevant imports for this example

```
>>> import scipy.spatial.distance as dist
>>> from permaviss.sample_point_clouds.examples import torus3D, take_sample
>>> from permaviss.simplicial_complexes.vietoris_rips import vietoris_rips
>>> from permaviss.simplicial_complexes.differentials import complex_differentials
>>> from permaviss.spectral_sequence.MV_spectral_seq import create_MV_ss
```

We start by taking a sample of 1300 points from a torus of section radius 1 and radius from centre to section centre 3. Since this sample is too big, we take a subsample of 150 points by using a minmax method. We store it in *point_cloud*.

```
>>> X = torus_3D(1300,3)
>>> point_cloud = take_sample(X,150)
```

Next we compute the distance matrix of *point_cloud*. Also we compute the Vietoris Rips complex of *point_cloud* up to a maximum dimension *3* and maximum filtration radius *1.6*.

```
>>> Dist = dist.squareform(dist.pdist(point_cloud))
>>> max_r = 1.6
>>> max_dim = 3
>>> C, R = vietoris_rips(Dist, max_r, max_dim)
```

Afterwards, we compute the complex differentials using arithmetic mod *p*, a prime number. Then we get the persistent homology of *point_cloud* with the specified parameters. We store the result in *PerHom*. Additionally, we inspect the second persistent homology group barcodes (notice that these might be empty).

```
>>> p = 5
>>> Diff = complex_differentials(C, p)
>>> PerHom, _, _ = persistent_homology(Diff, R, max_r, p)
>>> print(PerHom[2].barcode)
[[ 1.36770353  1.38090695]
 [ 1.51515438  1.6       ]]
```

Now we will proceed to compute again persistent homology of *point_cloud* using the Persistence Mayer-Vietoris spectral sequence instead. For this task we take the same parameters *max_r*, *max_dim* and *p* as before. We set *max_div*, which is the number of divisions along the coordinate with greater range in *point_cloud*, to be 2. This will indicate **create_MV_ss** to cover *point_cloud* by 8 hypercubes. Also, we set the *overlap* between neighbouring regions to be slightly greater than *max_r*. The method **create_MV_ss** prints the ranks of the computed pages and returns a spectral sequence object which we store in *MV_ss*.

```
>>> max_div = 2
>>> overlap = max_r*1.01
>>> MV_ss = create_MV_ss(point_cloud, max_r, max_dim, max_div, overlap, p)
PAGE: 1
[[  1   0   0   0   0   0   0   0   0]
 [ 98  14   0   0   0   0   0   0   0]
 [217  56   0   0   0   0   0   0   0]]
PAGE: 2
[[  1   0   0   0   0   0   0   0   0]
 [ 84   1   0   0   0   0   0   0   0]
 [161   5   0   0   0   0   0   0   0]]
PAGE: 3
[[  1   0   0   0   0   0   0   0   0]
 [ 84   1   0   0   0   0   0   0   0]
 [161   5   0   0   0   0   0   0   0]]
PAGE: 4
[[  1   0   0   0   0   0   0   0   0]
 [ 84   1   0   0   0   0   0   0   0]
 [161   5   0   0   0   0   0   0   0]]
```

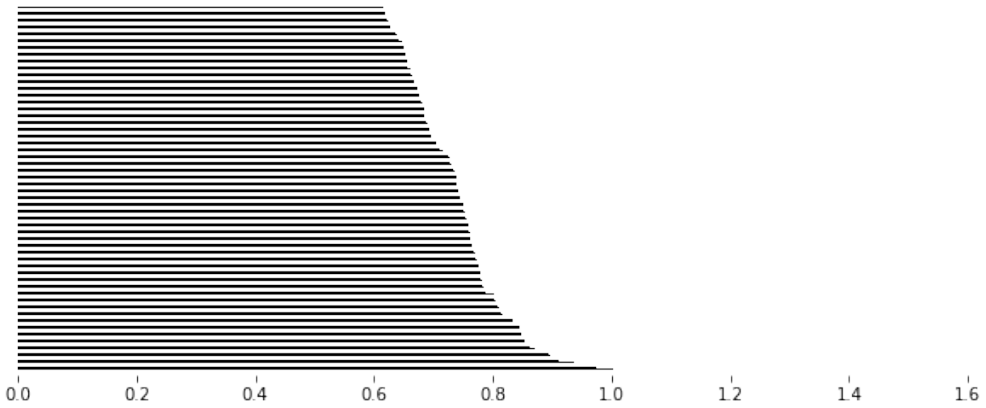
Now, we compare the computed persistent homology barcodes by both methods. Unless an *AssertError* comes up, this means that the computed barcodes **coincide**. Also, we plot the relevant barcodes.

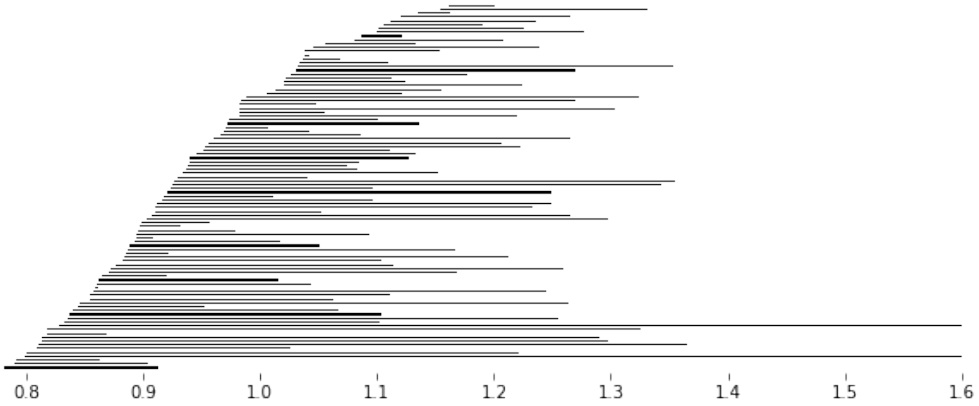
```
>>> for it, PH in enumerate(MV_ss.persistent_homology):
>>>     # Check that computed barcodes coincide
>>>     assert np.array_equal(PH.barcode, PerHom[it].barcode)
>>>     # Set plotting parameters
>>>     min_r = min(PH.barcode[:,0])
>>>     step = max_r/PH.dim
>>>     width = step / 2.
>>>     fig, ax = plt.subplots(figsize = (10,4))
>>>     ax = plt.axes(frameon=False)
>>>     y_coord = 0
>>>     # Plot barcodes
>>>     for k, b in enumerate(PH.barcode):
>>>         ax.fill([b[0],b[1],b[1],b[0]],[y_coord,y_coord,y_coord+width,y_
↪coord+width],'black',label='H0')
>>>         y_coord += step
>>>
```

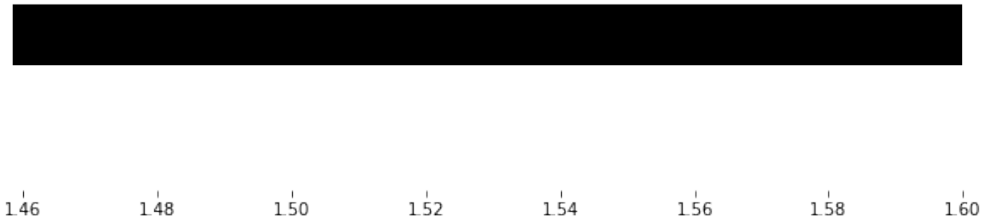
(continues on next page)

(continued from previous page)

```
>>>
>>>     # Show figure
>>>     ax.axes.get_yaxis().set_visible(False)
>>>     ax.set_xlim([min_r,max_r])
>>>     ax.set_ylim([-step, max_r + step])
>>>     plt.savefig("barcode_r{}.png".format(it))
>>>     plt.show()
```



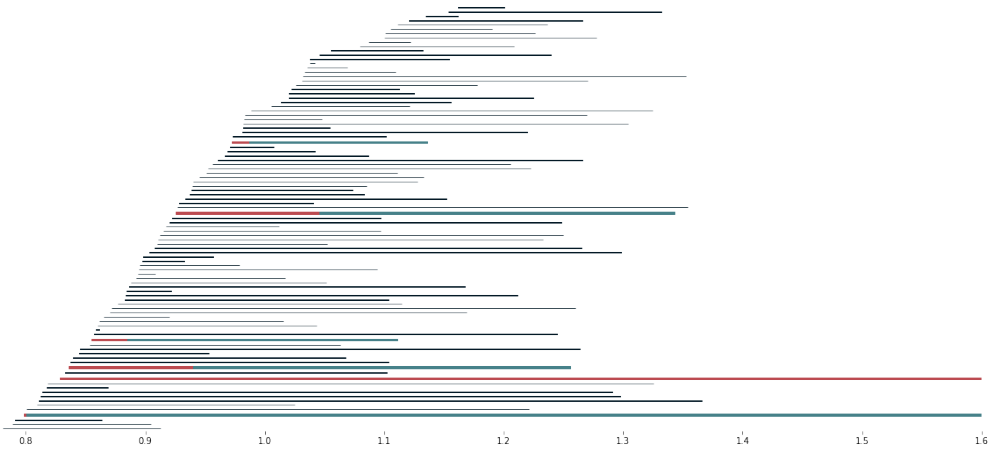




Here we look at the extension information on one dimensional persistence classes. For this we exploit the extra information stored in *MV_ss*. What we do is plot the one dimensional barcodes, highlighting those bars from the `(0,1)` position in the infinity page in red. Also, we highlight in blue when these bars are extended by a bar in the

`(1,0)` position on the infinity page. All the black bars are only coming from classes in the `(1,0)` position on the infinity page.

```
>>> PH = MV_ss.persistent_homology
>>> start_rad = min(PH[1].barcode[:,0])
>>> end_rad = max(PH[1].barcode[:,1])
>>> persistence = end_rad - start_rad
>>> fig, ax = plt.subplots(figsize = (20,9))
>>> ax = plt.axes(frameon=False)
>>> # ax = plt.axes()
>>> step = (persistence /2) / PH[1].dim
>>> width = (step/6.)
>>> y_coord = 0
>>> for b in PH[1].barcode:
>>>     if b[0] not in MV_ss.Hom[2][1][0].barcode[:,0]:
>>>         ax.fill([b[0],b[1],b[1],b[0]],[y_coord,y_coord,y_coord+width,y_
↪coord+width],c="#031926", edgecolor='none')
>>>     else:
>>>         index = np.argmax(b[0] <= MV_ss.Hom[2][1][0].barcode[:,0])
>>>         midpoint = MV_ss.Hom[2][1][0].barcode[index,1]
>>>         ax.fill([b[0], midpoint, midpoint, b[0]],[y_coord,y_coord,y_coord+step,y_
↪coord+step],c="#bc4b51", edgecolor='none')
>>>         ax.fill([midpoint, b[1], b[1], midpoint],[y_coord,y_coord,y_coord+step,y_
↪coord+step],c='#468189', edgecolor='none')
>>>         y_coord = y_coord + step
>>>
>>>     y_coord += 2 * step
>>>
>>> # Show figure
>>> ax.axes.get_yaxis().set_visible(False)
>>> ax.set_xlim([start_rad,end_rad])
>>> ax.set_ylim([-step, y_coord + step])
>>> plt.show()
```



We can also study the representatives associated to these barcodes. In the following, we go through all possible extended bars. In red, we plot representatives of a class from `(1,0)`. These are extended to representatives from `(0,1)` that we plot in dashed yellow lines.

```
>>> extension_indices = [i for i, x in enumerate(
>>>     np.any(MV_ss.extensions[1][0][1], axis=0)) if x]
>>>
```

