JAVAPLEX TUTORIAL

HENRY ADAMS AND ANDREW TAUSZ

Contents

1. Introduction

1.1. **javaPlex.** javaPlex is a Java software package for computing the persistent homology of filtered chain complexes, with special emphasis on applications arising in topological data analysis. The main author is Andrew Tausz. javaPlex is a re-write of the JPlex package, which was written by Harlan Sexton and Mikael Vejdemo Johansson. The main motivation for the development of javaPlex was the need for a flexible platform that supported new directions of research in topological data analysis and computational persistent

Date: November 18, 2011.

homology. The website for javaPlex is `http://code.google.com/p/javaplex/` and the javadoc tree for the library is at `http://javaplex.googlecode.com/svn/trunk/doc/index.html`.

Please email Henry at `henrya@math.stanford.edu` or Andrew at `atausz@stanford.edu` if you have questions about this tutorial.

If you are interested in javaPlex, then you may also be interested in the software package Dionysus by Dmitriy Morozov, available at `http://www.mrzv.org/software/dionysus`.

Some of the exercises in this tutorial are borrowed from Vin de Silva's *Plexercises*, available at `http://comptop.stanford.edu/u/programs/Plexercises2.pdf`.

1.2. **License.** javaPlex is an open source software package under the Open BSD License. The source code can be found at `http://code.google.com/p/javaplex/`. If you are interested in contributing to the project, we invite you to contact either of the authors.

1.3. **Installation for Matlab.** Open Matlab and check which version of Java is being used. In this tutorial, the symbol `>>` precedes commands to enter into your Matlab window.

```
>> version -java
ans = Java 1.5.0_13 with Apple Inc.   Java Hotspot(TM) Client VM mixed mode, sharing
```

javaPlex requires version number `1.5` or higher.

To install javaPlex for Matlab, go the the website `http://code.google.com/p/javaplex/downloads/list`. Download the zip file containing the Matlab examples. It should be called something like `matlab-examples-4.0.0.tar.gz`. Extract the zip file. The resulting folder should be called `matlab-examples`.

Change directories in Matlab to `matlab-examples`. Run the `load_javaplex.m` file.

```
>> load_javaplex
```

Installation is complete. Confirm that javaPlex is working properly with the following command.

```
>> api.Plex4.createExplicitSimplexStream()
ans = edu.stanford.math.plex4.streams.impl.ExplicitSimplexStream@513fd4
```

Your output should be the same except for the last several characters. Each time upon starting a new Matlab session, you will need to run `load_javaplex.m`.

1.4. **Accompanying files.** The following Matlab scripts containing the commands in this tutorial are available in the folder `matlab_examples/tutorial_examples`. This means that you don't need to type in each command individually.

- `core_subsets_example.m`
- `euler_characteristic_example.m`
- `explicit_metric_space_example.m`
- `explicit_simplex_example.m`
- `house_example.m`
- `image_patch_example.m`
- `landmark_example.m`
- `lazy_witness_example.m`
- `pointcloud_example.m`
- `rips_example.m`
- `witness_example.m`

The folder `matlab_examples/tutorial_examples` also contains the following Matlab functions

- `coreSubset.m`
- `dct.m`
- `eulerCharacteristic.m`

and the following Matlab data files

- `pointsRange.mat`
- `pointsTorusGrid.mat`

which are used in this tutorial.

The folder `matlab_examples/tutorial_solutions` contains the following solution scripts to tutorial exercises.

- `exercise_3_1_1.m`
- `exercise_3_1_2.m`
- `exercise_3_1_3.m`
- `exercise_4_2_1.m`
- `exercise_4_2_2.m`
- `exercise_5_1_2.m`
- `exercise_5_1_3.m`
- `exercise_5_4_4.m`
- `exercise_5_4_5.m`
- `flatKleinDistanceMatrix.m`
- `flatTorusDistanceMatrix.m`

See Appendix B for exercise solutions.

2. Math review

Below is a brief math review. For more details, see [2, 5, 7, 10].

2.1. **Simplicial complexes.** An abstract simplicial complex is given by the following data.

- A set Z of vertices or 0-simplices.
- For each $k \geq 1$, a set of k-simplices $\sigma = [z_0 z_1 ... z_k]$, where $z_i \in Z$.
- Each k-simplex has $k+1$ faces obtained by deleting one of the vertices. The following membership property must be satisfied: if σ is in the simplicial complex, then all faces of σ must be in the simplicial complex.

We think of 0-simplices as vertices, 1-simplices as edges, 2-simplices as triangular faces, and 3-simplices as tetrahedrons.

2.2. **Homology.** Betti numbers help describe the homology of a simplicial complex X. The value $Betti_k$, where $k \in \mathbb{N}$, is equal to the rank of the k-th homology group of X. Roughly speaking, $Betti_k$ gives the number of k-dimensional holes. In particular, $Betti_0$ is the number of connected components. For instance, a k-dimensional sphere has all Betti numbers equal to zero except for $Betti_0 = Betti_k = 1$.

2.3. **Filtered simplicial complexes.** A filtration on a simplicial complex X is a collection of subcomplexes $\{X(t) \mid t \in \mathbb{R}\}$ of X such that $X(t) \subset X(s)$ whenever $t \leq s$. The filtration time of a simplex $\sigma \in X$ is the smallest t such that $\sigma \in X(t)$. In javaPlex, filtered simplicial complexes (or more generally filtered chain complexes) are called streams.

2.4. **Persistent homology.** Betti intervals help describe how the homology of $X(t)$ changes with t. A k-dimensional Betti interval, with endpoints $[t_{start}, t_{end})$, corresponds roughly to a k-dimensional hole that appears at filtration time t_{start}, remains open for $t_{start} \leq t < t_{end}$, and closes at time t_{end}. We are often interested in Betti intervals that persist for a long filtration range.

Persistent homology depends heavily on functoriality: for $t \leq s$, the inclusion $i : X(t) \to X(s)$ of simplicial complexes induces a map $i_* : H_k(X(t)) \to H_k(X(s))$ between homology groups.

3. Explicit simplex streams

In javaPlex, filtered simplicial complexes (or more generally filtered chain complexes) are called streams. The class ExplicitSimplexStream allows one to build a simplicial complex from scratch. In Section 5 we will learn about other automated methods of generating simplicial complexes; namely the Vietoris–Rips, witness, and lazy witness constructions.

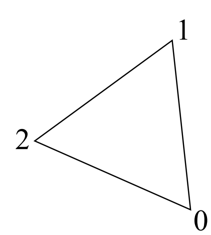
3.1. **Computing homology.** You should change your current Matlab directory to `tutorial_examples`, perhaps using the following command.

```
>> cd tutorial_examples
```

The Matlab script corresponding to this section is `explicit_simplex_example.m`, which is in the folder `tutorial_examples`. You may copy and paste commands from this script into the Matlab window, or you may run the entire script at once with the following command.

```
>> explicit_simplex_example
```

Circle example. Let's build a simplicial complex homeomorphic to a circle. We have three 0-simplices: [0], [1], [2], and three 1-simplices: [0,1], [0,2], [1,2].



To build a simplicial complex in javaPlex we simply build a stream in which all filtration times are zero. First we create an empty explicit simplex stream. Many command lines in this tutorial will end with a semicolon to supress unwanted output.

```
>> stream = api.Plex4.createExplicitSimplexStream();
```

Next we add simplicies using the methods `addVertex` and `addElement`. The first creates a vertex with a specified index, and the second creates a k-simplex (for $k > 0$) with the specified array of vertices. Since we don't specify any filtration times, by default all added simplices will have filtration time zero.

```
>> stream.addVertex(0);
>> stream.addVertex(1);
>> stream.addVertex(2);
>> stream.addElement([0, 1]);
>> stream.addElement([0, 2]);
>> stream.addElement([1, 2]);
>> stream.finalizeStream();
```

We print the total number of simplices in the complex.

```
>> num_simplices = stream.getSize()
num_simplices = 6
```

We create an object that will compute the homology of our complex. The first input parameter 3 indicates that homology will be computed in dimensions 0, 1, and 2 — that is, in all dimensions strictly less than 3. The second input 2 means that we will compute homology with $\mathbb{Z}_2$ coefficients, and this input can be any prime number.

```
>> persistence = api.Plex4.getModularSimplicialAlgorithm(3, 2);
```

We compute and print the intervals.

```
>> circle_intervals = persistence.computeIntervals(stream)
circle_intervals =

Dimension: 1
[0.0, infinity)
Dimension: 0
[0.0, infinity)
```

This gives us the expected Betti numbers $Betti_0 = 1$ and $Betti_1 = 1$.

We can also compute a representative cycle for each barcode.

```
>> circle_intervals = persistence.computeAnnotatedIntervals(stream)
circle_intervals =

Dimension: 1
[0.0, infinity): [1,2] + [0,2] + [0,1]
Dimension: 0
[0.0, infinity): [0]
```

A representative cycle generating the single 0-dimensional homology class is [0], and a representative cycle generating the single 1-dimensional homology class is [1,2] + [0,2] + [0,1].

9-sphere example. Let's build a 9-sphere, which is homeomorphic to the boundary of a 10-simplex. First we add a single 10-simplex to an empty explicit simplex stream. The result is not a simplicial complex because it does not contain the faces of the 10-simplex. We add all faces using the method `ensureAllFaces`. Then, we remove the 10-simplex using the method `removeElementIfPresent`. What remains is the boundary of a 10-simplex, that is, a 9-sphere.

```
>> dimension = 9;
>> stream = api.Plex4.createExplicitSimplexStream();
>> stream.addElement(0:(dimension + 1));
>> stream.ensureAllFaces();
```

```
>> stream.removeElementIfPresent(0:(dimension + 1));
>> stream.finalizeStream();
```

In the above, the `finalizeStream` function is used to ensure that the stream has been fully constructed and is ready for consumption by a persistence algorithm. Note that it can be omitted in the case where the simplex additions are done in increasing order. However, it should be used in general.

