JavaPlex Tutorial

Henry Adams and Andrew Tausz
henrya@math.stanford.edu and atausz@stanford.edu

May 22, 2011

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

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`.

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.

**** Explain how to download and extract
Extract the contents of the zip file containing the examples. It should be called something like `javaplex-examples-4.00.tar.`
Run the `load_javaplex.m` file.

```
>> load_javaplex
```

Each time upon starting a new Matlab session, you will need to run `load_javaplex.m`.

1.4. **Accompanying files.** Matlab scripts containing the commands in this tutorial are available in the folder `javaplex/src/matlab/for_distribution/tutorial_examples`.

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.

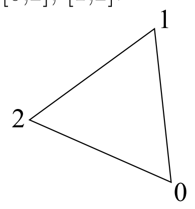
3. Explicit simplex streams

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 §5 we will learn about other automated methods of generating simplicial complexes; namely the Vietoris–Rips, witness, and lazy witness constructions.

3.1. **Explicit simplex streams and homology.** The Matlab script corresponding to this section is `explicit_simplex_exam` 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], and [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 get an empty ExplicitSimplexStream instance. Many command lines in this tutorial will end with a semicolon to supress unwanted output as below.

```
>> 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 simplex with the given array of vertices. By default, these will all have filtration times of zero.

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

We print the total number of simplices in the complex.

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

In order to compute the homology of our complex, we first create an object that will perform the computation. The following line obtains the default algorithm for performing simplicial homology. There are other variants on the persistence algorithm, and one can also change the ground field. This default object will perform the homology computation over $\mathbb{Z}/(2)$. The parameter 3 indicates that the maximum dimension to compute homology to is 3.

```
>> persistence = api.Plex4.getDefaultSimplicialAlgorithm(3);
```

We compute and print the intervals.

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

Dimension:  1
[0, infinity)

Dimension:  0
[0, infinity)
```

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

9-sphere example. Let's build a 9-sphere, which is homeomorphic to the boundary of a 10-simplex. First we create a complex containing only a single 10-simplex. This 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.

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

We print the total number of simplices in the complex.

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

We get the default persistence computation

```
>> persistence = api.Plex4.getDefaultSimplicialAlgorithm(dimension + 1);
```

and compute and print the intervals.

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

Dimension:  9
[0, infinity)

Dimension:  0
[0, infinity)
```

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

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.

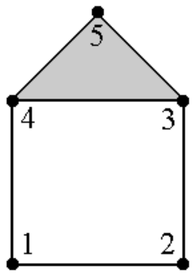
Exercise 3.1.2. Build a simplicial complex homeomorphic to the Klein bottle. Check that it has the same Betti numbers as the torus over $\mathbb{Z}_2$ coefficients but different Betti numbers over $\mathbb{Z}_3$ coefficients.

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. **Explicit simplex streams and persistent homology.**

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

Let's build a stream with nontrivial filtration times. We build a house, with the square appearing at time 0, the top vertex at time 1, the roof edges at times 2 and 3, and the roof 2-simplex 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);
```

We get the default persistence computation,

```
>> persistence = api.Plex4.getDefaultSimplicialAlgorithm(3);
```

compute the intervals,

```
>> filtration_index_intervals = persistence.computeIntervals(stream);
```

and transform the integral intervals to floating point intervals.

```
>> transformer = homology.filtration.IdentityConverter.getInstance();
>> filtration_value_intervals = transformer.transform(filtration_index_intervals)
filtration_value_intervals =

Dimension:  1
[3.000000, 7.000000)
[0.000000, infinity)

Dimension:  0
[1.000000, 2.000000)
[0.000000, 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.

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

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

We can print the Betti numbers (at time 7 or infinity) as an array

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

betti_numbers_array =
    1
    1
```

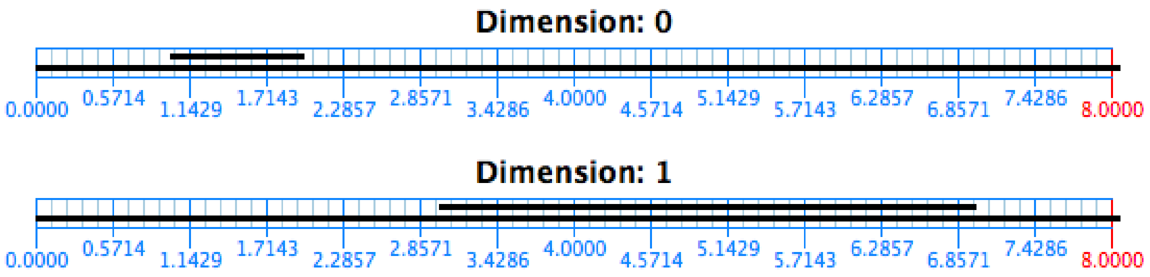
or as a list with entries of the form $k : Betti_k$.