(continues on next page)

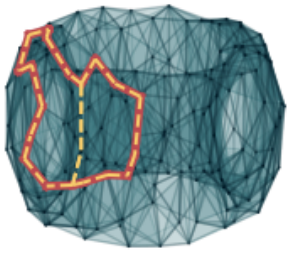
(continued from previous page)

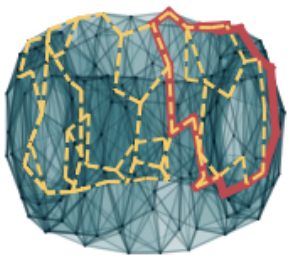
```
>>> for idx_cycle in extension_indices:
>>>     # initialize plot
>>>     fig = plt.figure()
>>>     ax = fig.add_subplot(111, projection='3d')
>>>     # plot simplicial complex
>>>     # plot edges
>>>     for edge in C[1]:
>>>         start_point = point_cloud[edge[0]]
>>>         end_point = point_cloud[edge[1]]
>>>         ax.plot([start_point[0], end_point[0]],
>>>                 [start_point[1], end_point[1]],
>>>                 [start_point[2], end_point[2]],
>>>                 color="#031926",
>>>                 alpha=0.5*((max_r-Dist[edge[0],edge[1]])/max_r))
>>>
>>>     # plot vertices
>>>     poly3d = []
>>>     for face in C[2]:
>>>         triangles = []
>>>         for pt in face:
>>>             triangles.append(point_cloud[pt])
>>>
>>>         poly3d.append(triangles)
>>>
>>>     ax.add_collection3d(Poly3DCollection(poly3d, linewidths=1,
>>>                                          alpha=0.1, color='#468189'))
>>>     # plot red cycle, that is, a cycle in (1,0)
>>>     cycle = MV_ss.tot_complex_reps[1][0][1][idx_cycle]
>>>     for cover_idx in iter(cycle):
>>>         if len(cycle[cover_idx]) > 0 and np.any(cycle[cover_idx]):
>>>             for l in np.nonzero(cycle[cover_idx])[0]:
>>>                 start_pt = MV_ss.nerve_point_cloud[0][cover_idx][
>>>                     MV_ss.subcomplexes[0][cover_idx][1][int(l)][0]]
>>>                 end_pt = MV_ss.nerve_point_cloud[0][cover_idx][
>>>                     MV_ss.subcomplexes[0][cover_idx][1][int(l)][1]]
>>>                 plt.plot(
>>>                     [start_pt[0], end_pt[0]], [start_pt[1],end_pt[1]],
>>>                     [start_pt[2], end_pt[2]], c="#bc4b51", linewidth=5)
>>>         # end if
>>>     # end for
>>>     # Plot yellow cycles from (0,1) that extend the red cycle
>>>     for idx, cycle in enumerate(MV_ss.tot_complex_reps[0][1][0]):
>>>         # if it extends the cycle in (1,0)
>>>         if MV_ss.extensions[1][0][1][idx, idx_cycle] != 0:
>>>             for cover_idx in iter(cycle):
>>>                 if len(cycle[cover_idx]) > 0 and np.any(
>>>                         cycle[cover_idx]):
>>>                     for l in np.nonzero(cycle[cover_idx])[0]:
>>>                         start_pt = MV_ss.nerve_point_cloud[0][
>>>                             cover_idx][MV_ss.subcomplexes[0][
>>>                                 cover_idx][1][int(l)][0]]
>>>                         end_pt = MV_ss.nerve_point_cloud[0][cover_idx][
>>>                             MV_ss.subcomplexes[0][cover_idx][
>>>                                 1][int(l)][1]]
>>>                         plt.plot([start_pt[0], end_pt[0]],
>>>                                  [start_pt[1],end_pt[1]],
>>>                                  [start_pt[2], end_pt[2]],
```

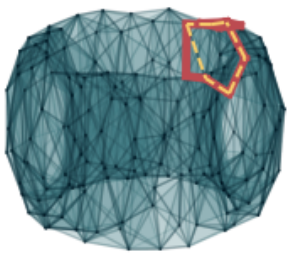
(continues on next page)

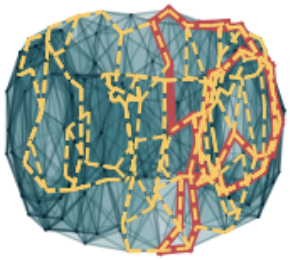
(continued from previous page)

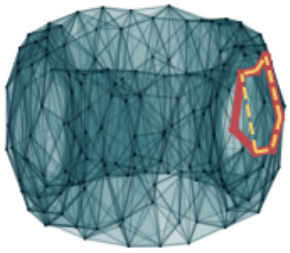
```
>>>                                     '--', c='#f7cd6c', linewidth=2)
>>>                 # end if
>>>             # end for
>>>         # end if
>>>     # end for
>>>     # Then we show the figure
>>>     ax.grid(False)
>>>     ax.set_axis_off()
>>>     plt.show()
>>>     plt.close(fig)
```











3.2 Random 3D point cloud

We can repeat the same procedure as with the torus, but with random 3D point clouds. First we do all the relevant imports for this example

```
>>> import scipy.spatial.distance as dist
>>> from permaviss.sample_point_clouds.examples import random_cube, take_sample
>>> from permaviss.simplicial_complexes.vietoris_rips import vietoris_rips
>>> from permaviss.simplicial_complexes.differentials import complex_differentials
>>> from permaviss.spectral_sequence.MV_spectral_seq import create_MV_ss
```

We start by taking a sample of 1300 points from a torus of section radius 1 and radius from center to section center 3. Since this sample is too big, we take a subsample of 91 points by using a minmax method. We store it in *point_cloud*.

```
>>> X = random_cube(1300,3)
>>> point_cloud = take_sample(X,91)
```

Next we compute the distance matrix of *point_cloud*. Also we compute the Vietoris Rips complex of *point_cloud* up to a maximum dimension *3* and maximum filtration radius *1.6*.

```
>>> Dist = dist.squareform(dist.pdist(point_cloud))
>>> max_r = 0.39
>>> max_dim = 4
>>> C, R = vietoris_rips(Dist, max_r, max_dim)
```

Afterwards, we compute the complex differentials using arithmetic mod p, a prime number. Then we get the persistent homology of *point_cloud* with the specified parameters. We store the result in *PerHom*.

```
>>> p = 5
>>> Diff = complex_differentials(C, p)
>>> PerHom, _, _ = persistent_homology(Diff, R, max_r, p)
```

Now we will proceed to compute again persistent homology of *point_cloud* using the Persistence Mayer-Vietoris spectral sequence instead. For this task we take the same parameters *max_r*, *max_dim* and *p* as before. We set *max_div*, which is the number of divisions along the coordinate with greater range in *point_cloud*, to be 2. This will indicate **create_MV_ss** to cover *point_cloud* by 8 hypercubes. Also, we set the *overlap* between neighbouring regions to be slightly greater than *max_r*. The method **create_MV_ss** prints the ranks of the computed pages and returns a spectral sequence object which we store in *MV_ss*.

```
>>> max_div = 2
>>> overlap = max_r*1.01
>>> MV_ss = create_MV_ss(point_cloud, max_r, max_dim, max_div, overlap, p)
    PAGE: 1
    [[  0   0   0   0   0   0   0   0   0]
     [ 11   1   0   0   0   0   0   0   0]
     [ 91  25   0   0   0   0   0   0   0]
     [208 231 236 227 168  84  24   3   0]]
    PAGE: 2
    [[ 0  0  0  0  0  0  0  0  0]
     [10  0  0  0  0  0  0  0  0]
     [67  3  0  0  0  0  0  0  0]
     [91  7  2  0  0  0  0  0  0]]
    PAGE: 3
    [[ 0  0  0  0  0  0  0  0  0]
     [10  0  0  0  0  0  0  0  0]
     [65  3  0  0  0  0  0  0  0]
     [91  7  1  0  0  0  0  0  0]]
    PAGE: 4
    [[ 0  0  0  0  0  0  0  0  0]
     [10  0  0  0  0  0  0  0  0]
     [65  3  0  0  0  0  0  0  0]
     [91  7  1  0  0  0  0  0  0]]
    PAGE: 5
    [[ 0  0  0  0  0  0  0  0  0]
     [10  0  0  0  0  0  0  0  0]
     [65  3  0  0  0  0  0  0  0]
     [91  7  1  0  0  0  0  0  0]]
```

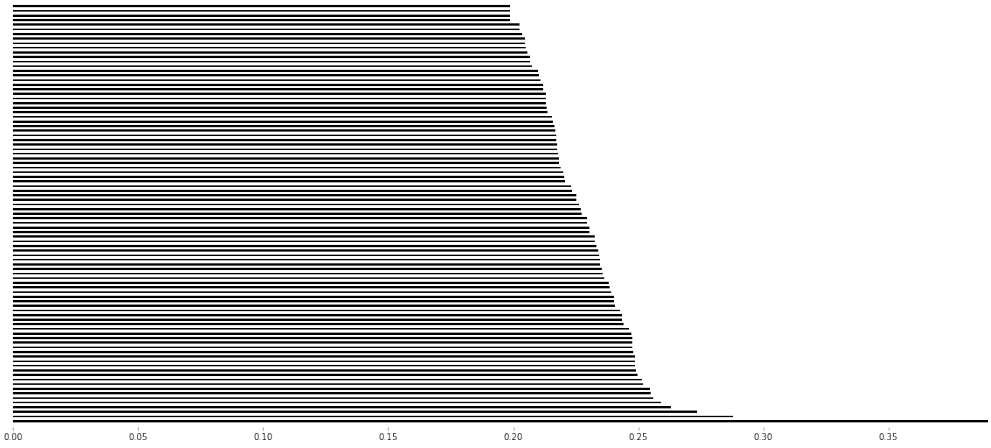
In particular, notice that in this example the second page differential is nonzero. Now, we compare the computed persistent homology barcodes by both methods. Unless an *AssertError* comes up, this means that the computed barcodes **coincide**. Also, we plot the relevant barcodes.