We print the total number of simplices in the complex.

```
>> num_simplices = stream.getSize()
num_simplices = 2046
```

We get the persistence algorithm

```
persistence = api.Plex4.getModularSimplicialAlgorithm(dimension + 1, 2);
```

and compute and print the intervals.

```
>> n_sphere_intervals = persistence.computeIntervals(stream)
n_sphere_intervals =

Dimension: 9
[0.0, infinity)
Dimension: 0
[0.0, infinity)
```

This gives us the expected Betti numbers $Betti_0 = 1$ and $Betti_9 = 1$.

Try computing a representative cycle for each barcode.

```
>> n_sphere_intervals = persistence.computeAnnotatedIntervals(stream)
```

We don't display the output from this command in the tutorial, because the representative 9-cycle is very long and contains all eleven 9-simplices.

See Appendix B for exercise solutions.

Exercise 3.1.1. Build a simplicial complex homeomorphic to the torus. Compute its Betti numbers. *Hint: You will need at least 7 vertices* [7, page 107]*. We recommend using a* 3×3 *grid of 9 vertices.*

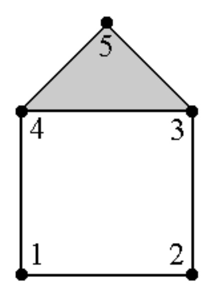
Exercise 3.1.2. Build a simplicial complex homeomorphic to the Klein bottle. Check that it has the same Betti numbers as the torus over $\mathbb{Z}_2$ coefficients but different Betti numbers over $\mathbb{Z}_3$ coefficients.

Exercise 3.1.3. Build a simplicial complex homeomorphic to the projective plane. Find its Betti numbers over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ coefficients.

3.2. **Computing persistent homology.** Let's build a stream with nontrivial filtration times.

House example. The Matlab script corresponding to this section is `house_example.m`.

We build a house, with the vertices and edges on the square appearing at time 0, with the top vertex appearing at time 1, with the roof edges appearing at times 2 and 3, and with the roof 2-simplex appearing at time 7.




```
>> stream = api.Plex4.createExplicitSimplexStream();
>> stream.addVertex(1, 0);
>> stream.addVertex(2, 0);
>> stream.addVertex(3, 0);
```

```
>> stream.addVertex(4, 0);
>> stream.addVertex(5, 1);
>> stream.addElement([1, 2], 0);
>> stream.addElement([2, 3], 0);
>> stream.addElement([3, 4], 0);
>> stream.addElement([4, 1], 0);
>> stream.addElement([3, 5], 2);
>> stream.addElement([4, 5], 3);
>> stream.addElement([3, 4, 5], 7);
>> stream.finalizeStream();
```

We get the persistence algorithm with $\mathbb{Z}_2$ coefficients

```
>> persistence = api.Plex4.getModularSimplicialAlgorithm(3, 2);
```

and compute the intervals.

```
>> intervals = persistence.computeIntervals(stream)
intervals =

Dimension: 1
[3.0, 7.0)
[0.0, infinity)
Dimension: 0
[1.0, 2.0)
[0.0, infinity)
```

There are four intervals. The first is a $Betti_1$ interval, starting at filtration time 3 and ending at 7. This 1-dimensional hole is formed by the three edges of the roof. It forms when edge $[4, 5]$ appears at filtration time 3 and closes when 2-simplex $[3, 4, 5]$ appears at filtration time 7.

We compute a representative cycle for each barcode.

```
>> intervals = persistence.computeAnnotatedIntervals(stream)
intervals =

Dimension: 1
[3.0, 7.0): [4,5] + [3,4] + -[3,5]
[0.0, infinity): [1,4] + [2,3] + [1,2] + [3,4]
Dimension: 0
[1.0, 2.0): -[3] + [5]
[0.0, infinity): [1]
```

One $Betti_0$ interval and one $Betti_1$ interval are semi-infinite.

```
>> infinite_barcodes = intervals.getInfiniteIntervals();
```

We can print the Betti numbers (at the largest filtration time 7) as an array

```
>> betti_numbers_array = infinite_barcodes.getBettiSequence()
betti_numbers_array =
    1
    1
```

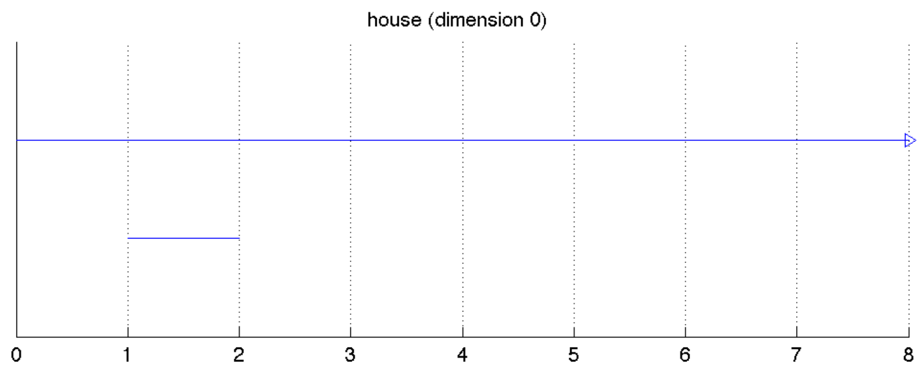
or as a list with entries of the form $k : Betti_k$.

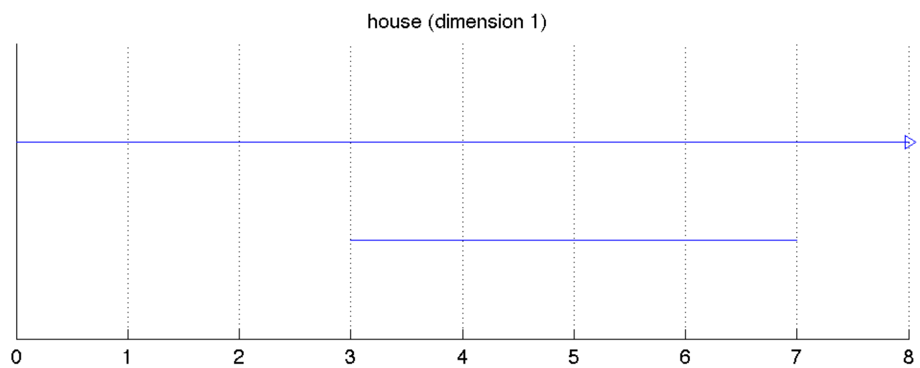
```
>> betti_numbers_string = infinite_barcodes.getBettiNumbers()
betti_numbers_string = {0: 1, 1: 1}
```

The Matlab function `plot_barcodes.m` lets us display the intervals as Betti barcodes. The Matlab structure array `options` contains different options for the plot. We choose the filename `house` and we choose the maximum filtration time for the plot to be eight.

```
>> options.filename = 'house';
>> options.max_filtration_value = 8;
>> plot_barcodes(intervals, options);
```

The file `house.png` is saved to your current directory.





The filtration times are on the horizontal axis. The $Betti_k$ number of the stream at filtration time t is the number of intervals in the dimension k plot that intersect a vertical line through t. Check that the displayed intervals agree with the filtration times we built into the house stream. At time 0, a connected component and a 1-dimensional hole form. At time 1, a second connected component appears, which joins to the first at time 2. A second 1-dimensional hole forms at time 3, and closes at time 7.

An important remark is that the methods `addElement` and `removeElementIfPresent` do not necessarily enforce the definition of a stream. They allow us to build inconsistent complexes in which some simplex $\sigma \in X(t)$ contains a subsimplex $\sigma' \notin X(t)$, meaning that $X(t)$ is not a simplicial complex. The method `validateVerbose` returns `1` if our stream is consistent and returns `0` with explanation if not.

```
>> stream.validateVerbose()
ans = 1
>> stream.addElement([1, 4, 5], 0);
>> stream.validateVerbose()
Filtration index of face [4,5] exceeds that of element [1,4,5] (3 > 0)
Stream does not contain face [1,5] of element [1,4,5]
ans = 0
```

4. Point cloud data

A point cloud is a finite metric space, that is, a finite set of points equipped with a notion of distance. One can create a Euclidean metric space by specifying the coordinates of points in Euclidean space, or one can create an explicit metric space by specifying all pairwise distances between points. In Section 5 we will learn how to build streams from point cloud data.

4.1. **Euclidean metric spaces.** The Matlab script corresponding to this section is `pointcloud_example.m`.

House example. Let's give Euclidean coordinates to the points of our house.

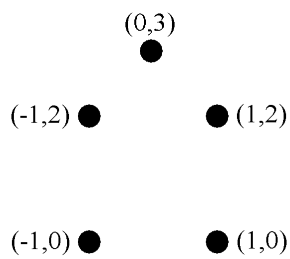


Figure 1. The house point cloud

You can enter these coordinates manually.

```
>> point_cloud = [-1,0; 1,0; 1,2; -1,2; 0,3]
point_cloud =

  -1    0
   1    0
   1    2
  -1    2
   0    3
```

Or, these coordinates are stored as a javaPlex example.

```
>> point_cloud = examples.PointCloudExamples.getHouseExample();
```

We create a metric space using these coordinates. The input to the `EuclideanMetricSpace` method is a matrix whose i-th row lists the coordinates of the i-th point.

```
>> m_space = metric.impl.EuclideanMetricSpace(point_cloud);
```

We can return the coordinates of a specific point. Note the points are indexed starting at 0.

```
>> m_space.getPoint(0)
ans =
  -1
```

```
    0

>> m_space.getPoint(2)
ans =
    1
    2
```

A metric space can return the distance between any two points.

```
>> m_space.distance(m_space.getPoint(0), m_space.getPoint(2))
ans = 2.8284
```

Figure 8 example. We select 1,000 points randomly from a figure eight, that is, the union of unit circles centered at $(0, 1)$ and $(0, -1)$.

```
>> point_cloud = examples.PointCloudExamples.getRandomFigure8Points(1000);
```

We plot the points.

```
>> figure
>> scatter(point_cloud(:,1), point_cloud(:,2), '.')
>> axis equal
```

Torus example. We select 2,000 points randomly from a torus in $\mathbb{R}^3$ with inner radius 1 and outer radius 2. The first input is the number of points, the second input is the inner radius, and the third input is the outer radius

```
>> point_cloud = examples.PointCloudExamples.getRandomTorusPoints(2000, 1, 2);
```

We plot the points.

```
>> figure
>> scatter3(point_cloud(:,1), point_cloud(:,2), point_cloud(:,3), '.')
>> axis equal
```

Sphere product example. We select 1,000 points randomly from the unit torus $S^1 \times S^1$ in $\mathbb{R}^4$. The first input is the number of points, the second input is the dimension of each sphere, and the third input is the number of sphere factors.

```
>> point_cloud = examples.PointCloudExamples.getRandomTorusPoints(1000, 1, 2);
```

Plotting the third and fourth coordinates of each point shows a circle S^1.

```
>> figure
>> scatter(point_cloud(:,3), point_cloud(:,4), '.')
>> axis equal
```

4.2. **Explicit metric spaces.** We can also create a metric space from a distance matrix using the method `ExplicitMetricSpace`. For a point cloud in Euclidean space, this method is generally less convenient than the command `EuclideanMetricSpace`. However, method `ExplicitMetricSpace` can be used for a point cloud in an arbitrary (perhaps non-Euclidean) metric space.

