JAVAPLEX TUTORIAL

HENRY ADAMS AND ANDREW TAUSZ

Contents

Date: May 19, 2017.

1. Introduction

1.1. **Javaplex.** Javaplex is a Java software package for computing the persistent homology of filtered simplicial complexes (or more generally, filtered chain complexes), with special emphasis on applications arising in topological data analysis [Tausz et al. 2014]. The main author is Andrew Tausz. Javaplex is a rewrite 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 homology. The website for Javaplex is `http://appliedtopology.github.io/javaplex/`, the documentation overview is at `https://github.com/appliedtopology/javaplex/wiki/Overview`, and the javadoc tree for the library is at `http://appliedtopology.github.io/javaplex/doc/`.

This tutorial is written for those using Javaplex with Matlab. However, one can run Javaplex without Matlab; see `https://github.com/appliedtopology/javaplex/wiki/Interoperability`.

If you are interested in Javaplex, then you may also be interested in the software package Dionysus by Dmitriy Morozov (`http://www.mrzv.org/software/dionysus`) or the software package Persus by Vidit Nanda (`http://www.sas.upenn.edu/~vnanda/perseus/index.html`).

Please email Henry at `adams@math.colostate.edu` or Andrew at `andrew.tausz@gmail.com` if you have questions about this tutorial.

1.2. **License.** Javaplex is an open source software package under the Open BSD License. The source code can be found at `https://github.com/appliedtopology/javaplex`.

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 to the latest release at `https://github.com/appliedtopology/javaplex/releases/latest/`. Download the zip file containing the Matlab examples, which should be called something like `matlab-examples-4.2.3.zip`. Extract the zip file. The resulting folder should be called `matlab_examples`; make sure you know the location of this folder and that it is not inside the zip file.

In Matlab, change Matlab's "Current Folder" to the directory `matlab_examples` that you just extracted from the zip file. In the Matlab command window, run the `load_javaplex.m` file.

```
>> load_javaplex
```

Also in the Matlab command window, type the following command.

```
>> import edu.stanford.math.plex4.*;
```

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 folder `tutorial_examples` contains Matlab scripts, such as `explicit_simplex_example.m` or `house_example.m`, which list all of the commands in this tutorial. This means that you don't need to type in each command individually. The folder `tutorial_examples` also contains the Matlab data files, such as `pointsRange.mat` or `pointsTorusGrid.mat`, which are used in this tutorial. The folder `tutorial_solutions` contains the solution scripts, such as `exercise_1.m`, for all of the tutorial exercises. See Appendix B for exercise solutions.

2. Math review

Below is a brief math review. For more details, see Armstrong [1983], Edelsbrunner and Harer [2010], Edelsbrunner et al. [2002], Hatcher [2002], and Zomorodian and Carlsson [2005].

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 $\sigma = [z_0, z_1, \ldots, z_k]$, 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 $X(t) \subset X(t')$ whenever $t \leq t'$. The filtration value 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 value t_{start}, remains open for $t_{start} \leq t < t_{end}$, and closes at value t_{end}. We are often interested in Betti intervals that persist for a long filtration range.

Persistent homology depends heavily on functoriality: for $t \leq t'$, the inclusion $i : X(t) \to X(t')$ of simplicial complexes induces a map $i_* : H_k(X(t)) \to H_k(X(t'))$ 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.

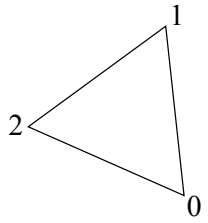
3.1. **Computing homology.** You should change your current Matlab directory to `tutorial_examples`, perhaps using the following command.

```
>> 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 values 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 values, by default all added simplices will have filtration value zero.

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

After we are done building the complex, calling the method `finalizeStream` is necessary before working with this complex!

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 $Betti_1 = 1$.

The persistence algorithm computing intervals can also find a representative cycle for each interval. However, there is no guarantee that the produced representative will be geometrically nice.

```
>> 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` method is used to ensure that the stream has been fully constructed and is ready for consumption by a persistence algorithm. It should be called every time after you build an explicit simplex stream.

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 1. Build a simplicial complex homeomorphic to the torus. Compute its Betti numbers. *Hint: You will need at least 7 vertices* [Hatcher 2002, page 107]. *We recommend using a* 3×3 *grid of 9 vertices.*

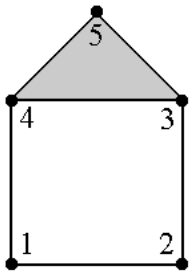
Exercise 2. Build a simplicial complex homeomorphic to the Klein bottle. Check that it has the same Betti numbers as the torus over $\mathbb{Z}_2$ coefficients but different Betti numbers over $\mathbb{Z}_3$ coefficients.

Exercise 3. 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 values. If your filtration values are not all integers, then please see the remark at the end of this section.

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 value 0, with the top vertex appearing at value 1, with the roof edges appearing at values 2 and 3, and with the roof 2-simplex appearing at value 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 value 3 and ending at 7, with a representative cycle formed by the three edges of the roof. This 1-dimensional hole forms when edge $[4, 5]$ appears at filtration value 3 and closes when 2-simplex $[3, 4, 5]$ appears at filtration value 7.

We can also store the intervals as Matlab matrices.

```
>> intervals_dim0 = edu.stanford.math.plex4.homology.barcodes.BarcodeUtility...
.getEndpoints(intervals, 0, 0)
intervals_dim0 =

    0    Inf
    1    2

>> intervals_dim1 = edu.stanford.math.plex4.homology.barcodes.BarcodeUtility...
.getEndpoints(intervals, 1, 0)
intervals_dim1 =

    0    Inf
    3    7
```

The second input of this command is the dimension of the intervals, and the third input is a Boolean flag: 0 to include infinite intervals, and 1 to exclude infinite intervals.

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()
infinite_barcodes =

Dimension: 1
[0.0, infinity): [1,4] + [2,3] + [1,2] + [3,4]
Dimension: 0
[0.0, infinity): [1]
```

We can print the Betti numbers at the largest filtration value (7 in this case) as an array

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

    1
    1
```

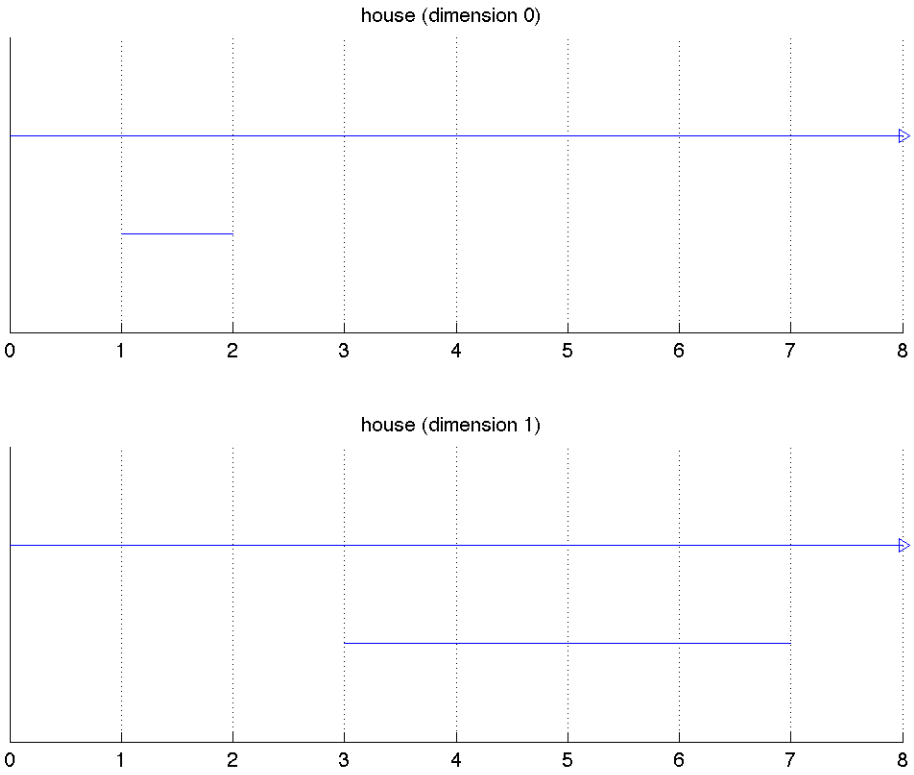
or as a list with entries of the form $k : Betti_k$.

```
>> 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 value 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 values are on the horizontal axis. The $Betti_k$ number of the stream at filtration value 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 values we built into the house stream. At value 0, a connected component

and a 1-dimensional hole form. At value 1, a second connected component appears, which joins to the first at value 2. A second 1-dimensional hole forms at value 3, and closes at value 7.

Remark. 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
```