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

The method `createBarcodePlot` lets us display the intervals as a Betti barcode. The three inputs are `filtration_value_intervals`, a string for the filename, and the maximum filtration time (which may be adjusted ±10%) for the plot.

```
>> api.Plex4.createBarcodePlot(filtration_value_intervals, 'house', 8)
```

The files `house_0.PNG` and `house_1.PNG` are 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 stream `house`. 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.

3.3. **Explicit simplex stream details.** We mention two remaining details about class ExplicitSimplexStream.

The methods `addElement` and `removeElementIfPresent` do not necessarily enforce the definition of a stream. They allow us to build inconsistent streams in which some simplex $\sigma \in X(t)$ contains a sub-simplex $\sigma' \notin X(t)$, 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 the points in Euclidean space, or one can create an explicit metric space by specifying all pairwise distances between points. In §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_examples.m`.

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

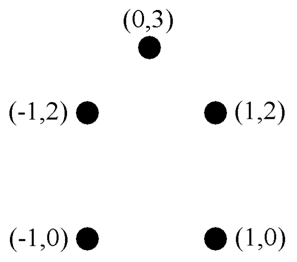


Figure 1. The house point cloud

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 100 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(100);
```

We plot the points.

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

Torus example. We select 2000 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
>> plot3(point_cloud(:,1), point_cloud(:,2), point_cloud(:,3), '.')
>> axis equal
```

Sphere product example. We select 1000 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 gives a circle S^1.

```
>> figure
>> plot(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 command `ExplicitMetricSpace`. For a point cloud in Euclidean space, this method is generally less convenient than the command `EuclideanMetricSpace`. However, command `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.m`.

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 matrix.

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

We return the distance between points 0 and 2.

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

5. Streams from point cloud data

In §3 we built instances of the class SimplexStream from scratch. In this section we construct streams from a point cloud Z. We use the three subclasses RipsStream, WitnessStream, and LazyWitnessStream, which build the Vietoris-Rips, witness, and lazy witness streams. See [4] for additional information.

All three subclasses take four of the same inputs: the granularity δ, the maximum dimension d_{max}, the maximum filtration time t_{max}, and a point cloud Z stored as a PointData instance. The first three 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 $t = 0, \delta, 2\delta, 3\delta, ..., N\delta$, where N is the largest integer such that $N\delta \leq t_{max}$. In this tutorial we use $\delta = 0.001$.

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 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 sometimes halts the computation.

I am currently working with Jan Segert on interactive visualizations of the Rips and Witness filtrations for the Wolfram Demonstrations Project. We have preliminary drafts of the demonstrations. Please email me if you'd like to check them out; in particular, the witness filtrations can be hard to visualize.

5.1. **Subclass RipsStream.** Let $d(\cdot\,,\,\cdot)$ denote the distance between two points. A natural stream to build is the Rips stream. The complex $\text{Rips}(Z, t)$ is defined as follows:

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

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

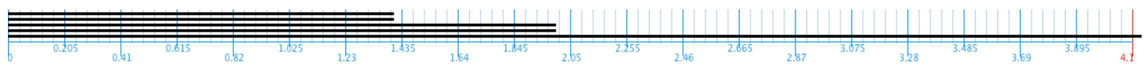
Let's build a Rips stream instance `ripsHouse` from the PointData instance `pdataHouse`. Note this stream is different than the ExplicitStream `house` we built in §**??**.

```
>> ripsHouse = Plex.RipsStream(0.001, 3, 4, pdataHouse);
```

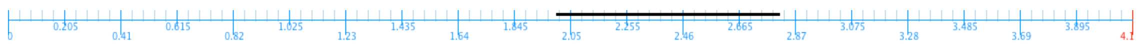
The order of the inputs is `RipsStream(`δ`, `d_{max}`, `t_{max}`, `Z`)`. Since $d_{max} = 3$ we can compute up to second dimensional persistent homology. For a Rips stream, the parameter t_{max} is the maximum possible edge length. Since $t_{max} = 4$ is greater than the diameter ($\sqrt{10}$) of our point cloud, all edges will eventually form.

We compute and display the Betti intervals. Typically the last input for the method `Plex.plot` will be t_{max}, since there is no reason to display filtration times that we haven't computed.