The Matlab script corresponding to this section is `explicit_metric_space_example.m`.

House example. The matrix `distances` summarizes the metric for our house points in Figure 1: entry (i, j) is the distance from point i to point j.

```
>> distances = [0,2,sqrt(8),2,sqrt(10);
2,0,2,sqrt(8),sqrt(10);
sqrt(8),2,0,2,sqrt(2);
2,sqrt(8),2,0,sqrt(2);
sqrt(10),sqrt(10),sqrt(2),sqrt(2),0]

distances =
    0         2.0000    2.8284    2.0000    3.1623
    2.0000    0         2.0000    2.8284    3.1623
    2.8284    2.0000    0         2.0000    1.4142
    2.0000    2.8482    2.0000    0         1.4142
    3.1623    3.1623    1.4142    1.4142    0
```

We create a metric space from this distance matrix.

```
>> m_space = metric.impl.ExplicitMetricSpace(distances);
```

We return the distance between points 0 and 2.

```
>> m_space.distance(0, 2)
ans = 2.8284
```

Exercise 4.2.1. One way to produce a torus is to take a square $[0,1]\times[0,1]$ and then identify opposite sides. This is called a flat torus. More explicitly, the quotient space

$$([0,1]\times[0,1])/\sim,$$

where $(0,y)\sim(1,y)$ for all $y\in[0,1]$ and where $(x,0)\sim(x,1)$ for all $x\in[0,1]$, is a flat torus. The Euclidean metric on $[0,1]\times[0,1]$ induces a metric on the flat torus. For example, in the induced metric on the flat torus, the distance between $(0,\frac{1}{2})$ and $(1,\frac{1}{2})$ is zero, since these two points are identified. The distance between $(\frac{1}{10},\frac{1}{2})$ and $(\frac{9}{10},\frac{1}{2})$ is $\frac{2}{10}$, by passing through the point $(0,\frac{1}{2})\sim(1,\frac{1}{2})$.

Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1]\times[0,1]$ and then computes the 1,000 $\times$ 1,000 distance matrix for these points under the induced metric on the flat torus. Create an explicit metric space from this distance matrix.

This exercise is continued by Exercise 5.4.4.

Exercise 4.2.2. One way to produce a Klein bottle is to take a square $[0,1]\times[0,1]$ and then identify opposite edges, with the left and right sides identified with a twist. This is called a flat Klein bottle. More explicitly, the quotient space

$$([0,1]\times[0,1])/\sim,$$

where $(0,y)\sim(1,1-y)$ for all $y\in[0,1]$ and where $(x,0)\sim(x,1)$ for all $x\in[0,1]$, is a flat Klein bottle. The Euclidean metric on $[0,1]\times[0,1]$ induces a metric on the flat Klein bottle. For example, in the induced metric on the flat Klein bottle, the distance between $(0,\frac{4}{10})$ and $(1,\frac{6}{10})$ is zero, since these two points are identified. The distance between $(\frac{1}{10},\frac{4}{10})$ and $(\frac{9}{10},\frac{6}{10})$ is $\frac{2}{10}$, by passing through the point $(0,\frac{4}{10})\sim(1,\frac{6}{10})$.

Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1]\times[0,1]$ and then computes the 1,000 $\times$ 1,000 distance matrix for these points under the induced metric on the flat Klein bottle. Create an explicit metric space from this distance matrix.

This exercise is continued by Exercise 5.4.5.

5. Streams from point cloud data

In Section 3 we built streams explicitly, or by hand. In this section we construct streams from a point cloud Z. We build Vietoris–Rips, witness, and lazy witness streams. See [4] for additional information.

The Vietoris–Rips, witness, and lazy witness streams all take three of the same inputs: the maximum dimension d_{max}, the maximum filtration time t_{max}, and the number of divisions N. These inputs allow the user to limit the size of the constructed stream, for computational efficiency. No simplices above dimension d_{max} are included. The persistent homology of the resulting stream can be calculated only up to dimension $d_{max} - 1$ (do you see why?). Also, instead of computing complex $X(t)$ for all $t \geq 0$, we only compute $X(t)$ for

$$t \in \left\{0,\ \frac{t_{max}}{N-1},\ \frac{2t_{max}}{N-1},\ \frac{3t_{max}}{N-1},\ \ldots,\ \frac{(N-2)t_{max}}{N-1},\ t_{max}\right\}.$$

The number of divisions N is an optional input. If this input parameter is not specified, then the default value $N = 20$ is used.

When working with a new dataset, don't choose d_{max} and t_{max} too large initially. First get a feel for how fast the simplicial complexes are growing, and then raise d_{max} and t_{max} nearer to the computational limits. If you ever choose d_{max} or t_{max} too large and Matlab seems to be running forever, pressing the `control` and `c` buttons simultaneously may halt the computation.

5.1. **Vietoris–Rips streams.** Let $d(\,\cdot\,,\,\cdot\,)$ denote the distance between two points in metric space Z. A natural stream to build is the Vietoris–Rips stream. The complex $\mathrm{VR}(Z, t)$ is defined as follows:

- the vertex set is Z.
- for vertices a and b, edge $[ab]$ is in $\mathrm{VR}(Z, t)$ if $d(a, b) \leq t$.
- a higher dimensional simplex is in $\mathrm{VR}(Z, t)$ if all of its edges are.

Note that $\mathrm{VR}(Z, t) \subset \mathrm{VR}(Z, s)$ whenever $t \leq s$, so the Vietoris–Rips stream is a filtered simplicial complex. Since a Vietoris–Rips complex is the maximal simplicial complex that can be built on top of its 1-skeleton, it is a *flag complex*.

The Matlab script corresponding to this section is `rips_example.m`.

House example. Let's build a Vietoris–Rips stream from the house point cloud in Section 4.1. Note this stream is different than the explicit house stream we built in Section 3.2.

```
>> max_dimension = 3;
>> max_filtration_value = 4;
>> num_divisions = 100;

>> point_cloud = examples.PointCloudExamples.getHouseExample();
>> stream = api.Plex4.createVietorisRipsStream(point_cloud, max_dimension,
max_filtration_value, num_divisions);
```

The order of the inputs is `createVietorisRipsStream(`Z, d_{max}, t_{max}, N`)`. For a Vietoris–Rips stream, the parameter t_{max} is the maximum possible edge length. Since $t_{max} = 4$ is greater than the diameter ($\sqrt{10}$) of our point cloud, all edges will eventually form.

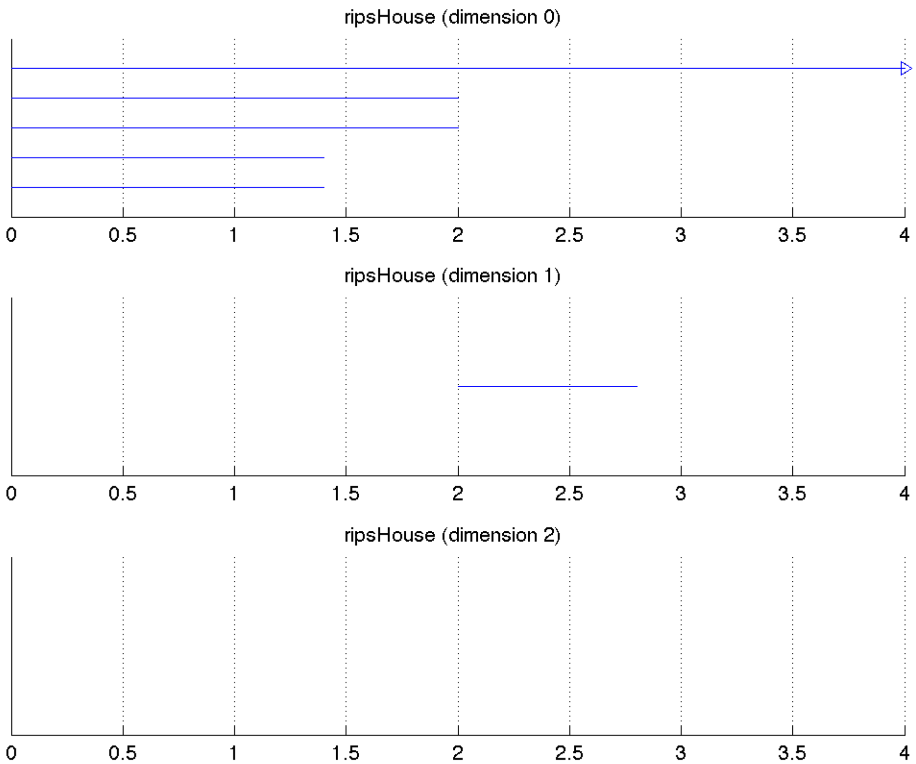
Since $d_{max} = 3$ we can compute up to second dimensional persistent homology.

```
>> persistence = api.Plex4.getModularSimplicialAlgorithm(max_dimension, 2);
>> intervals = persistence.computeIntervals(stream);
```

We display the Betti intervals. Parameter `options.max_filtration_value` is the largest filtration time to be displayed. Typically `options.max_filtration_value` is chosen to be `max_filtration_value`. Parameter `options.max_dimension` is the largest persistent homology dimension to be displayed. Typically `options.max_dimension` is chosen to be `max_dimension - 1` because in a stream with simplices computed up to dimension d_{max} we can only compute persistent homology up to dimension $d_{max} - 1$.

```
>> options.filename = 'ripsHouse';
>> options.max_filtration_value = max_filtration_value;
>> options.max_dimension = max_dimension - 1;
>> plot_barcodes(intervals, options);
```

The file `ripsHouse.png` is saved to your current directory.