Remark. If you want to use filtration values that are not integers, then you first need to specify an upper bound on the filtration values in your complex. This is demonstrated below, where the non-integer filtration value is 17.23 and the upper bound is 100.

```
>> stream = api.Plex4.createExplicitSimplexStream(100);
>> stream.addVertex(1, 17.23);
>> stream.finalizeStream();
```

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.

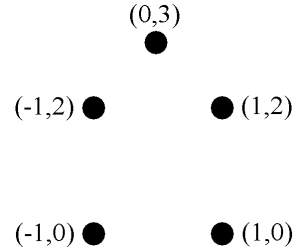


Figure 1. The house point cloud

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 alternatively, 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
>> view(60,40)
```

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.getRandomSphereProductPoints(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
```

Remark. Be careful: the constructor `metric.impl.ExplicitMetricSpace()` will accept matrices that fail to be symmetric, square, or nonnegative, creating "metrics" that do not satisfy the mathematical definition, and which may lead to errors down the road. The triangle inequality is similarly easy to ignore, but this is often useful: sometimes real world "distances" or "similarities" satisfy all of the axioms of a metric except for the triangle inequality, and one can still define a Vietoris–Rips complex on top of this data.

Exercise 4. One way to produce a torus is to take a square $[0, 1] \times [0, 1]$ and then identify opposite sides. This is called the flat torus. More explicitly, the flat torus is 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 $x \in [0, 1]$. 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 that 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 16.

Exercise 5. 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 the flat Klein bottle. More explicitly, the flat Klein bottle is 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 $x \in [0, 1]$. 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 that 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 17.

Exercise 6. One way to produce a projective plane is to take the unit sphere $S^2 \subset \mathbb{R}^3$ and then identify antipodal points. More explicitly, the projective plane is the quotient space

$$S^2/(x \sim -x).$$

The Euclidean metric on S^2 induces a metric on the projective plane.

Write a Matlab script or function that selects 1,000 random points from the unit sphere $S^2 \subset \mathbb{R}^3$ and then computes the 1,000 $\times$ 1,000 distance matrix for these points under the induced metric on the projective plane. Create an explicit metric space from this distance matrix.

This exercise is continued by Exercise 18.

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 de Silva and Carlsson [2004] for additional information.

The Vietoris–Rips, witness, and lazy witness streams all take three of the same inputs: the maximum dimension d_{max} of any included simplex, the maximum filtration value 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 $d_{max} - 1$ since homology in dimension $d_{max} - 1$ depends on the boundary matrix from d_{max}-simplices to $(d_{max} - 1)$-simplices. Also, instead of computing filtered simplcial 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. See also the remark in Section 7.1.

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 included in $\mathrm{VR}(Z, t)$ if $d(a, b) \leq t$.
- a higher dimensional simplex is included in $\mathrm{VR}(Z, t)$ if all of its edges are.

Note that $\mathrm{VR}(Z, t) \subset \mathrm{VR}(Z, t')$ whenever $t \leq t'$, 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 an example of a *clique complex* or 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, where the metric space is $Z = \{(-1, 0), (1, 0), (1, 2), (-1, 2), (0, 3)\}$. 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 ellipses in the command above should be omitted; they are included only to indicate that this command continues onto the next line.

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.