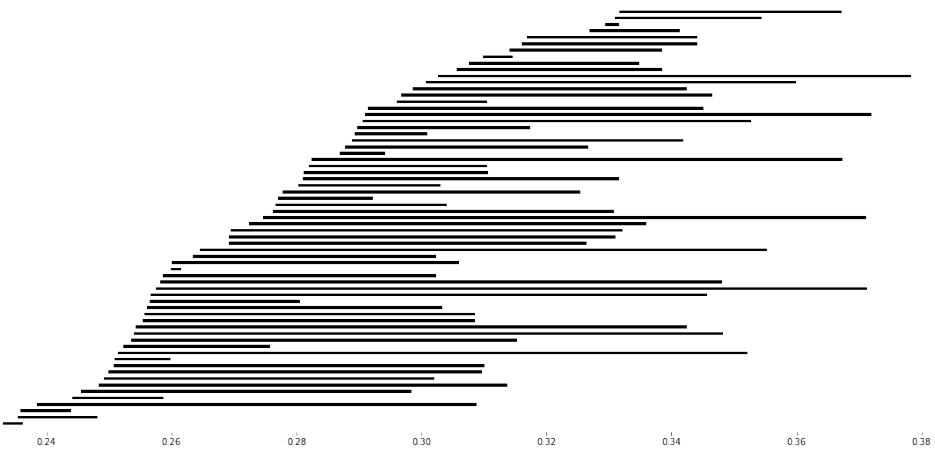
```
>>> for it, PH in enumerate(MV_ss.persistent_homology):
>>>     # Check that computed barcodes coincide
>>>     assert np.array_equal(PH.barcode, PerHom[it].barcode)
>>>     # Set plotting parameters
>>>     min_r = min(PH.barcode[:,0])
>>>     step = max_r/PH.dim
>>>     width = step / 2.
>>>     fig, ax = plt.subplots(figsize = (10,4))
>>>     ax = plt.axes(frameon=False)
>>>     y_coord = 0
>>>     # Plot barcodes
>>>     for k, b in enumerate(PH.barcode):
>>>         ax.fill([b[0],b[1],b[1],b[0]],[y_coord,y_coord,y_coord+width,y_
↪coord+width],'black',label='H0')
```

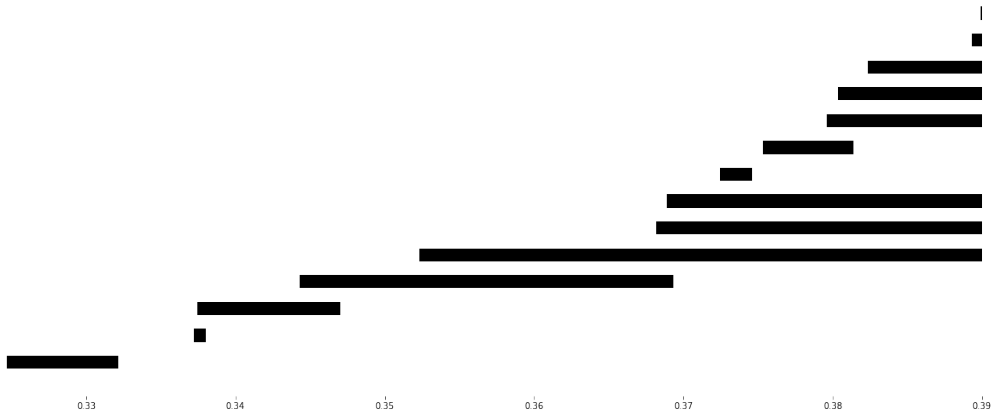
(continues on next page)

(continued from previous page)

```
>>>         y_coord += step
>>>
>>>
>>>     # Show figure
>>>     ax.axes.get_yaxis().set_visible(False)
>>>     ax.set_xlim([min_r,max_r])
>>>     ax.set_ylim([-step, max_r + step])
>>>     plt.savefig("barcode_r{}.png".format(it))
>>>     plt.show()
```







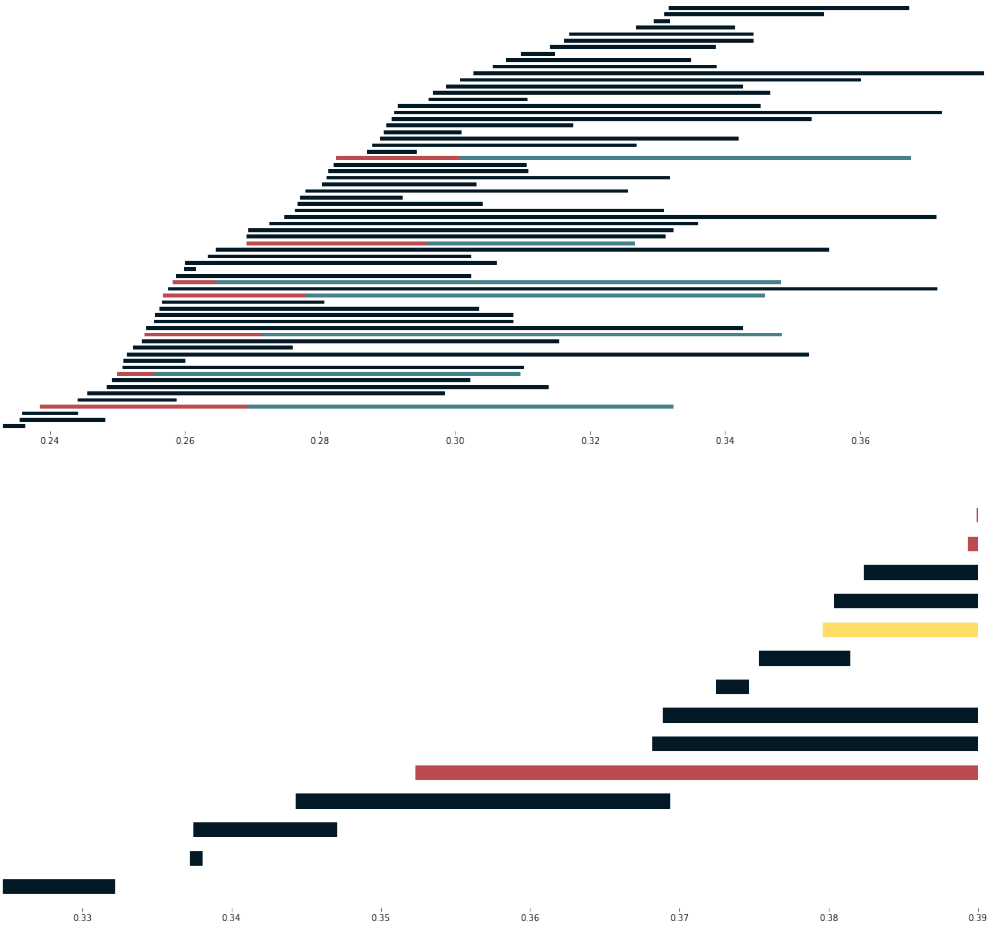
Here we look at the extension information on one dimensional persistence classes. For this we exploit the extra information stored in *MV_ss*. What we do is plot the one dimensional barcodes, highlighting those bars from the `(0,1)` position in the infinity page in red. Also, we highlight in blue when these bars are extended by a bar in the `(1,0)` position on the infinity page. All the black bars are only coming from classes in the `(1,0)` position on the infinity page. Similarly, we also highlight the bars on the second diagonal positions `(2,0)`, `(1,1)`, `(0,2)` by colours yellow, read and blue respectively. If a bar is not extended we write it in black (bars which are not extended are completely contained in `(0,2)`

```
>>> PH = MV_ss.persistent_homology
>>> no_diag = 3
>>> colors = [ "#ffdd66", "#bc4b51", "#468189"]
>>> for diag in range(1, no_diag):
>>>     start_rad = min(PH[diag].barcode[:,0])
>>>     end_rad = max(PH[diag].barcode[:,1])
>>>     persistence = end_rad - start_rad
>>>     fig, ax = plt.subplots(figsize = (20,9))
>>>     ax = plt.axes(frameon=False)
>>>     # ax = plt.axes()
>>>     step = (persistence /2) / PH[diag].dim
>>>     width = (step/6.)
>>>     y_coord = 0
>>>     for b in PH[diag].barcode:
>>>         current_rad = b[0]
>>>         for k in range(diag + 1):
>>>             if k == diag and current_rad == b[0]:
>>>                 break
>>>             if len(MV_ss.Hom[MV_ss.no_pages - 1][diag - k][k].barcode) != 0:
>>>                 for i, rad in enumerate(MV_ss.Hom[
>>>                         MV_ss.no_pages - 1][diag - k][k].barcode[:,0]):
>>>                     if np.allclose(rad, current_rad):
>>>                         next_rad = MV_ss.Hom[
>>>                             MV_ss.no_pages - 1][diag - k][k].barcode[i,1]
>>>                         ax.fill([current_rad, next_rad, next_rad, current_rad],
>>>                                 [y_coord,y_coord,y_coord+step,y_coord+step],
>>>                                 c=colors[k + no_diag - diag - 1])
>>>                         current_rad = next_rad
>>>                     # end if
>>>                 # end for
>>>             # end if
>>>
>>>         # end for
```

(continues on next page)

(continued from previous page)

```
>>>         if current_rad < b[1]:
>>>             ax.fill([current_rad, b[1], b[1], current_rad],
>>>                             [y_coord,y_coord,y_coord+step,y_coord+step],
>>>                             c="#031926")
>>>         # end if
>>>         y_coord = y_coord + 2 * step
>>>     # end for
>>>
>>>     # Show figure
>>>     ax.axes.get_yaxis().set_visible(False)
>>>     ax.set_xlim([start_rad, end_rad])
>>>     ax.set_ylim([-step, y_coord + step])
>>>     plt.show()
```



CHAPTER 4

API reference

4.1 permaviss.persistence_algebra

permaviss.persistence_algebra.PH_classic	PH_classic.py
permaviss.persistence_algebra.barcode_bases	barcode_bases.py
permaviss.persistence_algebra.image_kernel	image_kernel.py
permaviss.persistence_algebra.module_persistence_homology	module_persistence_homology.py

4.1.1 permaviss.persistence_algebra.PH_classic

PH_classic.py

This module implements a function which computes bases for the image and kernel of morphisms between persistence modules.

Functions

persistent_homology(D, R, max_rad, p)	Given the differentials of a filtered simplicial complex *X*, we compute its homology.

permaviss.persistence_algebra.PH_classic.**persistent_homology**(*D*, *R*, *max_rad*, *p*)

Given the differentials of a filtered simplicial complex *X*, we compute its homology.

In this function, the domain is on the columns and the range on the rows. Coordinates are stored as columns in an array. Barcode ranges are stored as pairs in an array of two columns.

Parameters

- **D** (`list(Numpy Array)`) – The ith entry stores the ith differential of the simplicial complex.
- **R** (`list(int)`) – The ith entry contains the radii of filtration for the ith skeleton of *X*. For example, the 1st entry contains, in order, the radii of each edge in *X*. In dimension 0 we have an empty list.
- **p** (*int (prime)*) – Chosen prime to perform arithmetic mod *p*.

Returns

- **Hom** (`list(barcode_bases)`) – The *i* entry contains the *i* Persistent Homology classes for *X*. These are stored as `barcode_bases`. If a cycle does not die we put max_rad as death radius. Additionally, each entry is ordered according to the standard barcode order.
- **Im** (`list(barcode_bases)`) – The *i* entry contains the image of the *i+1* differential as `barcode_bases`.
- **PreIm** (`list(Numpy Array (len(R[*]), Im[*].dim))` – Preimage matrices, 'how to go back from boundaries'
- **CycleKill** (`list(Numpy Array (len(R[*]), Hom[*].dim))` – Boundaries that killed homology cycles.