Check that these plots are consistent with the Vietoris–Rips definition: edges $[3,5]$ and $[4,5]$ appear at filtration time $t = \sqrt{2}$; the square appears at $t = 2$; the square closes at $t = \sqrt{8}$.

Torus example. Try the following sequence of commands. We start with 400 points from a 20×20 grid on the unit torus $S^1 \times S^1$ in $\mathbb{R}^4$, and add a small amount of noise to each point. We build the Vietoris–Rips stream.

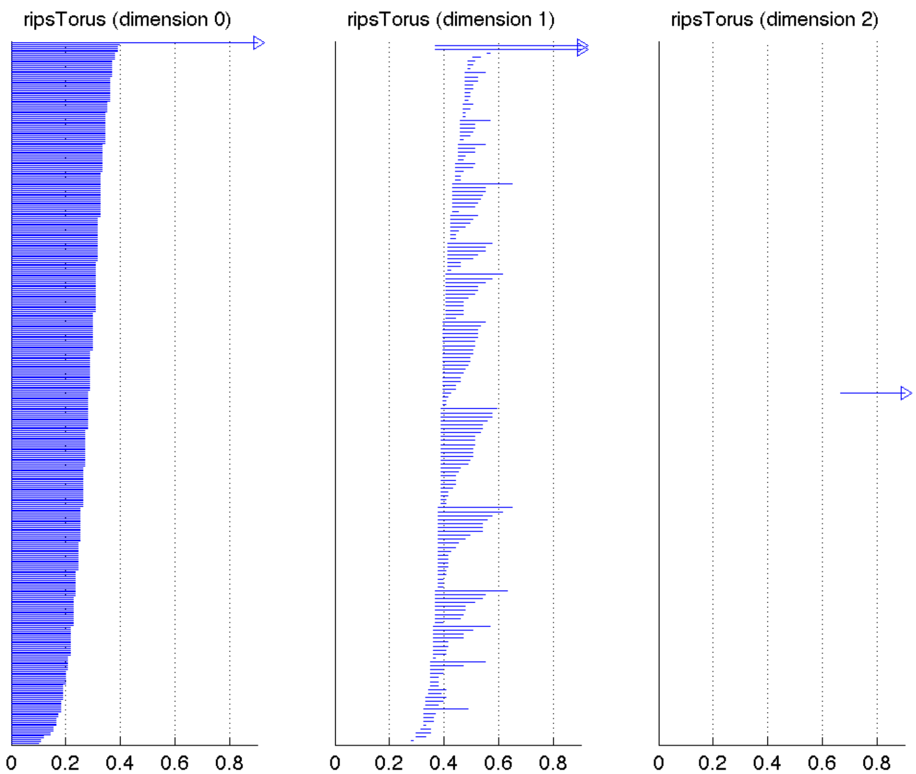
```
>> max_dimension = 3;
>> max_filtration_value = 0.9;
>> num_divisions = 100;
```

Load the file `pointsTorusGrid.mat`. The matrix `pointsTorusGrid` appears in your Matlab workspace.

```
>> load pointsTorusGrid.mat
>> point_cloud = pointsTorusGrid;
>> size(point_cloud)
ans = 400    4                              % 400 points in dimension 4

>> stream = api.Plex4.createVietorisRipsStream(point_cloud, max_dimension,
```

```
max_filtration_value, num_divisions);
>> num_simplices = stream.getSize()
num_simplices = 82479

>> persistence = api.Plex4.getModularSimplicialAlgorithm(max_dimension, 2);
>> intervals = persistence.computeIntervals(stream);

>> options.filename = 'ripsTorus';
>> options.max_filtration_value = max_filtration_value;
>> options.max_dimension = max_dimension - 1;
>> options.side_by_side = true;
>> plot_barcodes(intervals, options);
```

Setting the parameter `options.side_by_side` equal to `true` makes it such that the Betti barcodes of different dimensions are plotted side by side instead of above and below each other. The file `ripsTorus.png` is saved to your current directory.



The diameter of this torus (before adding noise) is $\sqrt{8}$, so choosing $t_{max} = 0.9$ likely will not show all homological activity. However, the torus will be reasonably connected by this time. Note the semi-infinite intervals match the correct numbers $Betti_0 = 1$, $Betti_1 = 2$, $Betti_2 = 1$ for a torus.

```
>> infinite_barcodes = intervals.getInfiniteIntervals();
>> betti_numbers_array = infinite_barcodes.getBettiSequence()
betti_numbers_array =
     1
```

```
2
1
```

This example makes it clear that the computed "semi-infinite" intervals do not necessarily persist until $t = \infty$: in a Vietoris–Rips stream, once t is greater than the diameter of the point cloud, the Betti numbers for $\mathrm{VR}(Z, t)$ will be $Betti_0 = 1$, $Betti_1 = Betti_2 = ... = 0$. The computed semi-infinite intervals are merely those that persist until $t = t_{max}$.

Remark. We can build Vietoris–Rips streams not only on top of Euclidean point clouds, but also on top of explicit metric spaces. For example, if `m_space` were an explicit metric space, then we could call a command such as

```
>> stream = api.Plex4.createVietorisRipsStream(m_space, max_dimension,
max_filtration_value, num_divisions);
```

Exercise 5.1.1. Slowly increase the values for t_{max}, d_{max} and note how quickly the size of the Vietoris–Rips stream and the time of computation grow. Either increasing t_{max} from 0.9 to 1 or increasing d_{max} from 3 to 4 roughly doubles the size of the Vietoris–Rips stream.

Exercise 5.1.2. Find a planar dataset $Z \subset \mathbb{R}^2$ and a filtration value t such that $\mathrm{VR}(Z, t)$ has nonzero $Betti_2$. Build a Vietoris–Rips stream to confirm your answer.

Exercise 5.1.3. Find a planar dataset $Z \subset \mathbb{R}^2$ and a filtration value t such that $\mathrm{VR}(Z, t)$ has nonzero $Betti_6$. When building a Vietoris–Rips stream to confirm your answer, don't forget to choose $d_{max} = 7$.

5.2. **Landmark selection.** For larger datasets, if we include every data point as a vertex, as in the Vietoris–Rips construction, our streams will quickly contain too many simplices for efficient computation. The witness stream and the lazy witness stream address this problem. In building these streams, we select a subset $L \subset Z$, called landmark points, as the only vertices. All data points in Z help serve as witnesses for the inclusion of higher dimensional simplices.

There are two common methods for selecting landmark points. The first is to choose the landmarks L randomly from point cloud Z. The second is a greedy inductive selection process called sequential maxmin. In sequential maxmin, the first landmark is picked randomly from Z. Inductively, if L_{i-1} is the set of the first $i-1$ landmarks, then let the i-th landmark be the point of Z which maximizes the function $z \mapsto d(z, L_{i-1})$, where $d(z, L_{i-1})$ is the distance between the point z and the set L_{i-1}.

Landmarks chosen using sequential maxmin tend to cover the dataset and to be spread apart from each other. A disadvantage is that outlier points tend to be selected. Sequential maxmin landmarks are used in [1] and [3].

The Matlab script corresponding to this section is `landmark_example.m`.

Figure 8 example. We create a point cloud of 1,000 points from a figure eight.

```
>> point_cloud = examples.PointCloudExamples.getRandomFigure8Points(1000);
```

We create both a random landmark selector and a sequential maxmin landmark selector. These selectors will pick 100 landmarks each.

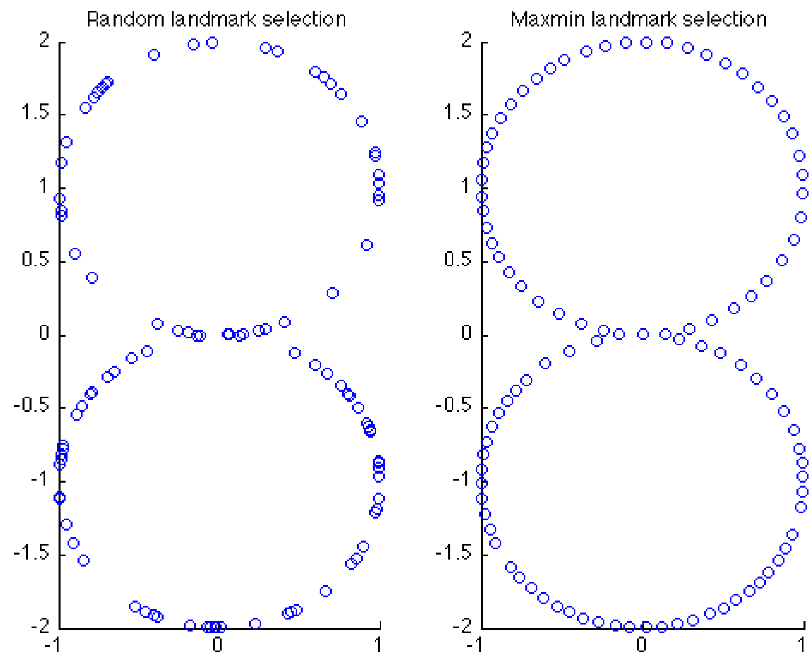
```
>> num_landmark_points = 100;
>> random_selector = api.Plex4.createRandomSelector(point_cloud, num_landmark_points);
>> maxmin_selector = api.Plex4.createMaxMinSelector(point_cloud, num_landmark_points);
```

We select 100 random landmarks and 100 landmarks via sequential maxmin. Note we need to increment the indices by 1 since Java uses 0-based arrays.

```
>> random_points = point_cloud(random_selector.getLandmarkPoints() + 1, :);
>> maxmin_points = point_cloud(maxmin_selector.getLandmarkPoints() + 1, :);
```

We plot the two sets of landmark points to see the difference between random and sequential maxmin landmark selection.

```
>> subplot(1, 2, 1);
>> scatter(random_points(:,1), random_points(:, 2));
>> title('Random landmark selection');
>> subplot(1, 2, 2);
>> scatter(maxmin_points(:,1), maxmin_points(:, 2));
>> title('Maxmin landmark selection');
```



Sequential maxmin seems to do a better job of choosing landmarks that cover the figure eight and that are spread apart.