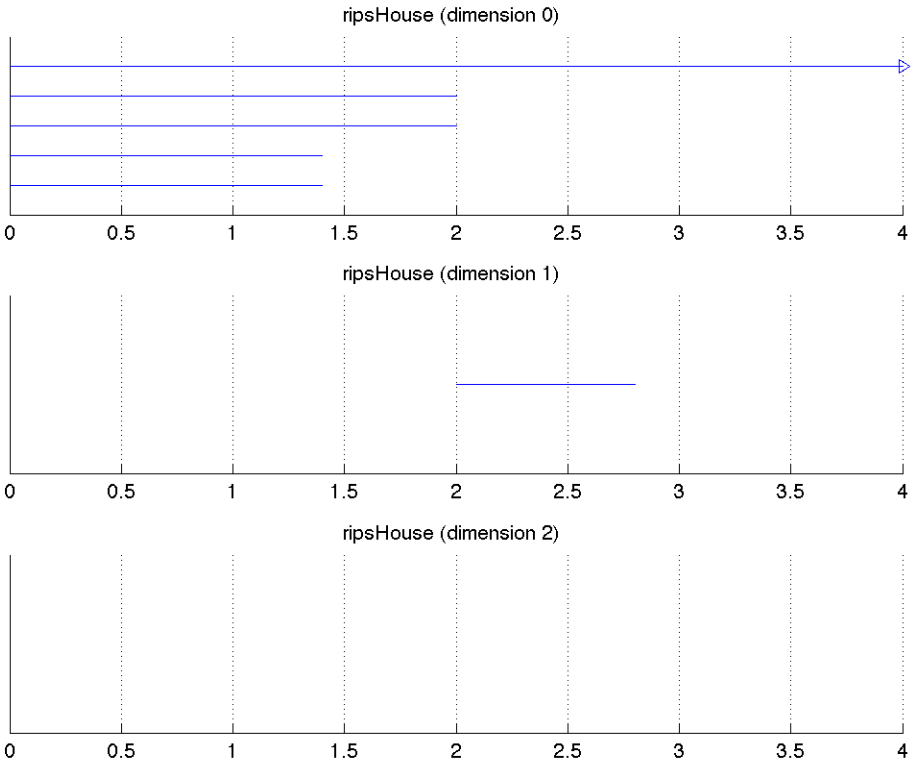
Since $d_{max} = 3$ we can compute up to second dimensional persistent homology.

```
>> 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 value 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 value $t = \sqrt{2}$; 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.

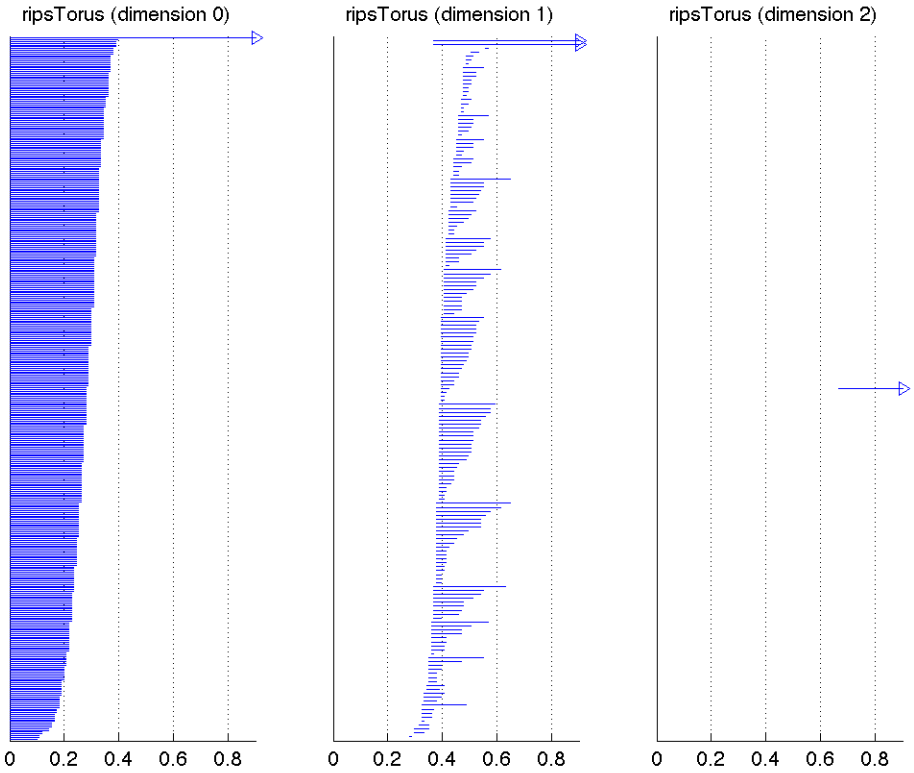
```
>> max_dimension = 3;
>> max_filtration_value = 0.9;
>> num_divisions = 100;
```

Make sure you are in the directory `tutorial_examples` (you may need to enter the command `cd tutorial_examples`), and then 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 more general metric spaces. For example, if `m_space` were an explicit metric space (see Section 4.2), then we could call the following command.

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

Exercise 7. 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 8. 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 9. 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. However, outlier points are less of an issue if one first takes dense core subsets as in Appendix A. Sequential maxmin landmarks are used by Adams and Carlsson [2009] and Carlsson et al. [2008].

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.

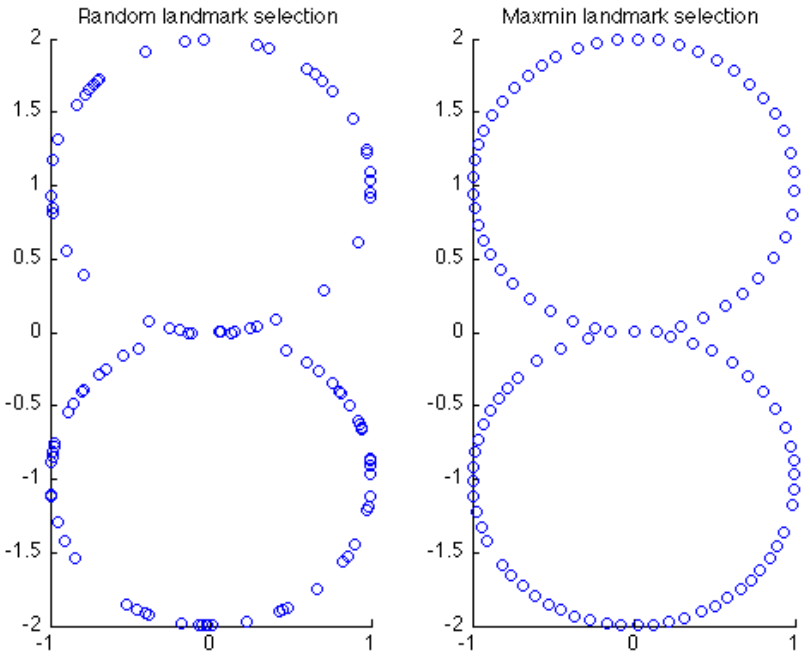
```
>> num_landmark_points = 100;
>> random_selector = api.Plex4.createRandomSelector(point_cloud, num_landmark_points);
>> maxmin_selector = api.Plex4.createMaxMinSelector(point_cloud, num_landmark_points);
```

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 more general 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 $\mathtt{R} = \max_{z \in Z}\{d(z, L)\}$. Number `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 10. 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 11. 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 $\mathrm{W}(Z, L, t) \subset \mathrm{W}(Z, L, t')$ whenever $t \leq t'$, so the witness stream is a filtered simplicial complex. Note that a landmark point can serve as a witness point.

Exercise 12. 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 value for the edge between vertices $(1, 0)$ and $(0, 3)$. Which point or points witness this edge? What is the filtration value 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 sequential maxmin 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.createMaxMinSelector(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 = 0.7033                          % Generally close to 0.7
>> max_filtration_value = R / 8;
```

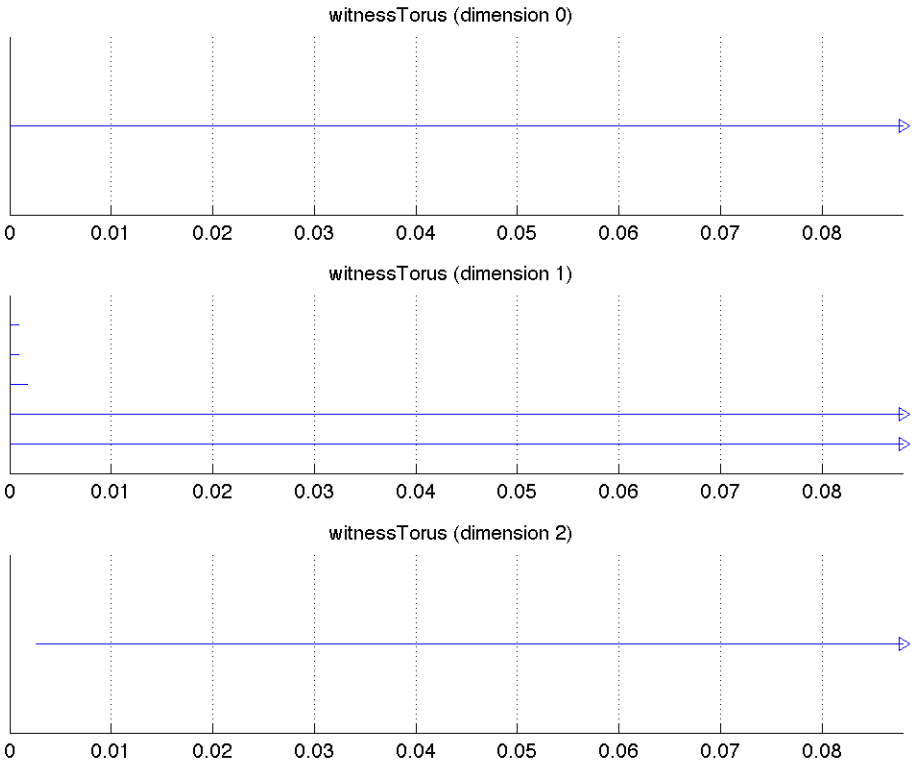
We create the witness stream.

```
>> stream = api.Plex4.createWitnessStream(landmark_selector, max_dimension, ...
max_filtration_value, num_divisions);
>> num_simplices = stream.getSize()
num_simplices = 1164                          % Generally close to 1200
```

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 de Silva and Carlsson [2004] 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
$$\max\big\{d(a, z), d(b, z)\big\} \leq t + m(z).$$
- a higher dimensional simplex is in $\mathrm{LW}_\nu(Z, L, t)$ if all of its edges are.

Note that $\mathrm{LW}_\nu(Z, L, t) \subset \mathrm{LW}_\nu(Z, L, t')$ whenever $t \leq t'$, so the lazy witness stream is a filtered simplicial complex. 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 13. 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 value for the edge between vertices $(1, 0)$ and $(0, 3)$. Which point or points witness this edge? What is the filtration value for the lone 2-simplex $[(1, 0), (0, 3), (-1, 0)]$?