```
>> intervals = Plex.Persistence.computeIntervals(ripsHouse);
>> Plex.plot(intervals, 'ripsHouse', 4)
```



ripsHouse: Dimension 0



ripsHouse: Dimension 1

The second dimensional Betti plot does not appear because there are no $Betti_2$ intervals. Check that these plots are consistent with the 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 $t = \sqrt{8}$.

Exercise 5.1.1. Change `ripsHouse` into an explicit stream

```
>> ripsExpl = Plex.makeExplicit(ripsHouse);
```

Check that you can display and edit stream `ripsExpl` using the methods of §3.

Torus example. Try the following sequence of commands. We select a 20×20 grid of noisy points from a torus and build the RipsStream `ripsT20`. The fourth command returns the total number of simplices in `ripsT20`.

```
>> pointsT20 = pointsTorus(20);
>> pdataT20 = EuclideanArrayData(pointsT20);
>> ripsT20 = Plex.RipsStream(0.001, 3, 0.9, pdataT20);
>> ripsT20.size
ans = 82831                  % Generally close to 80000
>> intervals = Plex.Persistence.computeIntervals(ripsT20);
>> Plex.FilterInfinite(intervals)
ans = BN{1, 2, 1}
>> Plex.plot(intervals, 'ripsT20', 0.9)
```

We do not include the Betti intrervals because the plots are very tall. 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.

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

Exercise 5.1.2. Slowly increase the values for t_{max}, d_{max}, or the grid length (20 above) and note how quickly `ripsT20.size` and the computation time grows. Separately increasing t_{max} from 0.9 to 1, d_{max} from 3 to 4, or the grid length from 20 to 22 each roughly doubles `ripsT20.size`.

Exercise 5.1.3. Find a planar dataset Z and a filtration value t such that $Betti_2(\text{Rips}(Z, t)) \neq 0$. Build a RipsStream to confirm your answer.

Exercise 5.1.4. Find a planar dataset Z and a filtration value t such that $Betti_6(\text{Rips}(Z, t)) \neq 0$. When building a RipsStream 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 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. We select 25 random landmarks from figure eight PointData instance `pdataF100`.

```
>> L1 = WitnessStream.makeRandomLandmarks(pdataF100, 25);
>> length(L1)
ans = 26
```

Vector `L` always has first entry zero. The remaining 25 entries contain the indices of the random landmark vertices.

The second method for selecting landmark points, called sequential maxmin, is a greedy inductive selection process. Pick the first landmark 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 $d(\,\cdot\,,\,\cdot\,)$

is the distance between the point and the set.

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].

JavaPlex does not yet have a command for sequential maxmin landmark selection, so we use the m-file `maxminLandmarks.m`. The first input is a matrix and the second input is the number of landmarks to be selected. The third input is either the character `'e'` or `'d'`. Use `'e'` in the EuclideanArrayData case where the first input should be interpreted as an $N \times n$ matrix of N points in $\mathbb{R}^n$. Use `'d'` in the DistanceData case where the first input should be interpreted as an $N \times N$ distance matrix for N points in an arbitrary metric space.

Case EuclideanArrayData. Recall the figure eight example.

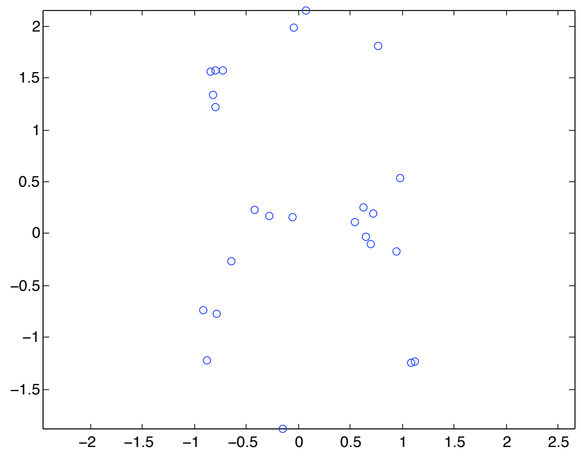
```
>> pdataF100 = EuclideanArrayData(pointsF100);
```

We select 25 sequential maxmin landmarks.

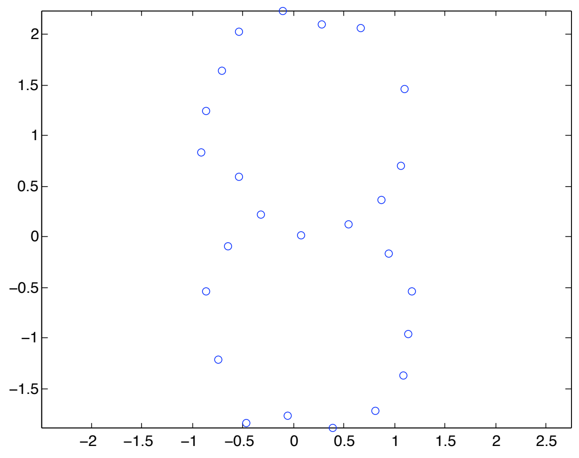
```
>> L2 = maxminLandmarks(pointsF100, 25, 'e');
```

Plot landmark points `L1` and `L2` to see the difference between random and sequential maxmin landmark selection.