Remark. We can select landmark points not only from Euclidean point clouds but also from explicit metric spaces. For example, if `m_space` is an explicit metric space, then we may select landmarks using a command such as the following.

```
>> maxmin_selector = api.Plex4.createMaxMinSelector(m_space, num_landmark_points);
```

Given point cloud Z and landmark subset L, we define $\mathtt{R} = \max_{z\in Z}\big\{d(z,L)\big\}$. Number $\mathtt{R}$ reflects how finely the landmarks cover the dataset. We often use it as a guide for selecting the maximum filtration value t_{max} for a witness or lazy witness stream.

Exercise 5.2.1. Let Z be the point cloud in Figure 1 from Section 4.1, corresponding to the house point cloud. Suppose we are using sequential maxmin to select a set L of 3 landmarks, and the first (randomly selected) landmark is $(1,0)$. Find by hand the other two landmarks in L.

Exercise 5.2.2. Let Z be a point cloud and L a landmark subset. Show that if L is chosen via sequential maxmin, then for any $l_i, l_j \in L$, we have $d(l_i, l_j) \geq \mathtt{R}$.

5.3. **Witness streams.** Suppose we are given a point cloud Z and landmark subset L. Let $m_k(z)$ be the distance from a point $z \in Z$ to its $(k+1)$-th closest landmark point. The witness stream complex $\mathrm{W}(Z, L, t)$ is defined as follows.

- the vertex set is L.
- for $k > 0$ and vertices l_i, the k-simplex $[l_0 l_1 ... l_k]$ is in $\mathrm{W}(Z, L, t)$ if all of its faces are, and if there exists a witness point $z \in Z$ such that

$$\max\big\{d(l_0, z), d(l_1, z), ..., d(l_k, z)\big\} \leq t + m_k(z).$$

Note that $\mathrm{W}(Z, L, t) \subset \mathrm{W}(Z, L, s)$ whenever $t \leq s$. Note that a landmark point can serve as a witness point.

Exercise 5.3.1. Let Z be the point cloud in Figure 1 from Section 4.1, corresponding to the house point cloud. Let $L = \{(1, 0), (0, 3), (-1, 0)\}$ be the landmark subset. Find by hand the filtration time for the edge between vertices $(1, 0)$ and $(0, 3)$. Which point or points witness this edge? What is the filtration time for the lone 2-simplex $[(1, 0), (0, 3), (-1, 0)]$?

The Matlab script corresponding to this section is `witness_example.m`.

Torus example. Let's build a witness stream instance for 10,000 random points from the unit torus $S^1 \times S^1$ in $\mathbb{R}^4$, with 50 random landmarks.

```
>> num_points = 10000;
>> num_landmark_points = 50;
>> max_dimension = 3;
>> num_divisions = 100;

>> point_cloud = examples.PointCloudExamples.getRandomSphereProductPoints(num_points,
1, 2);
>> landmark_selector = api.Plex4.createRandomSelector(point_cloud, num_landmark_points);
```

The next command returns the landmark covering measure `R` from Section 5.2. Often the value for t_{max} is chosen in proportion to `R`.

```
>> R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
R = 1.1067 \hspace{40mm}                    % Generally close to 1.1
>> max_filtration_value = R / 8;
```

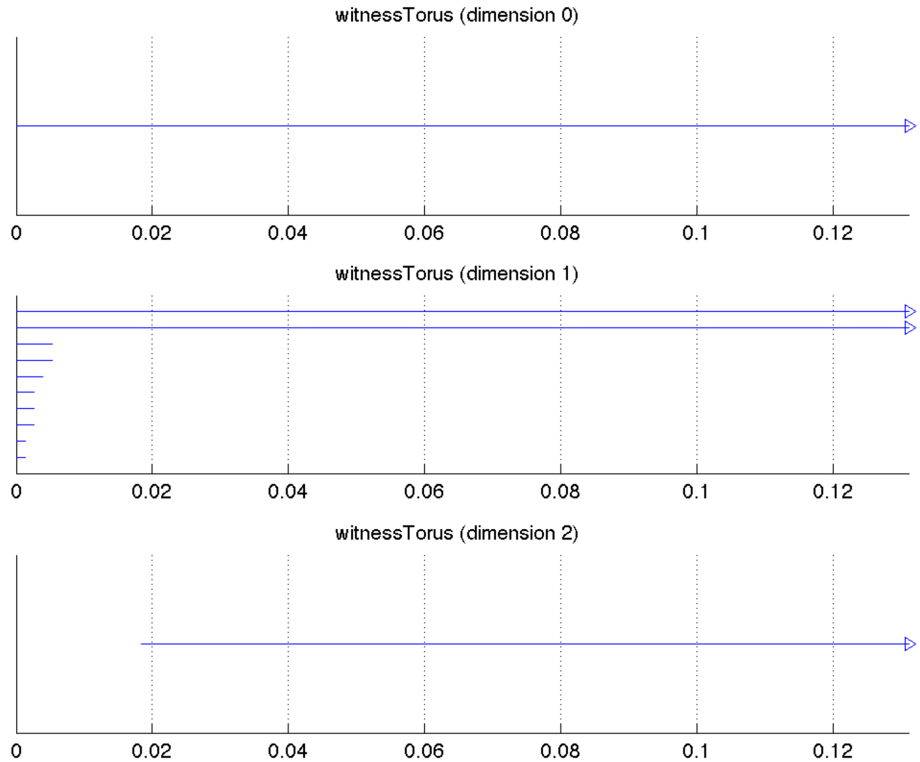
We create the witness stream.

```
>> stream = api.Plex4.createWitnessStream(landmark_selector, max_dimension,
max_filtration_value, num_divisions);
>> num_simplices = stream.getSize()
num_simplices = 1858                % Generally close to 2000
```

We plot the Betti intervals.

```
>> persistence = api.Plex4.getModularSimplicialAlgorithm(max_dimension, 2);
>> intervals = persistence.computeIntervals(stream);

>> options.filename = 'witnessTorus';
>> options.max_filtration_value = max_filtration_value;
>> options.max_dimension = max_dimension - 1;
>> plot_barcodes(intervals, options);
```

The file `witnessTorus.png` is saved to your current directory.



The idea of persistent homology is that long intervals should correspond to real topological features, whereas short intervals are considered to be noise. The plot above shows that for a long range, the torus numbers $Betti_0 = 1$, $Betti_1 = 2$, $Betti_2 = 1$ are obtained. Your plot should contain a similar range.

The witness stream above contains approximately 2,000 simplices, fewer than the approximately 80,000 simplices in the Vietoris–Rips stream from the torus example in Section 5.1. This is despite the fact that we started with a point cloud of 100,000 points in the witness case, but of only 400 points in the Vietoris–Rips case. This supports our belief that the witness stream returns good results at lower computational expense.

5.4. **Lazy witness streams.** A lazy witness stream is similar to a witness stream. However, there is an extra parameter ν, typically chosen to be 0, 1, or 2, which helps determine how the lazy witness complexes $\mathrm{LW}_\nu(Z, L, t)$ are constructed. See [4] for more information.

Suppose we are given a point cloud Z, landmark subset L, and parameter $\nu \in \mathbb{N}$. If $\nu = 0$, let $m(z) = 0$ for all $z \in Z$. If $\nu > 0$, let $m(z)$ be the distance from z to the ν-th closest landmark point. The lazy witness complex $\mathrm{LW}_\nu(Z, L, t)$ is defined as follows.

- the vertex set is L.
- for vertices a and b, edge $[ab]$ is in $\mathrm{LW}_\nu(Z, L, t)$ if there exists a witness $z \in Z$ such that

$$\max\{d(a, z), d(b, z)\} \leq t + m(z).$$

- a higher dimensional simplex is in $\mathrm{LW}_\nu(Z, L, t)$ if all of its edges are.

Note that $\mathrm{LW}_\nu(Z, L, t) \subset \mathrm{LW}_\nu(Z, L, s)$ whenever $t \leq s$. The adjective *lazy* refers to the fact that the lazy witness complex is a flag complex: since the 1-skeleton determines all higher dimensional simplices, less computation is involved.

Exercise 5.4.1. Let Z be the point cloud in Figure 1 from Section 4.1, corresponding to the house point cloud. Let $L = \{(1, 0), (0, 3), (-1, 0)\}$ be the landmark subset. Let $\nu = 1$. Find by hand the filtration time for the edge between vertices $(1, 0)$ and $(0, 3)$. Which point or points witness this edge? What is the filtration time for the lone 2-simplex $[(1, 0), (0, 3), (-1, 0)]$?

Exercise 5.4.2. Repeat the above exercise with $\nu = 0$ and with $\nu = 2$.

Exercise 5.4.3. Check that the 1-skeleton of a witness complex $\mathrm{W}(Z, L, t)$ is the same as the 1-skeleton of a lazy witness complex $\mathrm{LW}_2(Z, L, t)$. As a consequence, $\mathrm{LW}_2(Z, L, t)$ is the flag complex of $\mathrm{W}(Z, L, t)$.

2-sphere example. The Matlab script corresponding to this example is `lazy_witness_example.m`.