Example

```
>>> from permaviss.sample_point_clouds.examples import circle
>>> from permaviss.simplicial_complexes.differentials import
... complex_differentials
>>> from permaviss.simplicial_complexes.vietoris_rips import
... vietoris_rips
>>> import scipy.spatial.distance as dist
>>> point_cloud = circle(10, 1)
>>> max_rad = 1
>>> p = 5
>>> max_dim = 3
>>> Dist = dist.squareform(dist.pdist(point_cloud))
>>> compx, R = vietoris_rips(Dist, max_rad, max_dim)
>>> differentials = complex_differentials(compx, p)
>>> Hom, Im, PreIm = persistent_homology(differentials, R, max_rad, p)
>>> print(Hom[0])
Barcode basis
[[ 0.          1.        ]
 [ 0.          0.61803399]
 [ 0.          0.61803399]
 [ 0.          0.61803399]
 [ 0.          0.61803399]
 [ 0.          0.61803399]
 [ 0.          0.61803399]
 [ 0.          0.61803399]
 [ 0.          0.61803399]
 [ 0.          0.61803399]]
[[ 1.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  4.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  4.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  4.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  4.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  4.  0.  0.]
```

(continues on next page)

(continued from previous page)

```
 [ 0.  0.  0.  0.  0.  4.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  4.  0.]
 [ 0.  0.  0.  0.  0.  0.  4.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  4.]]
>>> print(Hom[1])
Barcode basis
[[ 0.61803399  1.        ]]
[[ 4.]
 [ 4.]
 [ 4.]
 [ 4.]
 [ 4.]
 [ 4.]
 [ 4.]
 [ 4.]
 [ 4.]
 [ 1.]]
>>> print(Im[0])
Barcode basis
[[ 0.61803399  1.        ]
 [ 0.61803399  1.        ]
 [ 0.61803399  1.        ]
 [ 0.61803399  1.        ]
 [ 0.61803399  1.        ]
 [ 0.61803399  1.        ]
 [ 0.61803399  1.        ]
 [ 0.61803399  1.        ]
 [ 0.61803399  1.        ]]
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 4.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  4.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  4.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  4.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  4.  0.  0.]
 [ 0.  0.  0.  0.  4.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  4.  0.]
 [ 0.  0.  0.  0.  0.  4.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  4.]]
>>> print(Im[1])
Barcode basis
[]
>>> print(PreIm[0])
[]
>>> print(PreIm[1])
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

4.1.2 permaviss.persistence_algebra.barcode_bases

barcode_bases.py

Classes

barcode_basis(bars[, prev_basis, ...])	This class implements barcode bases.

class permaviss.persistence_algebra.barcode_bases.**barcode_basis**(*bars*, *prev_basis=None*, *coordinates=array([], shape=(1, 0), dtype=float64)*, *store_well_defined=False*, *broken_basis=False*, *broken_differentials=None*)

This class implements barcode bases.

Associated bars and coordinates are given for some barcode base. Each coordinate is stored in a column, whereas bars are stored on rows. This generates a barcode basis with all the input data. There is the exception of broken barcode bases, which come up when solving the extension problem.

Note: Barcode bases are assumed to be well defined in the sense that:

1)They are linearly independent with respect to boxplus operation

2)They generate the respective module or submodule.

Parameters

- **bars** (Numpy Array (dim, 2)) – Each entry is a pair specifying birth and death radius of a bar.
- **prev_basis** (reference to a previously defined *barcode_basis*.) – This will be the basis in which the coordinates are given.
- **coordinates** (Numpy Array (dim, prev_basis.dim)) – Cooridnates of this basis in terms of *prev_basis*
- **store_well_defined** (bool, default is *False*) – Whether we want to store the indices of well defined bars. That is, whether we wish to store indices of bars where the birth radius is strictly smaller than the death radius.
- **broken_basis** (bool, default is *False*) – Whether the barcode basis is *broken*. This appears when solving the extension problem. A barcode base is *broken* if it is not natural.
- **broken_differentials** (Numpy Array (dim, dim)) – Matrix of broken differentials. These give coefficients of a barcode generator in term of other generators. This is used when the given generator dies, and we write it in terms of other generators that are still alive.

Returns

Return type *barcode_basis*

Raises

- **ValueError** – If *prev_coord.dim* is different to the number of rows in *coordinates*
- **ValueError** – If the number of rows in *bar* is different to the number of columns in *coordinates*.
- **ValueError** – If *broken_basis* = *True* but *broken_differentials* is not given.

Examples

```
>>> import numpy as np
>>> bars = np.array([[0,2],[0,1],[2,3],[2,2]])
>>> base1 = barcode_basis(bars, store_well_defined=True)
>>> base1.dim
3
>>> base1.well_defined
array([ True,  True,  True, False], dtype=bool)
>>> print(base1)
Barcode basis
[[0 2]
 [0 1]
 [2 3]]
```

```
>>> bars = np.array([[0,2],[2,3]])
>>> coordinates = np.array([[1,0],
...                         [1,0],
...                         [0,2]])
>>> base2 = barcode_basis(bars, prev_basis, coordinates)
>>> base2.dim
2
>>> print(base2)
Barcode basis
[[0 2]
 [2 3]]
[[1 0]
 [1 0]
 [0 2]]
```

```
>>> bars = np.array([[0,2],[0,1],[1,3]])
>>> broken_differential = np.array(
... [[0,0,0],
...  [0,0,0],
...  [0,1,0]])
>>> base3 = barcode_basis(bars, broken_basis=True,
... broken_differentials=broken_differential)
>>> base3.dim
3
>>> print(base3)
Barcode basis
[[0 2]
 [0 1]
 [1 3]]
```

__str__()
Printing function for barcode bases.

active (*rad*, *start=0*, *end=-1*)
Returns array with active indices at rad.

Option to restrict to generators from start to self.dim

Parameters

- **rad** (*float*) – Radius where we want to check which barcodes are active.
- **start** (*int, default is 0*) – Generator in *self* from which we start to search.
- **end** (*int, default is -1*) – Generator in *self* until which we end searching for active generators.

Returns One-dimensional array with indices of active generators. This is relative to the start index.

Return type Numpy Array

Example

```
>>> print(base3)
Barcode basis
[[0 2]
 [0 1]
 [1 3]]
>>> base3.active(1.2)
array([0, 2])
>>> base3.active(0.8, start=1)
array([0])
```

active_coordinates (*rad*)
Active submatrix of coordinates at rad

Example

```
>>> bars = np.array(
... [[0,5],[1,5],[1,5],[2,4]])
>>> baseP = barcode_basis(bars)
>>> barsC = np.array(
... [[1,4],[2,5]])
>>> coord = np.array(
... [[1,0],
...  [0,1],
...  [2,1],
...  [0,1]])
>>> baseC = barcode_basis(barsC, baseP, coord)
>>> print(baseC)
Barcode basis
[[1 4]
 [2 5]]
[[1 0]
 [0 1]
 [2 1]
 [0 1]]
>>> bars
array([[0, 5],
```

(continues on next page)

(continued from previous page)

```
       [1, 5],
       [1, 5],
       [2, 4]])
>>> baseC.active_coordinates(1.3)
array([[1],
       [0],
       [2]])
>>> baseC.active_coordinates(4.8)
array([[0],
       [1],
       [1]])
>>> baseC.active_coordinates(3)
array([[1, 0],
       [0, 1],
       [2, 1],
       [0, 1]])
```

active_domain(*rad*)

Active columns of coordinates at rad

Example

```
>>> print(baseC)
    Barcode basis
    [[1 4]
     [2 5]]
    [[1 0]
     [0 1]
     [2 1]
     [0 1]]
>>> baseC.active_domain(1.3)
array([[1],
       [0],
       [2],
       [0]])
```

birth_radius(*coordinates*)

This finds the birth radius of a list of coordinates.

Example

```
>>> base3 = barcode_basis([[0,2],[0,1],[1,3]])
>>> base3.birth_radius([1,0,3])
1
```

bool_select(*selection*)

Given a boolean array, we select generators from a barcode basis.

Example

```
>>> print(base3)
Barcode basis
[[0 2]
 [0 1]
 [1 3]]
>>> base5 = base3.bool_select([True,False,True])
>>> print(base5)
Barcode basis
[[0 2]
 [1 3]]
```

changes_list()

Returns an array with the values of changes occurring in the basis.

Returns One-dimensional array with radii where either a bar dies or is born in *self*.

Return type Numpy Array

Example

```
>>> print(base1)
Barcode basis
[[0 2]
 [0 1]
 [2 3]]
>>> base1.changes_list()
array([0, 1, 2, 3])
```

death(*rad*, *start=0*)

Returns an array with the indices dying at rad.

These indices are relative to the optional argument start.

Parameters

- **rad** (*float*) – Radius at which generators might be dying
- **start** (*int, default is 0*) –

Example

```
>>> print(base4)
Barcode basis
[[-1  3]
 [ 0  4]
 [ 1  4]
 [ 1  2]]
>>> base4.death(4)
array([1, 2])
>>> base4.death(4,2)
array([0])
```

death_radius(*coordinates*)

Find the death radius of given coordinates.

Example

```
>>> print(base3)
Barcode basis
[[0 2]
 [0 1]
 [1 3]]
>>> base3.death_radius([1,1,1])
3
```

sort (*precision=7*, *send_order=False*)

Sorts a barcode basis according to the standard barcode order.

That is, from smaller birth radius to bigger, and from bigger death radius to smaller. A precision up to n zeros is an optional argument.

Parameters

- **precision** (*int, default is 7*) – Number of zeros of precision.
- **send_order** (*bool, default is False*) – Whether we want to generate a Numpy Array storing the original order.

Returns One dimensional array storing original order of barcodes.

Return type Numpy Array

Example

```
>>> bars = np.array([[1,2],[0,4],[-1,3],[1,4]])
>>> base4 = barcode_basis(bars)
>>> base4.sort(send_order=True)
array([2, 1, 3, 0])
>>> base4 = barcode_basis(bars)
>>> print(base4)
Barcode basis
[[ 1  2]
 [ 0  4]
 [-1  3]
 [ 1  4]]
>>> base4.sort(send_order=True)
array([2, 1, 3, 0])
>>> print(base4)
Barcode basis
[[-1  3]
 [ 0  4]
 [ 1  4]
 [ 1  2]]
```

trans_active_coord (*coord*, *rad*, *start=0*)

Given coordinates on the active generators, this returns coordinates in the whole basis.

This also can be done relative to a start index higher than 0

Parameters

- **coord** (Numpy Array) – One dimensional array specifying the coordinates on the active basis.
- **rad** (*float*) – Radius on which we are checking for active generators.

- **start** (*int, default is 0*) – We ignore the indices smaller than this. Notice that *coord* as well as the produced absolute coordinates will be adjusted appropriately.

Returns One dimensional array with a

Return type Numpy Array

Example

```
>>> bars = [[0,5],[0,2],[1,4],[0.5,3]]
>>> base7 = barcode_basis(bars)
>>> base7.trans_active_coord([0,1,1],2.4)
array([ 0.,  0.,  1.,  1.])
>>> base7.trans_active_coord([-1,1],3.8)
array([-1.,  0.,  1.,  0.])
>>> base7.trans_active_coord([-1,1],2.2,1)
array([ 0., -1.,  1.])
```

update_broken(*A*, *rad*)

Updates a matrix A, using the broken differentials of the generators dying at rad.

We assume that the broken barcode basis is indexing the rows of A.

Parameters

- **A** (Numpy Array) – columns represent coordinates in the barcode_basis.
- **rad** (*float*) – Radius at which we want to update the coordinates using the broken_differentials.

Returns **A** – return updated matrix

Return type Numpy Array

Example