Exercise 14. Repeat the above exercise with $\nu = 0$ and with $\nu = 2$.

Exercise 15. 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`.

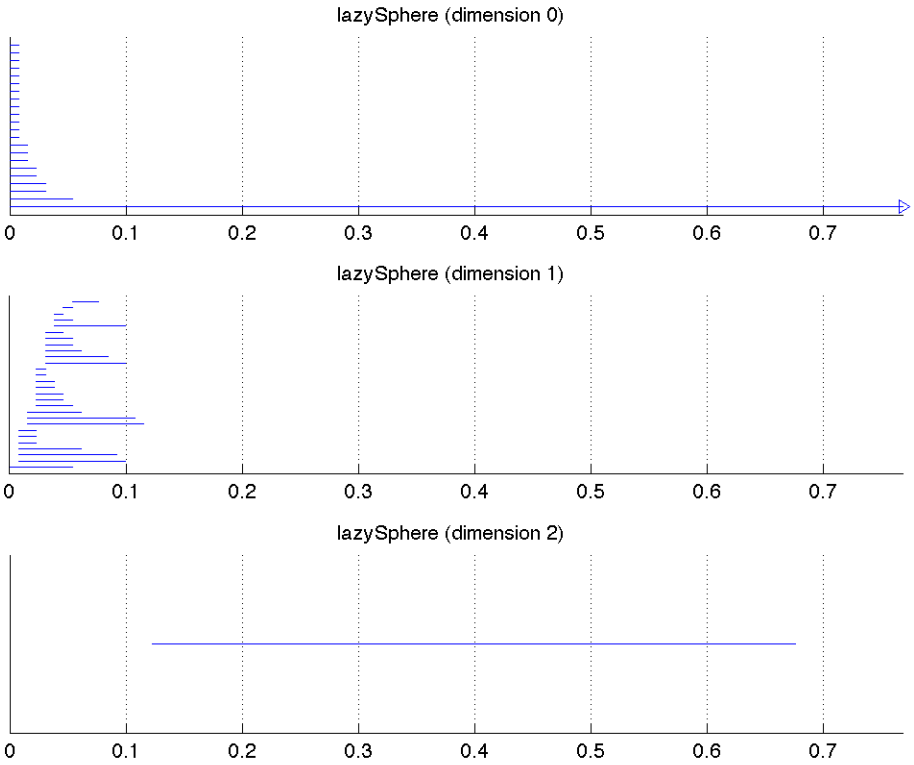
We use parameter $\nu = 1$.

```
>> 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.createMaxMinSelector(point_cloud, num_landmark_points);
```

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

```
>> R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
R = 0.3841                          % Generally close to 0.38
>> max_filtration_value = 2 * 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 = 56518                           % Generally close to 50000
>> 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 16. In Exercise 4 you created an explicit metric space for 1,000 random points on a flat torus. 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 17. In Exercise 4 you created an explicit metric space for 1,000 random points on a flat Klein bottle. 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.

Exercise 18. In Exercise 6 you created an explicit metric space for 1,000 random points on a projective plane. 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 projective plane, over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ coefficients.

Exercise 19. Sample points from an embedding of a double torus, that is, a surface of genus two, in $\mathbb{R}^3$. Build a lazy witness stream on this Euclidean metric space. Confirm the barcodes match the homology of a double torus. Choosing suitable parameters will not be easy.

6. Examples with real data

We now do two examples with real datasets: a data set of range image patches, and a data set of optical image patches

6.1. **Range image patches.** The corresponding Matlab script is `range_image_example.m`, and it relies on the files `pointsRange.mat` and `dct.m`.

In *On the nonlinear statistics of range image patches* [Adams and Carlsson 2009], we study a space of range image patches drawn from the Brown database [Lee et al. 2003]. 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 Adams and Carlsson [2009] 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 Adams and Carlsson [2009]: $X^5(300, 30)$ has the topology of a circle.

Make sure you are in the directory `tutorial_examples` (you may need to enter the command `cd tutorial_examples`), and then 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.

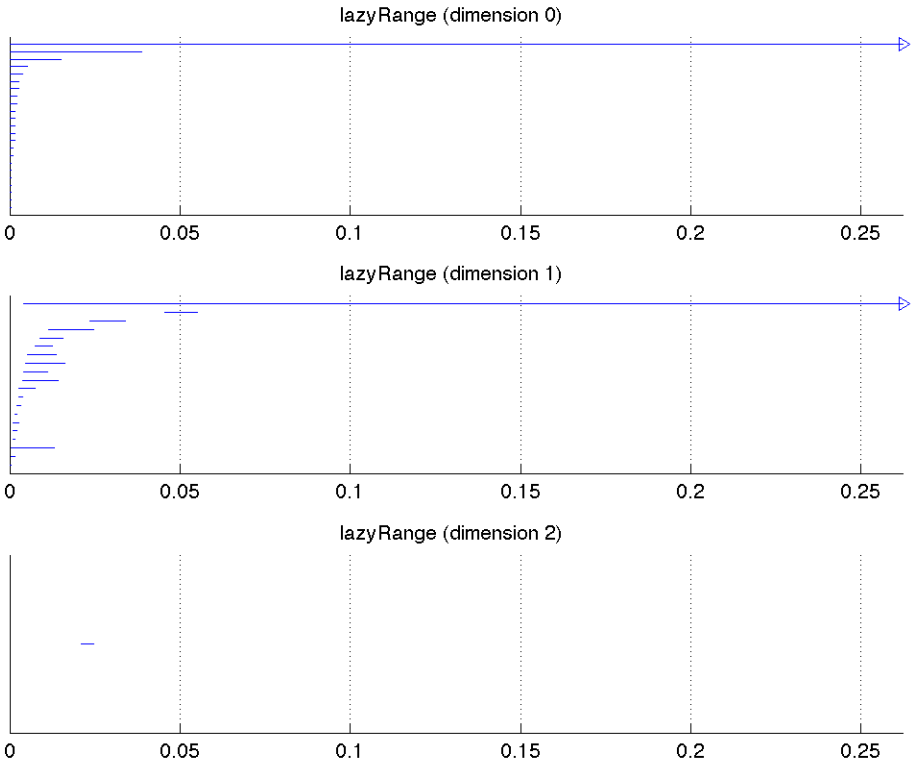


FIGURE 2. Betti intervals for the lazy witness complex built from $X^5(300, 30)$

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.

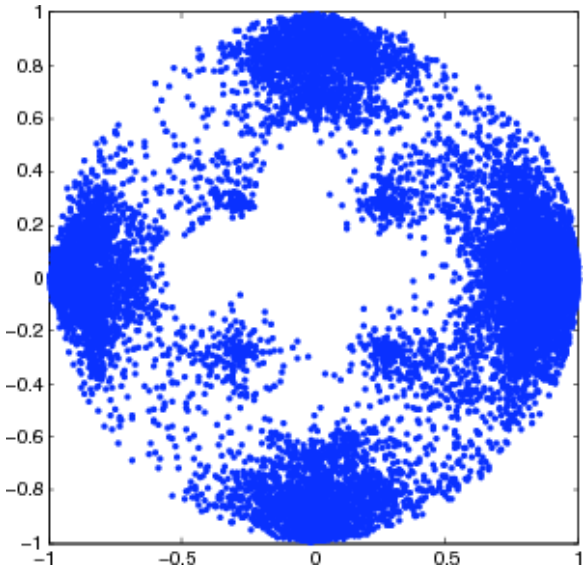
Our 5×5 normalized patches are currently in the pixel basis: every coordinate corresponds to the range value at one of the 25 pixels. The Discrete Cosine Transform (DCT) basis is a useful basis for our patches [Adams and Carlsson 2009; Lee et al. 2003]. We change to this basis in order to plot a projection of the loop evidenced by Figure 6. 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);
```