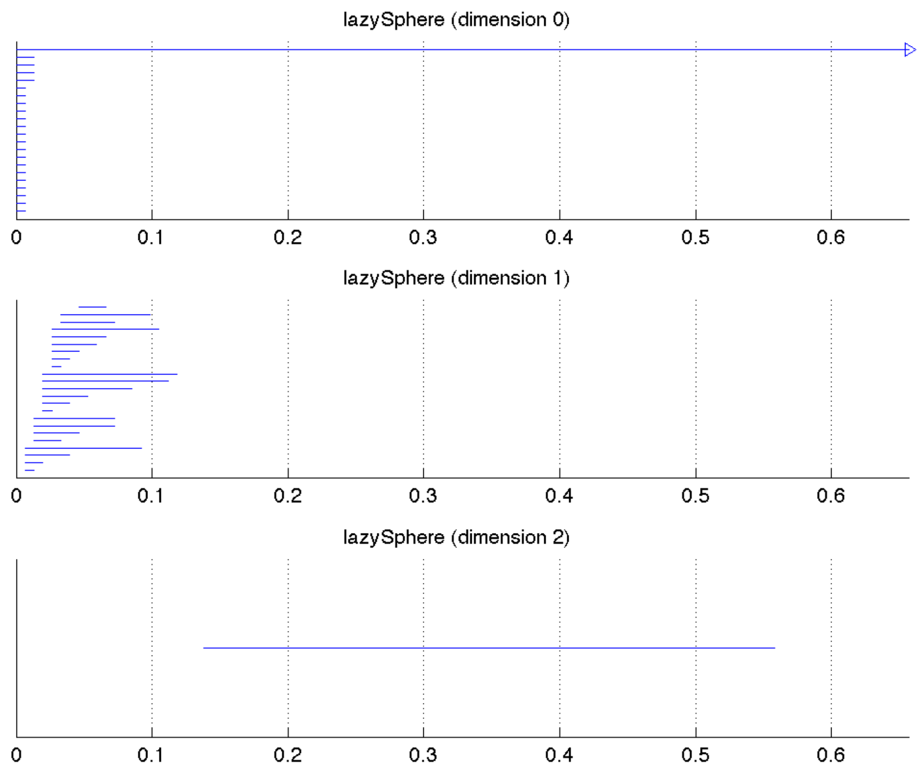
We use parameter $\nu = 1$.

```
>> max_dimension = 3;
>> num_points = 1000;
>> num_landmark_points = 50;
>> nu = 1;
>> num_divisions = 100;

>> point_cloud = examples.PointCloudExamples.getRandomSpherePoints(num_points,
max_dimension - 1);
>> landmark_selector = api.Plex4.createRandomSelector(point_cloud, num_landmark_points);
```

Often t_{max} is chosen in proportion to `R`.

```
>> R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
R = 0.6877                                  % Generally close to 0.7
>> max_filtration_value = R;
>> stream = streams.impl.LazyWitnessStream(landmark_selector.getUnderlyingMetricSpace(),
landmark_selector, max_dimension, max_filtration_value, nu, num_divisions);
>> stream.finalizeStream()
>> num_simplices = stream.getSize()
num_simplices = 79842                       % Generally between 30000 and 180000
>> persistence = api.Plex4.getModularSimplicialAlgorithm(max_dimension, 2);
>> intervals = persistence.computeIntervals(stream);

>> options.filename = 'lazySphere';
>> options.max_filtration_value = max_filtration_value;
>> options.max_dimension = max_dimension - 1;
>> plot_barcodes(intervals, options);
```

The file `lazySphere.png` is saved to your current directory.



Exercise 5.4.4. The first part of this exercise is the same as Exercise 4.2.1.

One way to produce a torus is to take a square $[0,1] \times [0,1]$ and then identify opposite sides. This is called a flat torus. More explicitly, the quotient space

$$([0,1] \times [0,1])/\sim,$$

where $(0, y) \sim (1, y)$ for all $y \in [0,1]$ and where $(x, 0) \sim (x, 1)$ for all $x \in [0,1]$, is a flat torus. The Euclidean metric on $[0,1] \times [0,1]$ induces a metric on the flat torus. For example, in the induced metric on the flat torus, the distance between $(0, \frac{1}{2})$ and $(1, \frac{1}{2})$ is zero, since these two points are identified. The distance between $(\frac{1}{10}, \frac{1}{2})$ and $(\frac{9}{10}, \frac{1}{2})$ is $\frac{2}{10}$, by passing through the point $(0, \frac{1}{2}) \sim (1, \frac{1}{2})$.

Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1] \times [0,1]$ and then computes the $1{,}000 \times 1{,}000$ distance matrix for these points under the induced metric on the flat torus. Create an explicit metric space from this distance matrix.

Build a lazy witness stream on this explicit metric space with 50 landmarks chosen via sequential maxmin and with $\nu = 1$. Confirm the barcodes match the homology of a torus.

Exercise 5.4.5. The first part of this exercise is the same as Exercise 4.2.1.

One way to produce a Klein bottle is to take a square $[0,1] \times [0,1]$ and then identify opposite edges, with the left and right sides identified with a twist. This is called a flat Klein bottle. More explicitly, the quotient space

$$([0,1] \times [0,1])/\sim,$$

where $(0, y) \sim (1, 1-y)$ for all $y \in [0,1]$ and where $(x, 0) \sim (x, 1)$ for all $x \in [0,1]$, is a flat Klein bottle. The Euclidean metric on $[0,1] \times [0,1]$ induces a metric on the flat Klein bottle. For example, in the induced metric on the flat Klein bottle, the distance between $(0, \frac{4}{10})$ and $(1, \frac{6}{10})$ is zero, since these two points are identified. The distance between $(\frac{1}{10}, \frac{4}{10})$ and $(\frac{9}{10}, \frac{6}{10})$ is $\frac{2}{10}$, by passing through the point $(0, \frac{4}{10}) \sim (1, \frac{6}{10})$.

Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1] \times [0,1]$ and then computes the $1{,}000 \times 1{,}000$ distance matrix for these points under the induced metric on the flat Klein bottle. Create an explicit metric space from this distance matrix.

Build a lazy witness stream on this explicit metric space with 50 landmarks chosen via sequential maxmin and with $\nu = 1$. Confirm the barcodes match the homology of a Klein bottle, over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ coefficients.

6. Example with real data

We now do an example with real data. The corresponding Matlab script is `image_patch_example.m`, and it relies on the files `pointsRange.mat` and `dct.m`.

In *On the nonlinear statistics of range image patches* [1], we study a space of range image patches drawn from the Brown database [8]. A range image is like an optical image, except that each pixel contains a distance instead of a grayscale value. Our space contains high-contrast, normalized, 5×5 pixel patches. We write each 5×5 patch as a vector with 25 coordinates and think of our patches as point cloud data in $\mathbb{R}^{25}$. We select from this space the 30% densest vectors, based on a density estimator called ρ_{300} (see Appendix A). In [1] this dense core subset is denoted $X^5(300, 30)$, and it contains 15,000 points. In the next example we verify a result from [1]: $X^5(300, 30)$ has the topology of a circle.

Load the file `pointsRange.mat`. The matrix `pointsRange` appears in your Matlab workspace.

```
>> load pointsRange.mat
>> size(pointsRange)
ans = 15000    25                          % 15000 points in dimension 25
```

Matrix `pointsRange` is in fact $X^5(300, 30)$: each of its rows is a vector in $\mathbb{R}^{25}$. Display some of the coordinates of `pointsRange`. It is not easy to visualize a circle by looking at these coordinates!

We pick 50 sequential maxmin landmark points, we find the value of `R`, and we build the lazy witness stream with parameter $\nu = 1$.

```
>> max_dimension = 3;
>> num_landmark_points = 50;
>> nu = 1;
>> num_divisions = 500;

>> landmark_selector = api.Plex4.createMaxMinSelector(pointsRange, num_landmark_points);
>> R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
R = 0.7759                             % Generally close to 0.75
>> max_filtration_value = R / 3;
>> stream = streams.impl.LazyWitnessStream(landmark_selector.getUnderlyingMetricSpace(),
landmark_selector, max_dimension, max_filtration_value, nu, num_divisions);
>> stream.finalizeStream()
>> num_simplices = stream.getSize()
num_simplices = 12036                        % Generally between 10000 and 25000

>> persistence = api.Plex4.getModularSimplicialAlgorithm(max_dimension, 2);
>> intervals = persistence.computeIntervals(stream);

>> options.filename = 'lazyRange';
```

```
>> options.max_filtration_value = max_filtration_value;
>> options.max_dimension = max_dimension - 1;
>> plot_barcodes(intervals, options);
```

The file `lazyRange.png` is saved to your current directory.

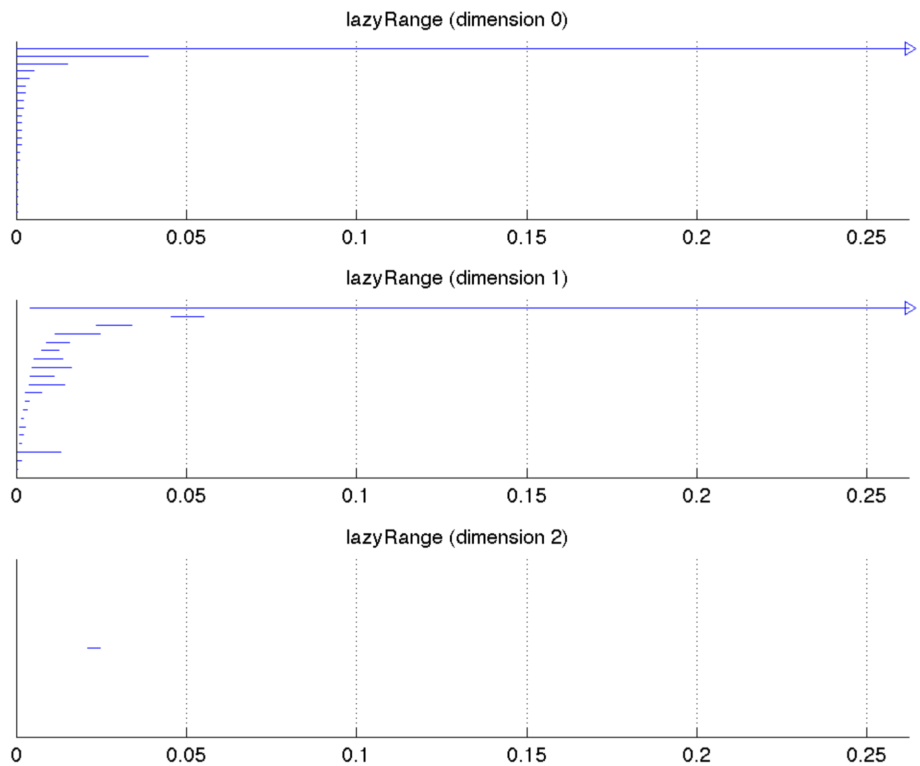


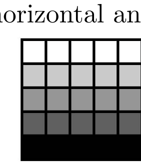
FIGURE 2. Betti intervals for the lazy witness complex built from $X^5(300, 30)$

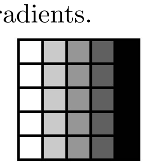
The plots above show that for a long range, the circle Betti numbers $Betti_0 = Betti_1 = 1$ are obtained. Your plot should contain a similar range. This is good evidence that the core subset $X^5(300, 30)$ is well-approximated by a circle.