```
>>> print(base3)
Barcode basis
[[0 2]
 [0 1]
 [1 3]]
>>> A = np.array(
... [[1,1,1],
...  [0,2,-1],
...  [0,1,1]])
>>> base3.update_broken(A, 1)
array([[1, 1, 1],
       [0, 0, 0],
       [0, 3, 0]])
```

4.1.3 permaviss.persistence_algebra.image_kernel

image_kernel.py

This module implements a function which computes bases for the image and kernel of morphisms between persistence modules.

Functions

image_kernel(A, B, F, p[, start_index, ...])	This computes basis for the image and kernel of a persistence morphism.

permaviss.persistence_algebra.image_kernel.**image_kernel**(*A*, *B*, *F*, *p*, *start_index=0*, *prev_basis=None*)

This computes basis for the image and kernel of a persistence morphism. f: A –> B

This is the algorithm described in https://arxiv.org/abs/1907.05228. Recall that for such an algorithm to work the *A* and *B* must be ordered. This is why the function first orders the barcode generators from start_index until A.dim. Additionally, it also orders the barcodes from B. By 'ordered' we mean that the barcodes are sorted according to the standard order of barcodes.

It can also compute relative barcode bases for the image. This is used when computing quotients. The optional argument start_index indicates the minimum index from which we want to compute barcodes relative to the previous generators. That is, given start_index, the function will return image barcodes for

<F[start_dim, ..., A.dim]> mod <F[0,1,..., start_dim-1]>.

At the end the bases for the image and kernel are returned in terms of the original ordering.

Additionally, this handles the case for when B is a broken barcode basis. Notice that in such a case, only the barcode basis of the image will be computed

Parameters

- **A** (barcode_basis object) – Basis of domain.
- **B** (barcode_basis object) – Basis of range. This can be a *broken* barcode basis.
- **F** (Numpy Array (*B*.dim, *A*.dim)) – Matrix associated to the considered persistence morphism.
- **p** (*int*) – prime number of finite field.
- **start_index** (*int, default is 0*) – Index from which we get a barcode basis for the image.
- **prev_basis** (*int, default is None*) – If *start_index* > 0, we need to also give a reference to a basis of barcodes from A[start_dim] until A[A.dim].

Returns

- **Ker** (barcode_basis object) – Absolute/relative basis of kernel of f.
- **Im** (barcode_basis object) – Absolute/relative basis of image of f.
- **PreIm** (*Numpy Array (A.dim, Im.dim)*) – Absolute/relative preimage coordinates of f. That is, each column stores the sums that generate the corresponding Image barcode.

Examples

```
>>> import numpy as np
>>> from permaviss.persistence_algebra.barcode_bases import
... barcode_basis
>>> A = barcode_basis([[1,8],[1,5],[2,5], [4,8]])
>>> B = barcode_basis([[-1,3],[0,4],[0,3.5],[2,5],[2,4],[3,8]])
>>> F = np.array([[4,1,1,0],[1,4,1,0],[1,1,4,0],[0,0,1,4],[0,0,4,1],
```

(continues on next page)

(continued from previous page)

```
... [0,0,0,1]])
>>> p = 5
>>> Im, Ker, PreIm = image_kernel(A,B,F,p)
>>> print(Im)
Barcode basis
[[ 1.   4. ]
 [ 1.   3.5]
 [ 2.   5. ]
 [ 4.   8. ]]
[[ 4.  0.  1.  0.]
 [ 1.  0.  1.  0.]
 [ 1.  2.  4.  0.]
 [ 0.  0.  1.  4.]
 [ 0.  0.  4.  1.]
 [ 0.  0.  0.  1.]]
>>> print(Ker)
Barcode basis
[[ 3.5  8. ]
 [ 4.   5. ]]
[[ 1.  0.]
 [ 1.  4.]
 [ 0.  0.]
 [ 0.  0.]]
>>> print(PreIm)
[[ 1.  1.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

Note: This algorithm will only work if the matrix of the persistence morphism is well defined. That is, a generator can only map to generators that have been born and have not yet died.

4.1.4 permaviss.persistence_algebra.module_persistence_homology

module_persistence_homology.py

This module implements the persistence module homology

Functions

module_persistence_homology(D, Base, p)	Given the differentials of a chain of tame persistence modules, we compute barcode bases for the homology of the chain.
quotient(M, N, p)	Assuming that N generates a submodule of M, we compute a barcode basis for the quotient M / N.

permaviss.persistence_algebra.module_persistence_homology.**module_persistence_homology**(*D*, *Base*, *p*)

Given the differentials of a chain of tame persistence modules, we compute barcode bases for the homology of the chain.

Parameters

- **D** (list (Numpy Array)) – List of differentials of the chain complex.
- **Base** (Numpy Array) – List containing barcode bases for each dimension
- **p** (*int (prime)*) – Prime number to perform arithmetic mod p

Returns

- **Hom** (list (barcode_basis)) – Cycles mod boundaries of differentials, starting with: birth rad, death rad. If a cycle does not die we put max_rad as death radius.
- **Im** (list (barcode_basis)) – List storing bases for the images of differentials
- **PreIm** (list (Numpy Array)) – List storing bases for the preimages of the differentials. That is, which generators produce each image generator. This leads to how to *go back* from boundaries to preimages.

permaviss.persistence_algebra.module_persistence_homology.**quotient**(*M*, *N*, *p*)

Assuming that N generates a submodule of M, we compute a barcode basis for the quotient M / N.

Parameters

- **M** (barcode_basis) – Basis for module
- **N** (barcode_basis) – Basis for submodule of N
- **p** (*int (prime)*) –

Returns **Q** – Barcode basis for the quotient M / N

Return type barcode_basis

4.2 permaviss.gauss_mod_p

permaviss.gauss_mod_p.arithmetic_mod_p	arithmetic_mod_c.py
permaviss.gauss_mod_p.gauss_mod_p	gauss_mod_p.py
permaviss.gauss_mod_p.functions	functions.py

4.2.1 permaviss.gauss_mod_p.arithmetic_mod_p

arithmetic_mod_c.py

Arithmetic functions for working mod c Also includes function for inverses mod a prime number p

Functions

add_arrays_mod_c(A, B, c)	Adds two arrays mod c.
add_mod_c(a, b, c)	Integer addition mod c.
inv_mod_p(a, p)	Returns the inverse of a mod p

permaviss.gauss_mod_p.arithmetic_mod_p.**add_arrays_mod_c**(*A*, *B*, *c*)

Adds two arrays mod c.

Parameters

- **a,b** (Numpy Array(int)) – Two integer arrays to be added.
- **c** (*int*) – Integer to mod out by.

Returns **C** – a + b (mod c)

Return type Numpy Array(int)

permaviss.gauss_mod_p.arithmetic_mod_p.**add_mod_c**(*a*, *b*, *c*)

Integer addition mod c.

Parameters

- **a,b** (*int*) – Integers to be added
- **c** (*int*) – Integer to mod out by

Returns **s**

Return type a + b (mod c)

permaviss.gauss_mod_p.arithmetic_mod_p.**inv_mod_p**(*a*, *p*)

Returns the inverse of a mod p

Parameters

- **a** (*int*) – Number to compute the inverse mod p
- **p** (*int (prime)*) –

Returns **m** – Integer such that m * a = 1 (mod p)

Return type int

Raises **ValueError** – If p is not a prime number

4.2.2 permaviss.gauss_mod_p.gauss_mod_p

gauss_mod_p.py

This module implements Gaussian elimination by columns modulo a prime number p.

Functions

gauss_barcodes(A, row_barcode, col_barcode, ...)	This function implements the Gaussian elimination by columns, but specialized for columns and rows with arbitrary finite barcodes.
gauss_col(A, p)	This function implements the Gaussian elimination by columns.
gauss_col_rad(A, R, start_index, p)	This function implements the Gaussian elimination by columns, but specialized for columns with birth radius.
index_pivot(vect)	Returns the pivot of a 1D array

permaviss.gauss_mod_p.gauss_mod_p.**gauss_barcodes**(*A*, *row_barcode*, *col_barcode*, *start_index*, *p*)

This function implements the Gaussian elimination by columns, but specialized for columns and rows with arbitrary finite barcodes.

It reduces columns starting from start_index by the previous ones. Returns the combinations that lead to those columns. For each column to reduce, performs gaussian elimination on corresponding submatrix.

ROW AND COLUMN BARCODES HAVE TO BE ORDERED, OTHERWISE THERE WILL BE PROBLEMS, ALMOST SURELY.

Parameters

- **A** (Numpy Array) – Matrix to be reduced
- **row_R** (Numpy Array) – Vector with radius of rows
- **col_R** (Numpy Array) – Vector with radius of columns
- **start_index** (*int*) – Index at which reduction starts
- **p** (*int(prime)*) – Prime number. The corresponding field will be Z mod p.

Returns **coefficients** – Matrix recording additions performed, so that we obtain the lifts and coefficients.

Return type Numpy Array

Raises **ValueError** – If reduced columns do not vanish.

permaviss.gauss_mod_p.gauss_mod_p.**gauss_col**(*A*, *p*)

This function implements the Gaussian elimination by columns.

A is reduced by left to right column additions. The reduced matrix has unique column pivots.

Parameters

- **A** (Numpy Array) – Matrix to be reduced
- **p** (*int(prime)*) – Prime number. The corresponding field will be Z mod p.

Returns

- **R** (Numpy Array) – Reduced matrix by left to right column additions.
- **T** (Numpy Array) – Matrix recording additions performed, so that AT = R

permaviss.gauss_mod_p.gauss_mod_p.**gauss_col_rad**(*A*, *R*, *start_index*, *p*)

This function implements the Gaussian elimination by columns, but specialized for columns with birth radius.

A is reduced by left to right column additions starting from start_index. Only columns from a lower index are added to columns with a higher index.

Parameters

- **A** (Numpy Array) – Matrix to be reduced
- **R** (Numpy Array) – Vector with radius
- **start_index** (*int*) – Index at which reduction starts
- **p** (*int(prime)*) – Prime number. The corresponding field will be Z mod p.

Returns **T** – Matrix recording additions performed, so that we obtain the lifts and coefficients.

Return type Numpy Array

Raises **ValueError** – If reduced columns do not vanish.

permaviss.gauss_mod_p.gauss_mod_p.**index_pivot**(*vect*)

Returns the pivot of a 1D array

Parameters **vect** (list(int)) – List of integers to compute pivot from.

Returns Index of last nonzero entry on *vect*. Returns -1 if the list is zero.

Return type int

4.2.3 permaviss.gauss_mod_p.functions

functions.py

This code implements multiplication mod p and solving a linear equation mod p.

Functions

multiply_mod_p(A, B, p)	Multiply matrices mod p.
solve_matrix_mod_p(A, B, p)	Same as *solve_mod_p()*, but with B and X being matrices.
solve_mod_p(A, b, p)	Find the vector x such that A * x = b (mod p)

permaviss.gauss_mod_p.functions.**multiply_mod_p**(*A*, *B*, *p*)
Multiply matrices mod p.

permaviss.gauss_mod_p.functions.**solve_matrix_mod_p**(*A*, *B*, *p*)
Same as *solve_mod_p()*, but with B and X being matrices.

That is, given two matrices A and B, we want to find a matrix X such that A * X = B (mod p)

Parameters

- **A** (Numpy Array) – 2D array
- **B** (Numpy Array) – 2D array
- **p** (*int (prime)*) –

Returns **X** – 2D array solution.

Return type Numpy Array

Raises **ValueError** – There is no solution to the given equation

permaviss.gauss_mod_p.functions.**solve_mod_p**(*A*, *b*, *p*)
Find the vector x such that A * x = b (mod p)

This method assumes that a solution exists to the equation A * x = b (mod p). If a solution does not exist, it raises a ValueError exception.

Parameters

- **A** (Numpy Array) – 2D array
- **b** (Numpy Array) – 1D array
- **p** (*int (prime)*) – Number to mod out by.

Returns **x** – 1D array. Solution to equation.

Return type Numpy Array

Raises **ValueError** – If a solution to the equation does not exist.