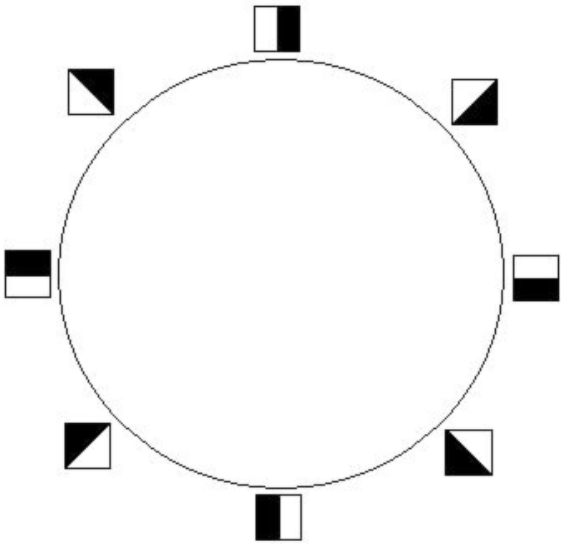
Two of the DCT basis vectors are horizontal and linear gradients.

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

```
>> figure
>> 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).

6.2. **Optical image patches.** The corresponding Matlab script is `optical_image_example.m`, and it relies on the files `pointsOpticalDct_k300.mat` and `pointsOpticalDct_k15.mat`.

The optical image database collected by van Hateren and van der Schaaf [1998] contains black and white digital photographs from a variety of indoor and outdoor scenes. Lee et al. [2003] study 3×3 patches from these images, and Carlsson et al. [2008] continue the analysis of image patches using persistent homology. Carlsson et al. [2008] begin with a large collection of high-contrast, normalized 3×3 pixel patches, each thought of as a point in $\mathbb{R}^9$. They change to the Discrete Cosine Transform (DCT) basis, which maps the patches to the unit sphere $S^7 \subset \mathbb{R}^8$. They select from this space the 30% densest vectors, based first on the density estimator ρ_{300} (see Appendix A), and next based on the density estimator ρ_{15}. In Carlsson et al. [2008] these dense core subset are denoted $X(300, 30)$ and $X(15, 30)$, and they contain 15,000 points. In the next example we verify two results from Carlsson et al. [2008]: $X(300, 30)$ has the topology of a circle, and $X(15, 30)$ has the topology of a three circle model.

Make sure you are in the directory `tutorial_examples` (you may need to enter the command `cd tutorial_examples`), and then load the file `pointsOpticalDct_k300.mat`. The matrix `pointsOpticalDct_k300` appears in your Matlab workspace.

```
>> load pointsOpticalDct_k300.mat
>> size(pointsOpticalDct_k300)
ans = 15000    8                          % 15000 points in dimension 8
```

Matrix `pointsOpticalDct_k300` is in fact $X(300, 30)$: each of its rows is a vector in $\mathbb{R}^8$.

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(pointsOpticalDct_k300, ...
num_landmark_points);
>> R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
R = 0.5919
>> max_filtration_value = R / 4;
>> 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 = 2351

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

>> options.filename = 'lazyOpticalDct-k300';
>> options.max_filtration_value = max_filtration_value;
>> options.max_dimension = max_dimension - 1;
>> plot_barcodes(intervals, options);
```

The file `lazyOpticalDct-k300.png` is saved to your current directory.

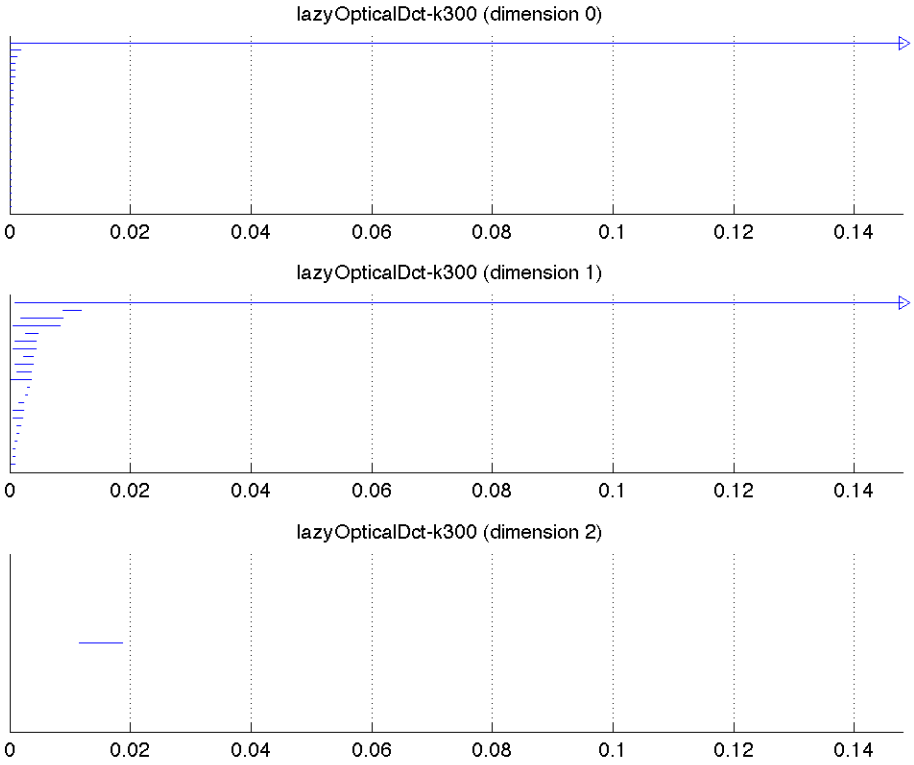
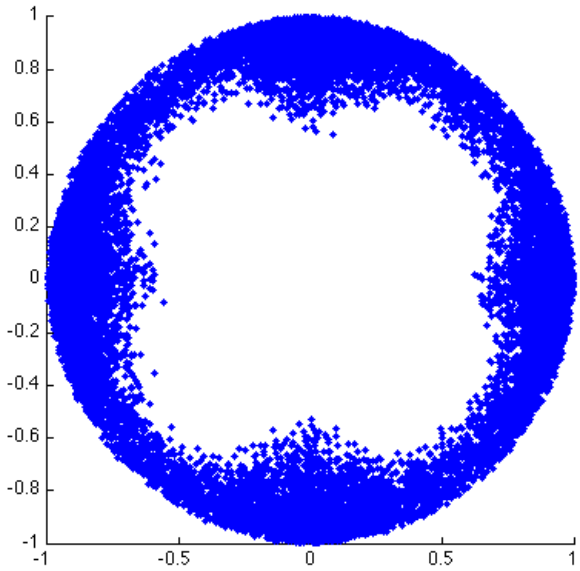


FIGURE 3. Betti intervals for the lazy witness complex built from $X(300, 30)$

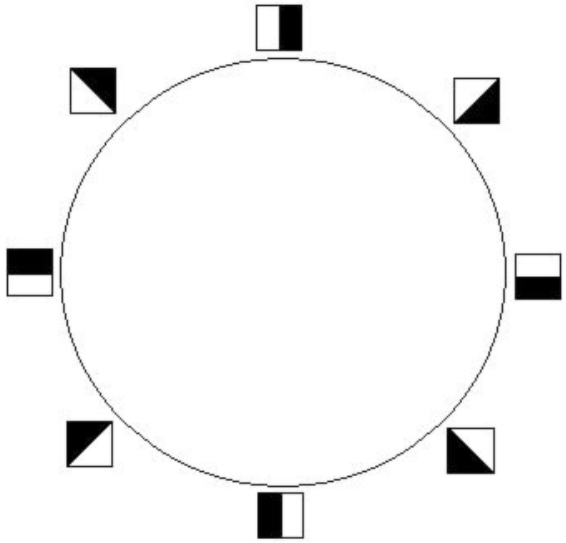
The plots above show that for a long range, the circle Betti numbers $Betti_0 = Betti_1 = 1$ are obtained. Your plot should contain a similar range. This is good evidence that the core subset $X(300, 30)$ is well-approximated by a circle.