Our 5×5 normalized patches are currently in the pixel basis: every coordinate corresponds to the range value at one of the 25 pixels. The Discrete Cosine Transform (DCT) basis is a useful basis for our patches [1, 8]. We change to this basis in order to plot a projection of the loop evidenced by Figure 2. The method `dct.m` returns the DCT change-of-basis matrix for square patches of size specified by the input parameter.

```
>> pointsRangeDct = pointsRange * dct(5);
```

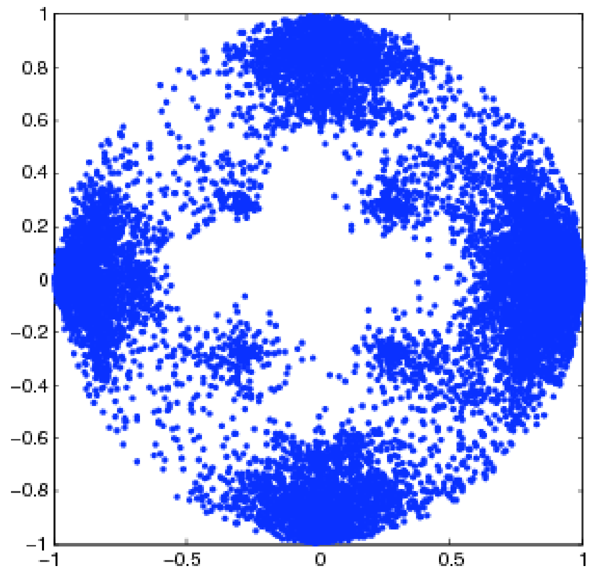
Two of the DCT basis vectors are horizontal and linear gradients.



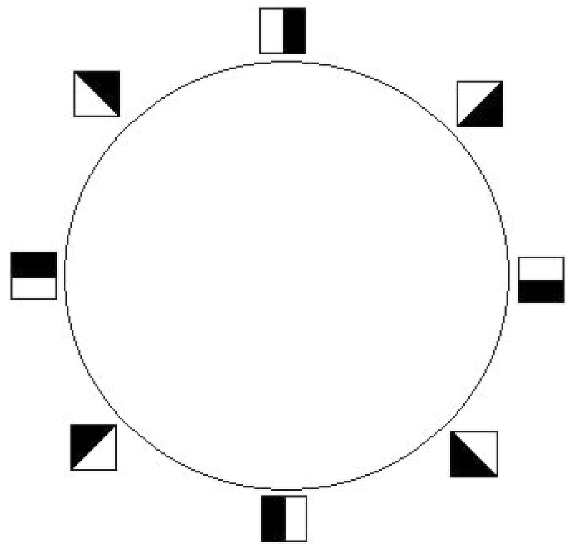


We plot the projection of `pointsRangeDct` onto the linear gradient DCT basis vectors.

```
>> scatter(pointsRangeDct(:,1), pointsRangeDct(:,5), '.')
>> axis square
```



(a) Projection of $X^5(300, 30)$



(b) Range primary circle

The projection of $X^5(300, 30)$ in Figure (a) shows a circle. It is called the range primary circle and is parameterized as shown in Figure (b).

7. Remarks

7.1. **Matlab functions with javaPlex commands.** Writing Matlab functions is very useful. In order to include javaPlex commands in an m-file function, include the command `import edu.stanford.math.plex4.*;` as the second line of the function — that is, as the first line underneath the function header. We include the m-file `eulerCharacteristic.m` as an example Matlab function.

Euler characteristic example. The corresponding Matlab script is `euler_characteristic_example.m`, and it relies on the file `eulerCharacteristic.m`.

First we create a 6-dimensional sphere.

```
>> dimension = 6;
>> stream = api.Plex4.createExplicitSimplexStream();
>> stream.addElement(0:(dimension + 1));
>> stream.ensureAllFaces();
>> stream.removeElementIfPresent(0:(dimension + 1));
>> stream.finalizeStream();
```

The function `eulerCharacteristic.m` accepts an explicit simplex stream and its dimension as input. The function demonstrates two different methods for computing the Euler characteristic.

```
>> eulerCharacteristic(stream, dimension)
The Euler characteristic is 2 = 8 - 28 + 56 - 70 + 56 - 28 + 8, using the alternating
sum of cells.
The Euler characteristic is 2 = 1 - 0 + 0 - 0 + 0 - 0 + 1, using the alternating sum
of Betti numbers.
```

7.2. **Representative cycles.** The persistence algorithm that computes barcodes can also find a representative cycle for each barcode. We do several examples of this in Section 3. However, there is no guarantee that the produced representative will be geometrically nice.

Appendices

Appendix A. Dense core subsets

A core subset of a dataset is a collection of the densest points, such as $X^5(300, 30)$ in Section 6. Since there are many density estimators, and since we can choose any number of the densest points, a dataset has a variety of core subsets. In this appendix we discuss how to create core subsets.

Real datasets can be very noisy, and outlier points can signicantly alter the computed topology. Therefore, instead of trying to approximate the topology of an entire dataset, we often proceed as follows. We create a family of core subsets and identify their topologies. Looking at a variety of core subsets can give a good picture of the entire dataset.

See [3, 4] for an example using multiple core subsets. The dataset contains high-contrast patches from natural images. The authors use three density estimators. As they change from the most global to the most local density estimate, the topologies of the core subsets change from a circle, to three intersecting circles, to a Klein bottle.

One way to estimate the density of a point z in a point cloud Z is as follows. Let $\rho_k(z)$ be the distance from z to its k-th closest neighbor. Let the density estimate at z be $\frac{1}{\rho_k(z)}$. Varying parameter k gives a family of density estimates. Using a small value for k gives a local density estimate, and using a larger value for k gives a more global estimate.

For Euclidean datasets, one can use the m-file `kDensitySlow.m` to produce density estimates $\frac{1}{\rho_k}$. The following command is typical.

```
>> densities = kDensitySlow(points, k);
```

Input `points` is an $N \times n$ matrix of N points in $\mathbb{R}^n$. Input k is the density estimate parameter. Output `densities` is a vertical vertex of length N containing the density estimate at each point.

M-file `coreSubset.m` builds a core subset. The following command is typical.

```
>> core = coreSubset(points, densities, numPoints);
```

Inputs `points` and `densities` are as above. Output `core` is a `numPoints` $\times$ n matrix representing the `numPoints` densest points.

Prime numbers example. The Matlab script corresponding to this example is `core_subsets_example.m`.

The command `primes(3571)` returns a vector listing all prime numbers less than or equal to 3571, which is the 500-th prime. We think of these primes as points in $\mathbb{R}$ and build the core subset of the 10 densest points with density parameter $k = 1$.

```
>> p = primes(3571)';
>> length(p)
ans = 500
>> densities1 = kDensitySlow(p, 1);
>> core1 = coreSubset(p, densities1, 10)
```

```
core1 =
    2
    3
    5
    7
    11
    13
    17
    19
    29
    31
```

We get a bunch of twin primes, which makes sense since $k = 1$. Let's repeat with $k = 50$.

```
>> densities50 = kDensitySlow(p, 50);
>> core50 = coreSubset(p, densities50, 10)
core50 =
    113
    127
    109
    131
    107
    137
    139
    157
    149
    151
```

With $k = 50$, we expect the densest points to be slightly larger than the 25-th prime, which is 97.

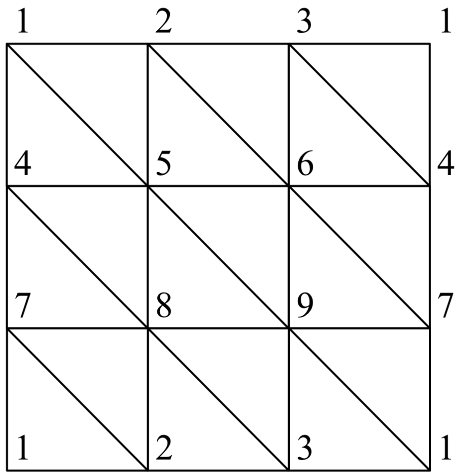
Note: As its name suggests, the m-file `kDensitySlow.m` is not the most efficient way to calculate ρ_k for large datasets. There is a faster file `kDensity.m` for this purpose, which uses the kd-tree data structure. It is not included in the tutorial because it requires one to download a kd-tree package for Matlab, available at `http://www.mathworks.com/matlabcentral/fileexchange/21512-kd-tree-for-matlab`. Please email Henry at `henrya@math.stanford.edu` if you're interested in using `kDensity.m`.

Appendix B. Exercise solutions

Several exercise solutions are accompanied by Matlab scripts, which are available in the folder `tutorial_solutions`.

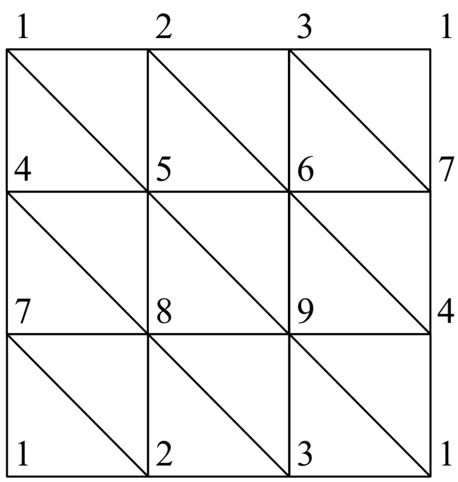
Exercise (3.1.1). Build a simplicial complex homeomorphic to the torus. Compute its Betti numbers. *Hint:* You will need at least 7 vertices [7, page 107]. We recommend using a 3×3 grid of 9 vertices.