4.3 permaviss.sample_point_clouds

permaviss.sample_point_clouds.examples

4.3.1 permaviss.sample_point_clouds.examples

Functions

ball(no_points, radius, dim)	Take random points from a ball of a given dimension.
circle(no_points, radius)	Take random points from a circle on the plane.
grid(hdiv, vdiv)	Take the nodes from a hdiv x vdiv grid on 2D plane.
grid_tridimensional(hdiv, vdiv, ddiv)	Take the nodes from a hdiv x vdiv x ddiv grid on 3D space.
random_circle(no_points, radius, epsilon[, ...])	Take random points around a circle on the plane.
random_cube(no_points, dim)	Take random points from a unit cube around the origin.
random_sphere(no_points, radius, dim)	Take random points from a sphere of a given dimension.
take_sample(point_cloud, no_samples)	Take a subsample from samples using a minmax algorithm.
torus(div, min_rad, max_rad)	Take samples from a torus in 4D space.
torus3D(no_points[, min_rad, max_rad])	Take samples from a torus embedded in 3D space.

permaviss.sample_point_clouds.examples.**ball**(*no_points*, *radius*, *dim*)
Take random points from a ball of a given dimension. This ball has centre *(0, 0, 0)*.

Parameters

- **no_points** (*int*) – Number of points wanted.
- **radius** (*float*) – Radius of the ball.
- **dim** (*int*) – Dimension of the ball.

Returns Coordinates of sampled points from the ball.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**circle**(*no_points*, *radius*)
Take random points from a circle on the plane. This circle has centre *(0, 0)*.

Parameters

- **no_points** (*int*) – Number of points wanted.
- **radius** (*float*) – Radius of the circle.

Returns Coordinates of sampled points from the circle.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**grid**(*hdiv*, *vdiv*)
Take the nodes from a hdiv x vdiv grid on 2D plane.

Parameters

- **hdiv** (*int*) – Number of rows.
- **vdiv** (*int*) – Number of columns.

Returns List of points.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**grid_tridimensional**(*hdiv*, *vdiv*, *ddiv*)
Take the nodes from a hdiv x vdiv x ddiv grid on 3D space.

Parameters

- **hdiv** (*int*) – Number of rows.
- **vdiv** (*int*) – Number of columns.
- **ddiv** (*int*) – Number of flats.

Returns List of points.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**random_circle**(*no_points, radius, epsilon, center=[0, 0]*)
Take random points around a circle on the plane.

Parameters

- **no_points** (*int*) – Number of points wanted.
- **radius** (*float*) – Radius of the circle.
- **epsilon** (*float*) – Noise that we want to apply to each sampled point.
- **centre** (*list(float, float)*) – Two entries specifying the position of the centre.

Returns Coordinates of sampled points from around the circle.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**random_cube**(*no_points*, *dim*)
Take random points from a unit cube around the origin. This cube can be of various dimensions.

Parameters

- **no_points** (*int*) – Number of points wanted.
- **dim** (*int*) – Dimension of the cube.

Returns Coordinates of sampled points from the cube.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**random_sphere**(*no_points*, *radius*, *dim*)
Take random points from a sphere of a given dimension. This sphere has centre *(0, 0, 0)*.

Parameters

- **no_points** (*int*) – Number of points wanted.
- **radius** (*float*) – Radius of the sphere.
- **dim** (*int*) – Dimension of the sphere.

Returns Coordinates of sampled points from the sphere.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**take_sample**(*point_cloud*, *no_samples*)
Take a subsample from samples using a minmax algorithm.

We start from a random point. Then choose the point further appart. Next, we take the point that is further appart from the taken points. Continuing we take all the samples from *point_cloud*.

Parameters

- **point_cloud** (Numpy Array) – List of points to take samples from.

- **np_samples** (*int*) – Number of samples that we want to take. It has to be smaller than the dimension of *point_cloud*.

Returns Matrix storing the coordinates of the sampled points.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**torus**(*div*, *min_rad*, *max_rad*)
Take samples from a torus in 4D space.

Parameters

- **no_points** (*int*) – Number of points to be taken.
- **min_rad** (float, default is *1*) – Radius of circle on section of the torus.
- **max_rad** (float, default is *3*) – Distance from the torus centre to the centre of the section.

Returns List of points.

Return type Numpy Array

permaviss.sample_point_clouds.examples.**torus3D**(*no_points*, *min_rad=1*, *max_rad=3*)
Take samples from a torus embedded in 3D space.

Parameters

- **no_points** (*int*) – Number of points to be taken.
- **min_rad** (float, default is *1*) – Radius of circle on section of the torus.
- **max_rad** (float, default is *3*) – Distance from the torus centre to the centre of the section.

Returns List of points.

Return type Numpy Array

4.4 permaviss.simplicial_complexes

permaviss.simplicial_complexes.vietoris_rips	vietoris_rips.py
permaviss.simplicial_complexes.flag_complex	
permaviss.simplicial_complexes.differentials	differentials.py

4.4.1 permaviss.simplicial_complexes.vietoris_rips

vietoris_rips.py

Functions

vietoris_rips(Dist, max_r, max_dim)	This computes the Vietoris-Rips complex with simplexes of dimension less or equal to max_dim, and with maximum radius specified by max_r

permaviss.simplicial_complexes.vietoris_rips.**_lower_neighbours**(*G*, *u*)

Given a graph *G* and a vertex *u* in *G*, we return a list with the vertices in *G* that are lower than *u* and are connected to *u* by an edge in *G*.

Parameters

- **G** (Numpy Array(no. of edges, 2)) – Matrix storing the edges of the graph.
- **u** (*int*) – Vertex of the graph.

Returns **lower_neighbours** – List of lower neighbours of *u* in *G*.

Return type list

permaviss.simplicial_complexes.vietoris_rips.**vietoris_rips**(*Dist*, *max_r*, *max_dim*)

This computes the Vietoris-Rips complex with simplexes of dimension less or equal to max_dim, and with maximum radius specified by max_r

Parameters

- **Dist** (Numpy Array(no. of points, no. of points)) – Distance matrix of points.
- **max_r** (*float*) – Maximum radius of filtration.
- **max_dim** (*int*) – Maximum dimension of computed Rips complex.

Returns

- **C** (list(Numpy Array)) – Vietoris Rips complex generated for the given parameters. List where the first entry stores the number of vertices, and all other entries contain a Numpy Array with the list of simplices in *C*.
- **R** (list(Numpy Array)) – List with radius of birth for the simplices in *C*. The *i* entry contains a 1D Numpy Array containing each of the birth radii for each *i* simplex in *C*.

4.4.2 permaviss.simplicial_complexes.flag_complex

Functions

flag_complex(G, no_vertices, max_dim)	Compute the flag complex of a graph *G* up to a maximum dimension *max_dim*.

permaviss.simplicial_complexes.flag_complex.**_lower_neighbours**(*G*, *u*)

Given a graph *G* and a vertex *u* in *G*, we return a list with the vertices in *G* that are lower than *u* and are connected to *u* by an edge in *G*.

Parameters

- **G** (Numpy Array(no. of edges, 2)) – Matrix storing the edges of the graph.
- **u** (*int*) – Vertex of the graph.

Returns **lower_neighbours** – List of lower neighbours of *u* in *G*.

Return type list

permaviss.simplicial_complexes.flag_complex.**flag_complex**(*G*, *no_vertices*, *max_dim*)

Compute the flag complex of a graph *G* up to a maximum dimension *max_dim*.

Parameters

- **G** (Numpy Array(no. of edges, 2)) – Matrix storing the edges of the graph.
- **no_vertices** (*int*) – Number of vertices in graph *G*
- **max_dim** (*int*) – Maximum dimension

Returns **fl_cpx** – Flag complex of *G*. The *0* entry stores the number of vertices. For a higher entry *i*, *fl_cpx[i]* stores a Numpy Array matrix with the *i* simplices from *fl_cpx*.

Return type list(Numpy Array)

4.4.3 permaviss.simplicial_complexes.differentials

differentials.py

Functions

complex_differentials(C, p)	Given a simplicial complex C, it returns a list D with its differentials mod p

permaviss.simplicial_complexes.differentials.**complex_differentials**(*C*, *p*)
Given a simplicial complex C, it returns a list D with its differentials mod p

Parameters

- **C** (list(int, Numpy Array, ...)) – The *0* entry stores the number of vertices. For a higher entry *i*, *C[i]* stores a Numpy Array matrix with the *i* simplices from *C*.
- **D** (list(Numpy Array)) – List of differentials of *C*. The *i* entry contains a Numpy Array of the *i* differential for the simplicial complex.

4.5 permaviss.covers

permaviss.covers.cubical_cover

4.5.1 permaviss.covers.cubical_cover

Functions

corners_hypercube(point_cloud)	Returns maximum and minimum corners of hypercube containing point_cloud.
generate_cover(max_div, overlap, point_cloud)	Divides a point cloud into a cubical cover.
intersection_covers(points_IN, simplex)	Computes the points in the intersection specified by a nerve simplex.
nerve_hypercube_cover(div)	Generates the nerve of an hypercube covering.
next_hypercube(pos, div)	Jumps to next hypercube in cubical cover

permaviss.covers.cubical_cover.**corners_hypercube**(*point_cloud*)
Returns maximum and minimum corners of hypercube containing point_cloud.

Parameters **point_cloud** (Numpy Array) – Coordinates of point data. Each row corresponds to a point.

Returns **min_corner, max_corner** – Minimum and maximum corners of containing hypercube.

Return type Numpy Array, Numpy Array

Example

```
>>> from permaviss.sample_point_clouds.examples import random_cube
>>> point_cloud = random_cube(5,2)
>>> point_cloud
array([[-0.30392908, -0.40559307],
       [ 0.4736051 , -0.28257937],
       [-0.41760472, -0.30445089],
       [-0.02406966,  0.001455  ],
       [-0.28425041, -0.11212227]])
>>> min_corner, max_corner = corners_hypercube(point_cloud)
>>> min_corner
array([-0.41760472, -0.40559307])
>>> max_corner
array([ 0.4736051,  0.001455 ])
```

permaviss.covers.cubical_cover.**generate_cover**(*max_div*, *overlap*, *point_cloud*)

Divides a point cloud into a cubical cover.

Receives a point cloud point_cloud in R^n and returns it divided into cubes and their respective intersections. It also generates the nerve of the covering.

Parameters

- **max_div** (*int*) – Number of divisions on the maximum side of point_cloud
- **overlap** (*float*) – Overlap between adjacent hypercubes.
- **point_cloud** (Numpy Array) – Each row contains the coordinates of a point

Returns

- **divided_point_cloud** (list(list(Numpy Array 2D))) – Point cloud coordinates indexed by nerve. The *i* entry contains the point cloud coordinates indexed by the *i* simplices of the nerve. That is, the first entry contains the coordinates contained in hypercubes. The second entry the coordinates of points in double intersections of hypercubes. And so on.
- **points_IN** (list(list(Numpy Array 1D))) – Identification Numbers (IN) of points in regions indexed by nerve. That is, this is the same as *divided_point_cloud* but storing IN instead of coordinates.
- **nerve** (list(Numpy Array)) – The nerve of the hypercube cover.