We plot the projection of `pointsOpticalDct_k300` onto the linear gradient DCT basis vectors.

```
>> figure
>> scatter(pointsOpticalDct_k300(:,1), pointsOpticalDct_k300(:,2), '.')
>> axis square
```



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



(b) Optical primary circle

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

We next analyze the core subset $X(15, 30)$. Load the file `pointsOpticalDct_k15.mat`. The matrix `pointsOpticalDct_k15` appears in your Matlab workspace.

```
>> load pointsOpticalDct_k15.mat
>> size(pointsOpticalDct_k15)
ans = 15000    8                          % 15000 points in dimension 8
```

Matrix `pointsOpticalDct_k15` is in fact $X(15, 30)$: each of its rows is a vector in $\mathbb{R}^8$.

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(pointsOpticalDct_k15, ...
num_landmark_points);
>> R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
R = 0.6554
```

```
>> max_filtration_value = R / 4;
>> 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 = 1570

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

>> options.filename = 'lazyOpticalDct-k15';
>> options.max_filtration_value = max_filtration_value;
>> options.max_dimension = max_dimension - 1;
>> plot_barcodes(intervals, options);
```

The file `lazyOpticalDct-k15.png` is saved to your current directory.

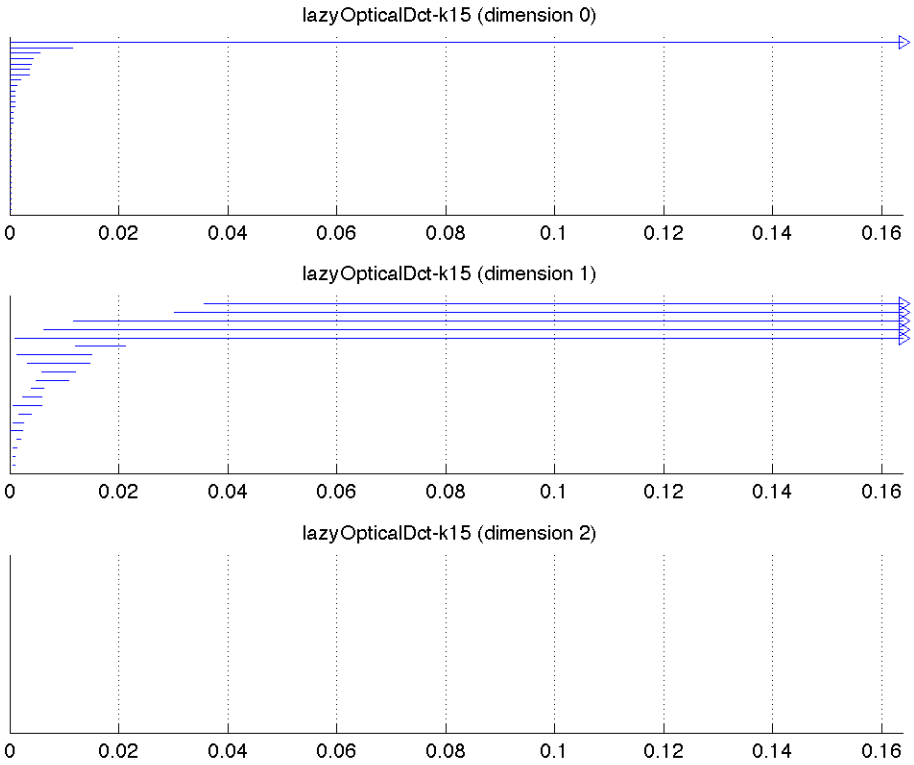
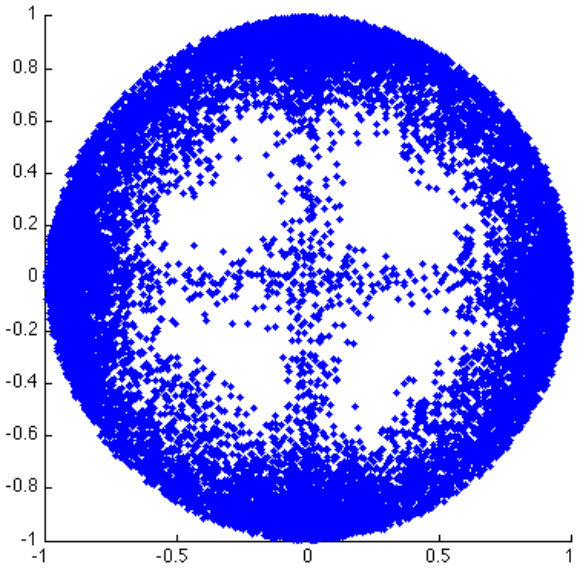


FIGURE 4. Betti intervals for the lazy witness complex built from $X(15, 30)$

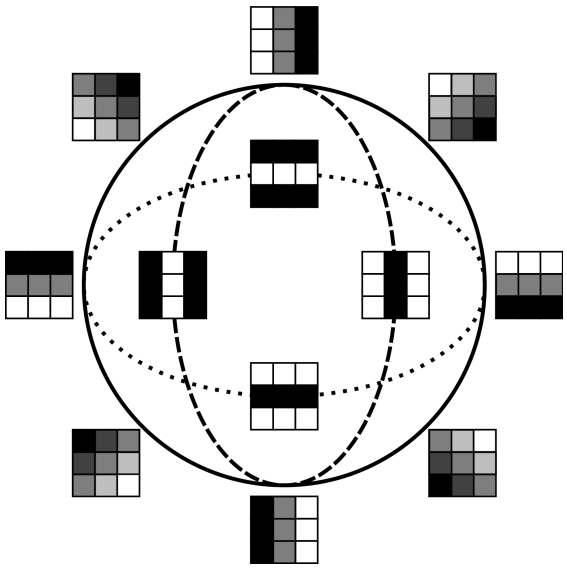
The plots above show that for a long range, the Betti numbers $Betti_0 = 1$ and $Betti_1 = 5$ are obtained. Your plot should contain a similar range. This is good evidence that the core subset $X(15, 30)$ is well-approximated by a connected space with five loops.

We plot the projection of `pointsOpticalDct_k300` onto the linear gradient DCT basis vectors.