Solution. See the Matlab script `exercise_3_1_1.m` in folder `tutorial_solutions` for a solution. We use 9 vertices, which we think of as a 3×3 grid numbered as a telephone keypad. We identify opposite sides.



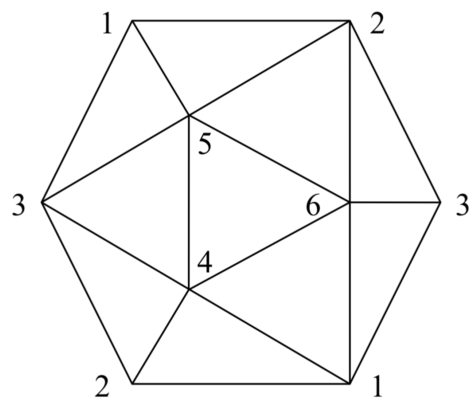
Exercise (3.1.2). Build a simplicial complex homeomorphic to the Klein bottle. Check that it has the same Betti numbers as the torus over $\mathbb{Z}_2$ coefficients but different Betti numbers over $\mathbb{Z}_3$ coefficients.

Solution. See the Matlab script `exercise_3_1_2.m` for a solution. We use 9 vertices, which we think of as a 3×3 grid numbered as a telephone keypad. We identify opposite sides, with left and right sides identified with a twist.



Exercise (3.1.3). Build a simplicial complex homeomorphic to the projective plane. Find its Betti numbers over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ coefficients.

Solution. See the Matlab script `exercise_3_1_3.m` for a solution. We use the minimal triangulation for the projective plane, which contains 6 vertices.



Exercise (4.2.1). Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1] \times [0,1]$ and then computes the 1,000 $\times$ 1,000 distance matrix for these points under the induced metric on the flat torus. Create an explicit metric space from this distance matrix.

Solution. See the Matlab script `exercise_4_2_1.m` and the Matlab function `flatTorusDistanceMatrix.m` for a solution.

Exercise (4.2.2). Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1] \times [0,1]$ and then computes the 1,000 $\times$ 1,000 distance matrix for these points under the induced metric on the flat Klein bottle. Create an explicit metric space from this distance matrix.

Solution. See the Matlab script `exercise_4_2_2.m` and the Matlab function `flatKleinDistanceMatrix.m` for a solution.

Exercise (5.1.1). Slowly increase the values for t_{max}, d_{max} and note how quickly the size of the Vietoris–Rips stream and the time of computation grow. Either increasing t_{max} from 0.9 to 1 or increasing d_{max} from 3 to 4 roughly doubles the size of the Vietoris–Rips stream.

Solution. No solution included.

Exercise (5.1.2). Find a planar dataset $Z \subset \mathbb{R}^2$ and a filtration value t such that $\mathrm{VR}(Z,t)$ has nonzero $Betti_2$. Build a Vietoris–Rips stream to confirm your answer.

Solution. See the Matlab script `exercise_5_1_2.m` for a solution. Our planar dataset is 6 evenly spaced points on the unit circle. We build a Vietoris–Rips stream which, at the correct filtration value, is an octahedron.

Exercise (5.1.3). Find a planar dataset $Z \subset \mathbb{R}^2$ and a filtration value t such that $\mathrm{VR}(Z,t)$ has nonzero $Betti_6$. When building a Vietoris–Rips stream to confirm your answer, don't forget to choose $d_{max} = 7$.

Solution. See the Matlab script `exercise_5_1_3.m` for a solution. Our planar dataset is 14 evenly spaced points on the unit circle. We build a Vietoris–Rips stream which, at the correct filtration value, is homeomorphic to the 6-sphere. It has 14 vertices because it is obtained by suspending the 0-sphere six times, for a total of $2 + (6 \times 2) = 14$ vertices.

Exercise (5.2.1). Let Z be the point cloud in Figure 1 from Section 4.1, corresponding to the house point cloud. Suppose we are using sequential maxmin to select a set L of 3 landmarks, and the first (randomly selected) landmark is $(1,0)$. Find by hand the other two landmarks in L.

Solution. L is the set $\{(1,0),(0,3),(-1,0)\}$.

Exercise (5.2.2). Let Z be a point cloud and L a landmark subset. Show that if L is chosen via sequential maxmin, then for any $l_i, l_j \in L$, we have $d(l_i, l_j) \geq \mathtt{R}$.

Solution. Without loss of generality, assume that $i < j$ and that landmarks l_i and l_j were the i-th and j-th landmarks selected by the inductive sequential maxmin process. Let L_{j-1} be the first $j-1$ landmarks chosen.

We proceed using a proof by contradiction. Suppose that $d(l_i, l_j) < \mathtt{R}$. By definition of $\mathtt{R}$, there exists a $z \in Z$ such that $d(z, L) = \mathtt{R}$. Note that

$$d(l_j, L_{j-1}) \leq d(l_j, l_i) = d(l_i, l_j) < \mathtt{R} = d(z, L) \leq d(z, L_{j-1}).$$

This contradicts the fact that landmark l_j was chosen at the j-th step of sequential maxmin. Hence, it must be the case that $d(l_i, l_j) \geq \mathtt{R}$.

Exercise (5.3.1). Let Z be the point cloud in Figure 1 from Section 4.1, corresponding to the house point cloud. Let $L = \{(1,0), (0,3), (-1,0)\}$ be the landmark subset. Find by hand the filtration time for the edge between vertices $(1,0)$ and $(0,3)$. Which point or points witness this edge? What is the filtration time for the lone 2-simplex $[(1,0),(0,3),(-1,0)]$?

Solution. The edge between $(1,0)$ and $(0,3)$ has filtration time zero. Points $(1,2)$ or $(0,3)$ witness this edge. The lone 2-simplex has filtration time zero.

Exercise (5.4.1). Let Z be the point cloud in Figure 1 from Section 4.1, corresponding to the house point cloud. Let $L = \{(1,0), (0,3), (-1,0)\}$ be the landmark subset. Let $\nu = 1$. Find by hand the filtration time for the edge between vertices $(1,0)$ and $(0,3)$. Which point or points witness this edge? What is the filtration time for the lone 2-simplex $[(1,0),(0,3),(-1,0)]$?

Solution. The edge between $(1,0)$ and $(0,3)$ has filtration time $2-\sqrt{2}$. Point $(1,2)$ witnesses this edge. The lone 2-simplex has filtration time $\sqrt{2}$, which is when the edge between $(1,0)$ and $(-1,0)$ appears.

Exercise (5.4.2). Repeat the above exercise with $\nu = 0$ and with $\nu = 2$.

Solution. First we do the case when $\nu = 0$. The edge between $(1,0)$ and $(0,3)$ has filtration time 2. Point $(1,2)$ witnesses this edge. The lone 2-simplex has filtration time 2.

Next we do the case when $\nu = 2$. The edge between $(1,0)$ and $(0,3)$ has filtration time zero. Points $(1,2)$ or $(0,3)$ witness this edge. The lone 2-simplex has filtration time zero.

Exercise (5.4.3). Check that the 1-skeleton of a witness complex $\mathrm{W}(Z, L, t)$ is the same as the 1-skeleton of a lazy witness complex $\mathrm{LW}_2(Z, L, t)$. As a consequence, $\mathrm{LW}_2(Z, L, t)$ is the flag complex of $\mathrm{W}(Z, L, t)$.

Solution. This follows from the definition of the witness stream and the definition of the lazy witness stream for $\nu = 2$.

Exercise (5.4.4). Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1] \times [0,1]$ and then computes the $1{,}000 \times 1{,}000$ distance matrix for these points under the induced metric on the flat torus. Create an explicit metric space from this distance matrix.

Build a lazy witness stream on this explicit metric space with 50 landmarks chosen via sequential maxmin and with $\nu = 1$. Confirm the barcodes match the homology of a torus.

Solution. See the Matlab script `exercise_5_4_4.m` and the Matlab function `flatTorusDistanceMatrix.m` for a solution.

Exercise (5.4.5). Write a Matlab script (or function—see Section 7.1) that first selects 1,000 random points from the square $[0,1] \times [0,1]$ and then computes the $1{,}000 \times 1{,}000$ distance matrix for these points under the induced metric on the flat Klein bottle. Create an explicit metric space from this distance matrix.

Build a lazy witness stream on this explicit metric space with 50 landmarks chosen via sequential maxmin and with $\nu = 1$. Confirm the barcodes match the homology of a Klein bottle, over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ coefficients.

Solution. See the Matlab script `exercise_5_4_5.m` and the Matlab function `flatKleinDistanceMatrix.m` for a solution.

References

[1] H. Adams and G. Carlsson, *On the nonlinear statistics of range image patches*, SIAM J. Img. Sci., 2, (2009), pp. 110-117.

[2] M. A. Armstrong, *Basic Topology*, Springer, New York, Berlin, 1983.

[3] G. Carlsson, T. Ishkhanov, V. de Silva, and A. Zomorodian, *On the local behavior of spaces of natural images*, Int. J. Computer Vision, 76 (2008), pp. 1–12.

[4] V. de Silva and G. Carlsson, *Topological estimation using witness complexes*, in Proceedings of the Symposium on Point-Based Graphics, ETH, Zürich, Switzerland, 2004, pp. 157–166.

[5] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction*, American Mathematical Society, Providence, 2010.

[6] H. Edelsbrunner, D. Letscher, and A. Zomorodian, *Topological persistence and simplification*, Discrete Computat. Geom., 28 (2002), pp. 511–533.

[7] A. Hatcher, *Algebraic Topology*, Cambridge University Press, Cambridge, UK, 2002.

[8] A. B. Lee, K. S. Pedersen, and D. Mumford, *The nonlinear statistics of high-contrast patches in natural images*, Int. J. Computer Vision, 54 (2003), pp. 83–103.

[9] H. Sexton and M. Vejdemo-Johansson, JPlex simplicial complex library. `http://comptop.stanford.edu/programs/jplex/`.

[10] A. Zomorodian and G. Carlsson, *Computing persistent homology*, Discrete Computat. Geom., 33 (2005), pp. 247–274.

E-mail address, H. Adams: `henrya@math.stanford.edu`

E-mail address, A. Tausz: `atausz@stanford.edu`