Example

```
>>> point_cloud = circle(5, 1)
>>> max_div = 2
>>> overlap = 0.5
>>> divided_point_cloud, points_IN, nerve = generate_cover(max_div,
... overlap, point_cloud)
```

(continues on next page)

(continued from previous page)

```
>>> divided_point_cloud[0]
[array([[-0.80901699, -0.58778525],
       [ 0.30901699, -0.95105652]]), array([[ 0.30901699,  0.95105652],
       [-0.80901699,  0.58778525]]), array([[ 1.        ,  0.        ],
       [ 0.30901699, -0.95105652]]), array([[ 1.        ,  0.        ],
       [ 0.30901699,  0.95105652]])]
>>> divided_point_cloud[1]
[array([], shape=(0, 2), dtype=float64), array([[ 0.30901699,
-0.95105652]]), array([], shape=(0, 2), dtype=float64), array([],
shape=(0, 2), dtype=float64), array([[ 0.30901699,  0.95105652]]),
array([[ 1.,  0.]])]
>>> points_IN[0]
[array([3, 4]), array([1, 2]), array([0, 4]), array([0, 1])]
>>> points_IN[1]
[array([], dtype=float64), array([4]), array([], dtype=float64),
array([], dtype=float64), array([1]), array([0])]
>>> nerve[0]
4
>>> nerve[1]
array([[ 0.,  1.],
       [ 0.,  2.],
       [ 1.,  2.],
       [ 0.,  3.],
       [ 1.,  3.],
       [ 2.,  3.]])
>>> nerve[2]
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  3.],
       [ 0.,  2.,  3.],
       [ 1.,  2.,  3.]])
>>> nerve[3]
array([[ 0.,  1.,  2.,  3.]])
```

permaviss.covers.cubical_cover.**intersection_covers**(*points_IN*, *simplex*)

Computes the points in the intersection specified by a nerve simplex.

Parameters

- **points_IN** (list(list(Numpy Array 1D))) – Identification Numbers (IN) of points in covering hypercubes.
- **simplex** (Numpy Array) – Simplex in nerve which specifies intersection between hypercubes.

Returns **points_IN_intersection** – IN of points in the intersection specified by simplex.

Return type list(int)

Example

```
>>> import numpy as np
>>> points_IN = [np.array([0,3,5]),np.array([0,1]), np.array([1,5])]
>>> simplex = np.array([0,1])
>>> intersection_covers(points_IN, simplex)
[0]
```

permaviss.covers.cubical_cover.**nerve_hypercube_cover**(*div*)

Generates the nerve of an hypercube covering.

Given an array of divisions of an hypercube cover, this returns the nerve.

Parameters **div** (Numpy Array) – 1D array of divisions per dimension.

Returns **nerve** – Nerve associated to hypercube covering.

Return type list(Numpy Array)

Example

Nerve for three dimensional covering. There are 6 hypercubes and the divisions per dimension are given as 1 x 3 x 2.

```
>>> div = [1,3,2]
>>> nerve = nerve_hypercube_cover(div)
>>> nerve[0]
6
>>> nerve[1]
array([[ 0.,  1.],
       [ 0.,  2.],
       [ 1.,  2.],
       [ 0.,  3.],
       [ 1.,  3.],
       [ 2.,  3.],
       [ 2.,  4.],
       [ 3.,  4.],
       [ 2.,  5.],
       [ 3.,  5.],
       [ 4.,  5.]])
>>> nerve[2]
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  3.],
       [ 0.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 2.,  3.,  4.],
       [ 2.,  3.,  5.],
       [ 2.,  4.,  5.],
       [ 3.,  4.,  5.]])
>>> nerve[3]
array([[ 0.,  1.,  2.,  3.],
       [ 2.,  3.,  4.,  5.]])
>>> nerve[4]
[]
```

permaviss.covers.cubical_cover.**next_hypercube**(*pos*, *div*)

Jumps to next hypercube in cubical cover

Parameters

- **pos** (list) – List of integer values specifying the position of the current hypercube. This is edited to the next hypercube.
- **div** (list) – List of integer values specifying how many hypercubes divide each dimension.

Example

```
>>> pos = [1,1,1]
>>> div = [3,3,2]
>>> next_hypercube(pos, div)
>>> pos
[1, 2, 0]
```

4.6 permaviss.spectral_sequence

permaviss.spectral_sequence.MV_spectral_seq	This module implements the Mayer-Vietoris spectral sequence management.
permaviss.spectral_sequence.spectral_sequence_class	
permaviss.spectral_sequence.local_chains_class	This module takes care of chains stored on first page of the spectral sequence.

4.6.1 permaviss.spectral_sequence.MV_spectral_seq

This module implements the Mayer-Vietoris spectral sequence management.

Functions

create_MV_ss(point_cloud, max_r, max_dim, ...)	This function creates a Mayer Vietoris spectral sequence with the given parameters.
local_persistent_homology(nerve_point_cloud, ...)	This function computes the Vietoris Rips complex and persistent homology of a covering region.

permaviss.spectral_sequence.MV_spectral_seq.**create_MV_ss**(*point_cloud*, *max_r*, *max_dim*, *max_div*, *overlap*, *p*)

This function creates a Mayer Vietoris spectral sequence with the given parameters. The procedure has four main steps:

1) Obtain a cover and a nerve associated to it.
2) Compute the persistent homology on each cover, intersections, and so on.
3) Compute spectral sequence pages until they collapse.
4) Solve the extension problem

Parameters

- **point_cloud** (*Numpy Array*) – Coordinates for given points. Each row corresponds to a point.
- **max_r** (*float*) – Maximum radius of persistence.
- **max_dim** (*int*) – Maximum dimension of simplexes in Vietoris-Rips complex.
- **max_div** (*int*) – Number of division hypercubes on the dimension with maximum length on point cloud.

- **overlap** (*float*) – Overlap between adjacent covers.

Returns MV_ss

Return type spectral_sequence object containing all the information.

Example

```
>>> from permaviss.sample_point_clouds.examples import random_cube,
... take_sample
>>> X = random_cube(1000,3)
>>> point_cloud = take_sample(X,130)
>>> max_r = 0.36
>>> max_dim = 3
>>> p = 3
>>> max_div = 2
>>> overlap = max_r*1.01
>>> MV_ss = create_MV_ss(point_cloud, max_r, max_dim, max_div, overlap,
... p)
PAGE: 1
[[ 25   7   4   1   0   0   0   0   0]
 [160 118 128 144 112  56  16   2   0]
 [310 380 436 445 336 168  48   6   0]]
PAGE: 2
[[ 21   0   0   0   0   0   0   0   0]
 [ 98   2   0   0   0   0   0   0   0]
 [131   5   1   0   0   0   0   0   0]]
PAGE: 3
[[ 21   0   0   0   0   0   0   0   0]
 [ 97   2   0   0   0   0   0   0   0]
 [131   5   0   0   0   0   0   0   0]]
PAGE: 4
[[ 21   0   0   0   0   0   0   0   0]
 [ 97   2   0   0   0   0   0   0   0]
 [131   5   0   0   0   0   0   0   0]]
>>> print(MV_ss.persistent_homology[0].dim)
131
>>> print(MV_ss.persistent_homology[1].dim)
97
>>> print(MV_ss.persistent_homology[2].dim)
21
```

permaviss.spectral_sequence.MV_spectral_seq.**local_persistent_homology**(*nerve_point_cloud*, *max_r*, *max_dim*, *p*, *n_dim*, *spx_idx*)

This function computes the Vietoris Rips complex and persistent homology of a covering region.

It is meant to be run in parallel.

Parameters

- **nerve_point_cloud** (list(list(Numpy Array))) – Local point cloud coordinates indexed by nerve. The first entry contains a list of the point cloud coordinates for each covering region. The second entry contains a list of the point cloud coordinates for each double intersection of covering regions. And so on.

- **points_IN** (list(list(Numpy Array))) – Local Identification Numbers (IN) indexed by nerve. That is, this is the same as *nerve_point_cloud*, but containing IN instead of coordinates for each point.
- **max_r** (*float*) – Maximum radius for computing persistent homology.
- **max_dim** (*int*) – Maximum dimension for complexes
- **n_dim** (*int*) – Current dimension in Nerve of cover
- **spx_idx** (*int*) – Index of n_dim simplex of the covering nerve.

Returns

- **local_complex** (list(Numpy Array)) – See *permaviss.simplicial_complexes.vietoris_rips*
- **local_differentials** (list(Numpy Array)) – See *permaviss.simplicial_complexes.differentials*
- **Hom, Im, PreIm** (list(barcode_basis), list(barcode_basis),) – list(Numpy Array) See *permaviss.persistence_algebra.PH_classic.persistent_homology()*

4.6.2 permaviss.spectral_sequence.spectral_sequence_class

Classes

spectral_sequence(nerve, nerve_point_cloud, ...)	Space and methods for Mayer-Vietoris spectral sequences

class permaviss.spectral_sequence.spectral_sequence_class.**spectral_sequence**(*nerve*, *nerve_point_cloud*, *points_IN*, *max_dim*, *max_r*, *no_pages*, *p*)

Space and methods for Mayer-Vietoris spectral sequences

Parameters

- **nerve** (list(Numpy Array)) – Simplicial complex storing the nerve of the covering. This is stored as a list, where the ith entry contains a Numpy Array storing all the ith simplices; a simplex for each row.
- **nerve_point_cloud** (list(list(Numpy Array))) – Point clouds indexed by nerve of the cover, see *permaviss.covers.cubical_cover*
- **points_IN** (list(list(Numpy Array))) – Point Identification Numbers (IN) indexed by nerve of the cover, see *permaviss.covers.cubical_cover*
- **max_dim** (*int*) – Maximum dimension of simplices.
- **max_r** (*float*) – Maximum persistence radius.
- **no_pages** (*int*) – Number of pages of the spectral sequence
- **p** (*int (prime)*) – The prime number so that our computations are mod p

nerve, nerve_point_cloud, points_IN, max_dim, max_r, no_pages, p
as described above

nerve_differentials
Differentials of Nerve. Used for computing Cech Complex.

Type `list(Numpy Array)`

no_rows, no_columns
Number of rows and columns in each page

Type int, int

nerve_differentials
List storing the differentials of the Nerve. The ith entry stores the matrix of the ith differential.

Type `list(Numpy Array)`

subcomplexes
List storing the simplicial complex on each cover element. For integers *n_dim*, *k* and *dim* the variable *subcomplexes[n_dim][k][dim]* stores the *dim*-simplices on the cover indexed by the *k* simplex of dimension *n_dim* in the nerve.

Type `list(list(list(Numpy Array)))`

zero_diff
List storing the vertical differential matrices on the 0 page of the spectral sequence. For integers *n_dim*, *k* and *dim* the variable *zero_diff[n_dim][k][dim]* stores the *dim* differential of the complex on the cover indexed by the *k* simplex of dimension *n_dim* in the nerve.

Type `list(list(list(Numpy Array)))`

cycle_dimensions
List storing the number of bars on each local persistent homology. Given two integers *n_dim* and *dim*, the variable *cycle_dimensions[n_dim][dim]* contains a list where each entry corresponds to an *n_dim* simplex in the nerve. For each such entry, we store the number of nontrivial persistent homology classes of dimension *dim* in the corresponding cover.

Type `list(list(list(int)))`

Hom
Homology for each page of the spectral sequence. Given three integers which we denote *n_dim*, *nerv_spx* and *deg* we have that *Hom[0][n_dim][nerv_spx][deg]* stores a `barcode_basis` with the *deg*-persistent homology of the covering indexed by *nerve[n_dim][nerv_spx]*. All these store the homology on the *0* page of the spectral sequence. Additionally, for integers $k > 0$, *n_dim* and *deg*, we store in *Hom[k][n_dim][deg]* the `barcode_basis` for the homology on the *(deg, n_dim)* entry in the *k* page of the spectral sequence.

Type `list(...(list(barcode_basis)))`

Im
Image for each page of the spectral sequence. Given three integers which we denote *n_dim*, *nerv_spx* and *deg* we have that *Im[0][n_dim][nerv_spx][deg]* stores a `barcode_basis` for the image of the *deg+1*-differential of the covering indexed by *nerve[n_dim][nerv_spx]*. All these store the images on the *0* page of the spectral sequence. Additionally, for integers $k > 0$, *n_dim* and *deg*, we store in *Im[k][n_dim][deg]* the `barcode_basis` for the image on the *(deg, n_dim)* entry in the *k* page of the spectral sequence.