```
>> figure
>> scatter(pointsOpticalDct_k15(:,1), pointsOpticalDct_k15(:,2), '.')
>> axis square
```



(a) Projection of $X(15, 30)$



(b) Three circle model

Figure (a) shows a projection of $X(15, 30)$. The space $X(15, 30)$ is parameterized by the three circle model shown in Figure (b). The solid outer circle in the three circle model is the primary circle and contains linear gradients. The dotted and dashed inner circles are the horizontal and vertical secondary circles which contain quadratic gradients. Each secondary circle intersects the primary circle twice, but the secondary circles do not intersect each other; this results in a connected space with $Betti_1 = 5$. The primary circle reflects nature's preference for linear gradients in all directions, and the secondary circles reflect nature's preference for the horizontal and vertical directions.

6.3. **Cyclo-octane molecule conformations.** The corresponding Matlab script is `cyclo_octane_example.m`, and it relies on the file `pointsCycloOctane.mat`.

The cyclo-octane molecule C_8H_{16} consists of a ring of 8 carbons atoms, each bonded to a pair of hydrogen atoms (Figure 5). A conformation of this molecule is a chemically and physically possible realization in 3D space, modulo translations and rotations. The locations of the carbon atoms in a conformation determine the locations of the hydrogen atoms via energy minimization, and hence each molecule conformation can be mapped to a point in $\mathbb{R}^{24} = \mathbb{R}^{8\cdot 3}$, as there are eight carbon atoms in the molecule and each carbon location is represented by three coordinates x, y, z. This map realizes the conformation space of cyclo-octane as a subset of $\mathbb{R}^{24}$. It turns out that the conformation space is a two-dimensional stratefied space, i.e. a two-dimensional manifold with singularities. Furthermore, Brown et al. [2008], Martin et al. [2010], and Martin and Watson [2011] show that the conformation space of cyclo-octane is the union of a sphere with a Klein bottle, glued together along two circles of singularities (see Figures 7 and 8 in Martin and Watson [2011]). Indeed, the algorithm they develop allows them to triangulate this conformation space from a finite sample.

Zomorodian [2012] uses the cyclo-octane dataset as an example to show that we can efficiently recover the homology groups of the conformation space using persistent homology. In this section we essentially follow Zomorodian's example. We begin with a sample of 6,040 points on the conformation space (this data is publicly available at Shawn Martin's webpage `http://www.sandia.gov/~smartin/software.html`) and compute the resulting persistent homology. We obtain the Betti numbers $Betti_0 = Betti_1 = 1$ and

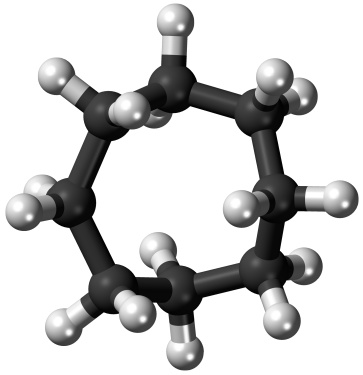


FIGURE 5. The cyclo-octane molecule consists of a ring of 8 carbon atoms (black), each bonded to a pair of hydrogen atoms (white).

$Betti_2 = 2$, which match the homology groups of the union of a sphere with a Klein bottle, glued together along two circles of singularities.

Make sure you are in the directory `tutorial_examples` (you may need to enter the command `cd tutorial_examples`), and then load the file `pointsCycloOctane.mat`. The matrix `pointsCycloOctane` appears in your Matlab workspace.

```
>> load pointsCycloOctane.mat
>> size(pointsCycloOctane)
ans = 6040    24                          % 6040 points in dimension 24
```

Matrix `pointsCycloOctane` is a sample of 6,040 points from the cyclo-octane conformation space: each of its rows is a vector in $\mathbb{R}^{24}$.

We pick 100 sequential maxmin landmark points, and we build the lazy witness stream with parameter $\nu = 1$.

```
>> max_dimension = 3;
>> num_landmark_points = 100;
>> max_filtration_value = 0.5;
>> nu = 1;
>> num_divisions = 500;

>> landmark_selector = api.Plex4.createMaxMinSelector(pointsCycloOctane, ...
num_landmark_points);
>> R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
R = 0.8046
>> 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 = 8025

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

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

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

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

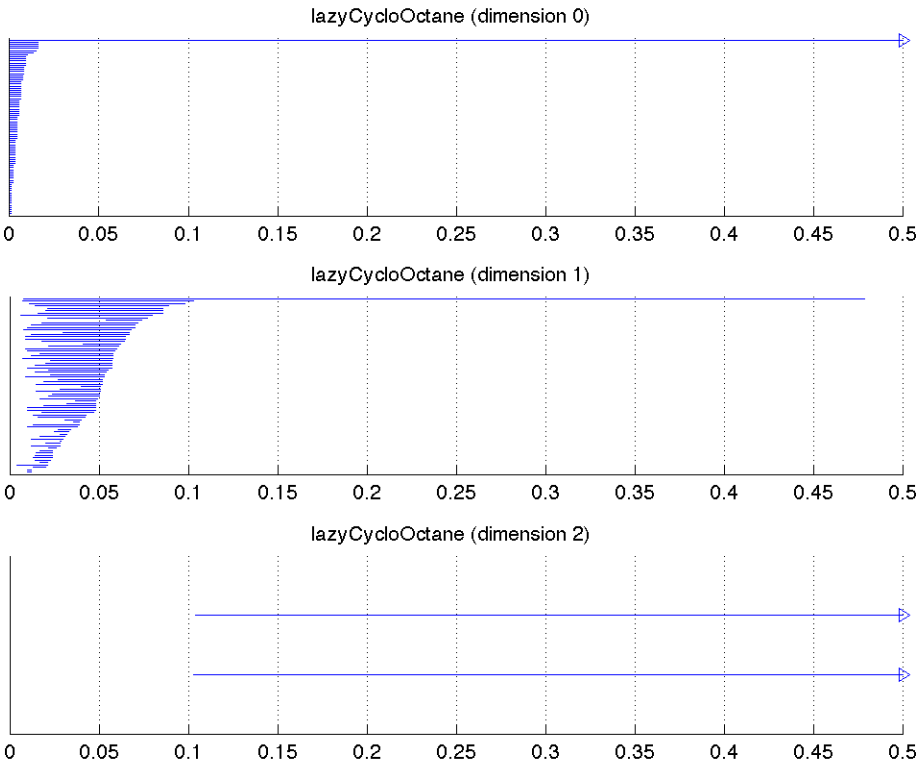


FIGURE 6. Betti intervals for the lazy witness complex built from $X(300, 30)$

The plots above show that for a long range, the Betti numbers $Betti_0 = Betti_1 = 1$ and $Betti_2 = 2$ are obtained, which matches the homology of the union of a sphere with a Klein bottle, glued together along two circles of singularities.

7. REMARKS

7.1. **Java heap size.** Depending on the size of your Javaplex computations, you may need to increase the maximum Java heap size. This should not be necessary for the examples in this tutorial.

The following command returns your maximum heap size in bytes.

```
>> java.lang.Runtime.getRuntime.maxMemory
ans = 130875392
```

My Matlab installation sets the limit to approximately 128 megabytes by default. To increase your limit to, say, 1.5 gigabytes, create a file named `java.opts` in your Matlab directory which contains the text

`-Xmx1500m` and then restart Matlab. For more information, please see this link: `http://www.mathworks.com/support/solutions/en/data/1-18I2C/`.