Type `list(...(list(barcode_basis)))`

PreIm
Preimages for each page of the spectral sequence. Given three integers which we denote *n_dim*, *nerv_spx* and *deg* we have that *PreIm[0][n_dim][nerv_spx][deg]* stores a `Numpy Array` for the Preimage of the *deg+1*-differential of the covering indexed by *nerve[n_dim][nerv_spx]*. Additionally, for integers $k > 0$,

n_dim and *deg*, we store in *PreIm[k][n_dim][deg]* a `Numpy Array` for the preimages of the differential images in the *(deg, n_dim)* entry in the *k* page of the spectral sequence.

Type `list(...(list(Numpy Array)))`

tot_complex_reps
The asterisc * on the type can be either [] or `list(Numpy Array)`. This is used for storing complex representatives for the cycles.

Type `list(list(*))`

page_dim_matrix
Array storing the dimensions of the entries in each page. Notice that the order in which we store columns and rows differs from all the previous attributes.

Type `Numpy Array(no_pages+1, max_dim, no_columns)`

persistent_homology
List storing the persistent homology generated by the spectral sequence. The *i* entry contains the *i* dimensional persistent homology.

Type `list(barcode_basis)`

order_diagonal_basis
This intends to store the original order of *persistent_homology* before applying the standard order.

Type *list*

extensions
Nested lists, where the first two indices are for the column and row. The last index indicates the corresponding extension matrix.

Type `list(list(list(Numpy Array)))`

Notes

The indexing on the 0 page is different from that of the next pages. This is because we do not want to store all the 0 page information on the same place.

add_output_first(*output*, *n_dim*)
Stores the 0 page data of *n_dim* column after it has been computed in parallel by *multiprocessing.pool*

Parameters

- **output** (`list`) – Result after using *multiprocessing.pool* on `local_persistent_homology()`
- **n_dim** (*int*) – Column of *0*-page whose data has been computed.

add_output_higher(*Hom*, *Im*, *PreIm*, *end_n_dim*, *end_deg*, *current_page*)
Stores higher page data that has been computed along a sequence of consecutive differentials.

The studied sequence of differentials ends in

(end_n_dim, end_deg)

coming from

(end_n_dim + current_page, end_deg - current_page + 1)

and continuing until reaching an integer $r > 0$ such that either

end_n_dim + r * current_page > self.no_columns

or

end_deg - r * current_page + 1 > 0

Parameters

- **Hom** (list(barcode_basis)) – Homology of a sequence of differentials in the spectral sequence. This is computed using *permaviss.persistence_algebra.module_persistence_homology*.
- **Im** (list(barcode_basis)) – Images of a sequence of differentials in the spectral sequence.
- **PreIm** (list(Numpy Array)) – Preimages of a sequence of differentials in the spectral sequence.
- **end_n_dim** (*int*) – Integer specifying the column position where the sequence of differentials ends.
- **end_deg** (*int*) – Integer specifying the row position where the sequence of differentials ends.
- **current_page** (*int*) – Current page of the spectral sequence.

cech_diff(*n_dim*, *deg*, *start_chains*)

Given chains in (n_dim + 1, deg), compute Cech differential.

Parameters

- **deg** (*n_dim,*) – Codomain position in spectral sequence.
- **chains** (local_chains object) – Chains on (n_dim+1, deg) that are stored as references in chains[0] and local coordinates as rows in chains[1].

Returns **image_chains** – Image coordinates of Cech differential.

Return type Local Coordinates

cech_diff_and_lift(*n_dim*, *deg*, *start_chains*, *R*)

Given chains in position (n_dim, deg), computes horizontal differential followed by lift by vertical differential.

Procedure: (1) take chains in position (n_dim, deg) (2) compute the Cech differential of these chains. We do this in parallel over the covers in (n_dim-1, deg) (3) Lift locally. Steps (2) and (3) are parallelized at the same time.

Parameters

- **deg, current_page** (*n_dim,*) – Postition on spectral sequence and current page.
- **chains** (list(list(Numpy Array))) –

Returns

- **betas** (*coordinates on first pages*)
- **[lift_references, lift_coordinates]** (*local coordinates lifted by*) – vertical differential.

cech_diff_and_lift_local(*R*, *start_chains*, *n_dim*, *deg*, *nerve_spx_index*)

Takes some chains in position (n_dim+1, deg) and computes Cech diff followed by a lift by vertical differential. This is done locally at cover information in (n_dim, deg).

This method is meant to be run in parallel.

Parameters

- **R** (list) – Vector of radii

- **start_chains** (local_chains object) – Chains in position (n_dim + 1, deg)
- **deg, nerve_spx_index** (*n_dim,*) – Position in spectral sequence and local index.

Returns

- **betas_1_page** (Numpy Array) – Coefficients of lift to 1st page on position (n_dim, deg)
- **local_lift_references** (list) – List of local references of lift.
- **local_lift_coordinates** (Numpy Array) – Local coordinates of lift.

cech_diff_local (*start_chains*, *n_dim*, *deg*, *nerve_spx_index*)
Local Cech differential, starting from chains in (n_dim + 1, deg).

Parameters

- **start_chains** (local_chains object) – Chains to compute Cech differential from.
- **deg, nerve_spx_index** (*n_dim,*) – Position in spectral sequence and local index.

Returns

- **local_image_ref** (list) – List of local references of image.
- **local_image_coord.T** (Numpy Array) – Local coordinates of image. Expressions correspond to rows while local simplices correspond to columns.

compute_higher_representatives (*n_dim*, *deg*, *current_page*)
Computes total complex representatives for current_page classes in position (n_dim, deg).

Resulting representatives written in *self.Hom_reps[current_page][n_dim][deg]*

Parameters **deg, current_page** (*n_dim,*) – Position on the spectral sequence and current page.

compute_two_page_representatives (*n_dim*, *deg*)
Computes total complex representatives for second page classes in position (n_dim, deg).

Resulting representatives are written in self.Hom_reps[1][n_dim][deg]

Parameters **deg** (*n_dim,*) – These specify the position on the spectral sequence where we want to compute and store the second page representatives.

extension (*start_n_dim*, *start_deg*)
Take information from spectral sequence class, and calculate extension coefficients for a given position (start_deg, start_n_dim).

first_differential (*n_dim*, *deg*)
Compute differential on first page (n_dim, deg) –> (n_dim-1, deg)

Parameters **deg** (*n_dim,*) – Differential domain position on first page.

Returns **Betas** – Coefficients of image of first page differentials. The image of each class from (n_dim, deg) is given as a row.

Return type np.array

first_page_lift (*n_dim*, *deg*, *start_chains*, *R*)
Given some chains in position (n_dim, deg), lift to first page accross several covers.

Parameters

- **deg, current_page** (*n_dim,*) – Postition on spectral sequence and current page.
- **start_chains** (local_chains object) – Chains in position (n_dim, deg) that we lift to first page.

- **R** (list) – Values at which we lift *start_chains*

Returns

- **Betas_1_page** (Numpy Array) – Coordinates on first page. Rows correspond to expressions and columns to homology classes.
- **lift_chains** (local_coord object) – Chains after lifting vertically by horizontal differential.

first_page_local_lift (*n_dim*, *deg*, *local_coord*, *lift_radii*, *nerve_spx_index*)

Lift to first page on a given open cover.

Parameters

- **deg** (*n_dim,*) – Position on spectral sequence.
- **local_coord** (Numpy Array) – Local coordinates to be lifted to first page and vertical differential. Rows are expressions while columns correspond to local simplices.
- **lift_radi** (list) – Values at which we want to lift *start_chains* by the vertical differential.
- **nerve_spx_inex** (*int*) – Local index. This function is meant to be parallelized over this.

Returns

- **gammas** (Numpy Array) – 2D Matrix expressing coefficients of lift. Each expression corresponds to a column, while image generators correspond to rows.
- **betas** (Numpy Array) – 2D Matrix expressing coefficients in terms of homology classes on page 1. Expressions correspond to columns, while homology classes correspond to rows.

high_differential (*n_dim*, *deg*, *current_page*)

Compute differential on *current-page* (n_dim, deg) –> (n_dim - current_page, deg + current_page - 1).

Parameters **deg** (*n_dim,*) – Differential domain position.

Returns **Betas** – Coefficients of image of current_page differentials. The image of each class from (n_dim, deg) is given as a row.

Return type np.array

lift_to_page (*n_dim*, *deg*, *target_page*, *Betas*, *Beta_barcode*)

Lifts chains in position (n_dim, deg) from page 1 to target_page

Returns Betas and image coordinates.

Parameters

- **deg** (*n_dim,*) – Differential domain position.
- **target_page** (*int*) – Lift classes up to this page.
- **Betas** (*np.array*) – Coordinates of classes on first page.
- **Betas_barcode** (*np.array*) – Barcodes of classes to be lifted.

Returns

- **Betas.T** (*np.array*) – Coefficients of image of current_page differentials. The image of each class from (n_dim, deg) is given as a row.

- **Gammas.T** (*np.array*) – Coefficients of added differentials of (current_page - 1) page. This is such that the sum of differentials using Gammas, plus adding classes using target_betas leads to the original Betas.

local_cech_matrix(*n_dim*, *deg*, *nerve_spx_index*, *nerve_face_index*, *nerve_coeff*)
Returns matrix of Cech differential in (n_dim, deg) restricted on component (nerve_face_index, nerve_spx_index).

Parameters

- **deg** (*n_dim,*) – Position in spectral sequence.
- **nerve_face_index** (*nerve_spx_index,*) – Local indices in domain and codomain respectively.
- **nerve_coeff** (*int*) – Coefficient in nerve differential determined by the pair nerve_spx_index and nerve_face_index.

Returns boundary – Matrix of size (subcpx[n_dim-1][nerve_face_index][deg], subcpx[n_dim][nerve_spx_index][deg]) that represents the local cech differential.

Return type Numpy Array

4.6.3 permaviss.spectral_sequence.local_chains_class

This module takes care of chains stored on first page of the spectral sequence.

The format will be, for a certain position (n_deg, deg): References on each open cover of intersection degree *n_deg* Local chain coordinates as rows for each reference.

Classes

local_chains(*args)	Chains in local forms, references and local coordinates.

class permaviss.spectral_sequence.local_chains_class.**local_chains**(**args*)
Chains in local forms, references and local coordinates.

p
Prime number to be used for computations.

Type int or None

__add__(*B*)
Given two local chains, adds then and returns the result.

Assumes that references are the same

Parameters **B** (*:class:local_chains* object) –

add_entry(*index*, *ref*, *coord*)
Add a references and coordinates at some index.

Checks that the input is correct.

Parameters

- **index** (*int*) – Position in local_chains object.
- **ref** (list) – Reference list for current index.

- **coord** (Numpy Array or list) – Local coordinates where expressions are indexed by rows. If it is a list it is the empty list [].

Raises **ValueError** – If the given pair of ref and coord do not respect the format rules.

copy_seq()
Given a sequence of local chains, makes a copy and returns it.

minus()
Minus all local coordinates of self.

sums(*coord_sums*)
Sum chains as indicated by "coord_sums"

Parameters

- **chains** (*local_chains* object) – Chains in local_chains format to be added.
- **coord_sums** (Numpy Array) – Each sum is given by rows in coord_sums. These store the coefficients that the chain entries need to be added.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

Z