7.2. **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.
```

7.3. **Displaying the simplices in a stream.** It is possible to display the simplices in a stream, along with their filtration values. You can also obtain the vertices of each simplex as a Matlab matrix. For an example of how to do this, please see the file `dump_example.m` in the folder `basic_examples`.

7.4. **Computing the bottleneck distance.** It is possible to compute the bottleneck distance between two barcodes. For an example of how to do this, please see the file `bottleneck_distance_example.m` in the folder `tutorial_examples`.

8. Acknowledgements

We would like to thank the authors of Brown et al. [2008], Martin et al. [2010], and Martin and Watson [2011] for allowing us to use the cyclo-octane dataset in this tutorial.

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 significantly 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 Carlsson et al. [2008] and de Silva and Carlsson [2004] 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.

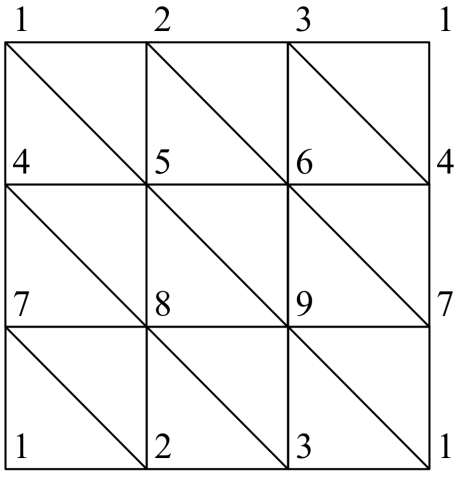
Note: As its name suggests, the m-file `kDensitySlow.m` is not the most efficient way to calculate ρ_k for large datasets. There is a faster file `kDensity.m` for this purpose, which uses the kd-tree data structure. It is not included in the tutorial because it requires one to download a kd-tree package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

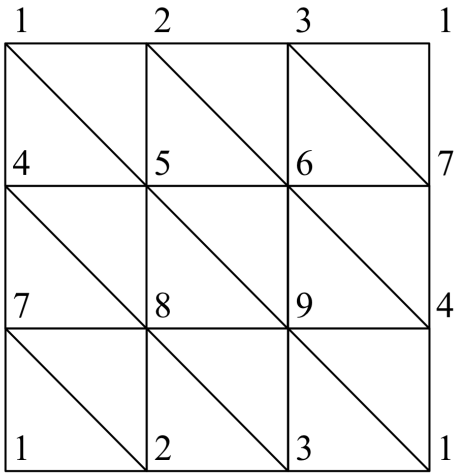
Exercise 1. Build a simplicial complex homeomorphic to the torus. Compute its Betti numbers. *Hint: You will need at least 7 vertices* [Hatcher 2002, page 107]. *We recommend using a* 3×3 *grid of 9 vertices.*

Solution. See the Matlab script `exercise_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.



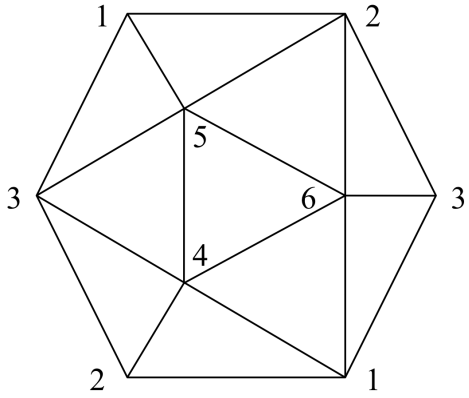
Exercise 2. Build a simplicial complex homeomorphic to the Klein bottle. Check that it has the same Betti numbers as the torus over $\mathbb{Z}_2$ coefficients but different Betti numbers over $\mathbb{Z}_3$ coefficients.

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



Exercise 4. Write a Matlab script or function that 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.m` and the Matlab function `flatTorusDistanceMatrix.m` for a solution.

Exercise 5. Write a Matlab script or function that 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_5.m` and the Matlab function `flatKleinDistanceMatrix.m` for a solution.

Exercise 6. Write a Matlab script or function that selects 1,000 random points from the unit sphere $S^2 \subset \mathbb{R}^3$ and then computes the 1,000 $\times$ 1,000 distance matrix for these points under the induced metric on the projective plane. Create an explicit metric space from this distance matrix.

Solution. See the Matlab script `exercise_6.m` and the Matlab function `projPlaneDistanceMatrix.m` for a solution.

Exercise 7. 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 8. 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_8.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 9. 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_9.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 10. 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 11. 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 12. 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 value for the edge between vertices $(1, 0)$ and $(0, 3)$. Which point or points witness this edge? What is the filtration value for the lone 2-simplex $[(1, 0), (0, 3), (-1, 0)]$?

Solution. Since m_1 is the distance from a point to its second closest landmark, we have $m_1((\pm 1, 0)) = 2 = m_1((\pm 1, 2))$ and $m_1((0, 3)) = \sqrt{10}$. It follows that the edge between $(1, 0)$ and $(0, 3)$ has filtration value zero, and that points $(1, 2)$ and $(0, 3)$ are both witness for this edge at filtration value zero.

Since m_2 is the distance from a point to its third closest landmark, we have $m_2((\pm 1, 0)) = \sqrt{10} = m_2((0, 3))$ and $m_2((\pm 1, 2)) = 2\sqrt{2}$. It follows that the lone 2-simplex has filtration value zero, and that all five points of Z are witnesses for this 2-simplex at filtration value zero.

Exercise 13. 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 value for the edge between vertices $(1,0)$ and $(0,3)$. Which point or points witness this edge? What is the filtration value for the lone 2-simplex $[(1,0),(0,3),(-1,0)]$?

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

Exercise 14. 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 value 2. Point $(1,2)$ witnesses this edge. The lone 2-simplex has filtration value 2.

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

Exercise 15. 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 16. In Exercise 4 you created an explicit metric space for 1,000 random points on a flat torus. 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_16.m` and the Matlab function `flatTorusDistanceMatrix.m` for a solution.

Exercise 17. In Exercise 5 you created an explicit metric space for 1,000 random points on a flat Klein bottle. 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_17.m` and the Matlab function `flatKleinDistanceMatrix.m` for a solution.

Exercise 18. In Exercise 6 you created an explicit metric space for 1,000 random points on a projective plane. 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 projective plane, over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ coefficients.

Solution. See the Matlab script `exercise_18.m` and the Matlab function `projPlaneDistanceMatrix.m` for a solution.

Exercise 19. Sample points from an embedding of a double torus, that is, a surface of genus two, in $\mathbb{R}^3$. Build a lazy witness stream on this Euclidean metric space. Confirm the barcodes match the homology of a double torus. Choosing suitable parameters will not be easy.

Solution. See the Matlab script `exercise_19.m` and the Matlab function `getDoubleTorusPoints.m`. Thanks to Ulrich Bauer for this solution.

References

H. Adams and G. Carlsson. On the nonlinear statistics of range image patches. *SIAM J. Imag. Sci.*, 2: 110–117, 2009.

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

M. W. Brown, S. Martin, S. N. Pollock, E. A. Coutsias, and J. P. Watson. Algorithmic dimensionality reduction for molecular structure analysis. *Journal of Chemical Physics*, 129:064118, 2008.

G. Carlsson, T. Ishkhanov, V. de Silva, and A. Zomorodian. On the local behavior of spaces of natural images. *Int. J. Comput. Vision*, 76:1–12, 2008.

V. de Silva and G. Carlsson. Topological estimation using witness complexes. In *Eurographics Symposium on Point-Based Graphics*, June 2004.

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

H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete Comput. Geom.*, 28:511–533, 2002.

A. Hatcher. *Algebraic Topology*. Cambridge University Press, Cambridge, 2002.

A. B. Lee, K. S. Pedersen, and D. Mumford. The nonlinear statistics of high-contrast patches in natural images. *Int. J. Comput. Vision*, 54:83–103, 2003.

S. Martin and J. P. Watson. Non-manifold surface reconstruction from high-dimensional point cloud data. *Computational Geometry*, 44:427–441, 2011.

S. Martin, A. Thompson, E. A. Coutsias, and J. P. Watson. Topology of cyclo-octane energy landscape. *Journal of Chemical Physics*, 132:234115, 2010.

A. Tausz, M. Vejdemo-Johansson, and H. Adams. JavaPlex: A research software package for persistent (co)homology. In H. Hong and C. Yap, editors, *Proceedings of ICMS 2014*, Lecture Notes in Computer Science 8592, pages 129–136, 2014. Software available at `http://appliedtopology.github.io/javaplex/`.

J. H. van Hateren and A. van der Schaaf. Independent component filters of natural images compared with simple cells in primary visual cortex. *Proc. R. Soc. Lond. B*, 265:359–366, 1998.

A. Zomorodian. *Advances in Applied and Computational Topology*. American Mathematical Society, 2012.

A. Zomorodian and G. Carlsson. Computing persistent homology. *Discrete Comput. Geom.*, 33:249–274, 2005.

E-mail address, Henry Adams: `adams@math.colostate.edu`

E-mail address, Andrew Tausz: `andrew.tausz@gmail.com`