```
>> pointsL1 = pointsF100(L1(2:end),:);
>> plot(pointsL1(:,1), pointsL1(:,2), 'o'), axis equal
>> pointsL2 = pointsF100(L2(2:end),:);
>> figure;   % This makes our next plot appear in a new window
>> plot(pointsL2(:,1), pointsL2(:,2), 'o'), axis equal
```



(a) Random landmarks



(b) Sequential maxmin

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

Case DistanceData. Recall the house example.

```
>> pdataHouseDD = DistanceData(distances);
```

We select 3 sequential maxmin landmarks.

```
>> L = maxminLandmarks(distances, 3, 'd')
L =
0
3
1
2
```

Given point cloud Z and landmark subset L, we define $\mathtt{R} = \max_{z \in Z}\big\{d(z, L)\big\}$. 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 WitnessStream or LazyWitnessStream instance.

Exercise 5.2.1. Let Z be the point cloud in Figure 1 from §4.1, corresponding to PointData instance `pdataHouse`. 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. **Subclass WitnessStream.** 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 $W(Z, L, t)$ 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 $W(Z, L, t)$ 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 $W(Z, L, t) \subset W(Z, L, s)$ whenever $t \leq s$. Note that a landmark point can serve as a witness point.

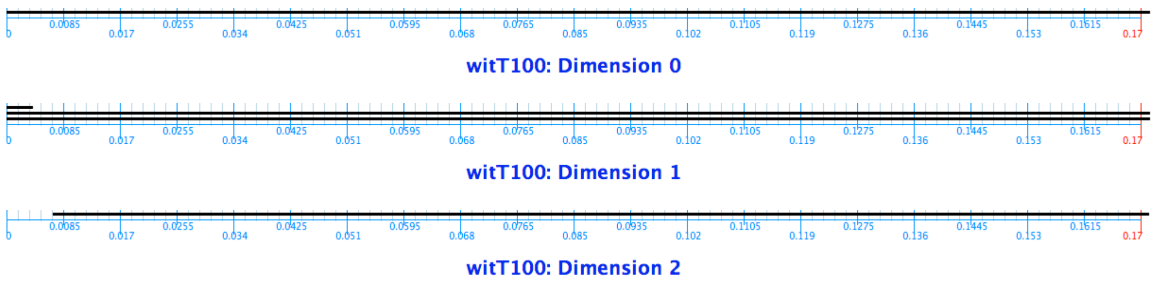
Exercise 5.3.1. Let Z be the point cloud in Figure 1 from §4.1, corresponding to PointData instance `pdataHouse`. Let $L = \{(1, 0), (0, 3), (-1, 0)\}$ be the landmark subset. Find by hand the filtration time for the edge between vertices $(1, 0)$ and $(0, 3)$. Which point or points witness this edge?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

Torus example. Let's build a WitnessStream instance for 100^2 points from a noisy torus, with 50 random landmarks. The fourth command returns the landmark covering measure R from §5.2. The fifth command returns our witness stream. The order of inputs is WitnessStream(δ, d_{max}, t_{max}, L, Z). Often the value for t_{max} is chosen in proportion to R.

```
>> pointsT100 = pointsTorus(100);
>> pdataT100 = EuclideanArrayData(pointsT100);
>> L = WitnessStream.makeRandomLandmarks(pdataT100, 50);
>> R = WitnessStream.estimateRmax(pdataT100, L)
R = 1.3188                   % Generally close to 1.2
>> witT100 = Plex.WitnessStream(0.001, 3, R/8, L, pdataT100);
>> witT100.size
ans = 3320                   % Generally close to 3000
```

We plot the Betti intervals.

```
>> intervals = Plex.Persistence.computeIntervals(witT100);
>> Plex.plot(intervals, 'witT100', R/8)
```



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 WitnessStream `witT100` contains approximately 3,000 simplices, fewer than the approximately 80,000 simplices in RipsStream `ripsT20`. This is despite the fact that we started with 100^2 points in the witness case, but only 20^2 points in the Rips case. This supports our belief that the witness stream returns good results at lower computational expense.

5.4. **Subclass LazyWitnessStream.** 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 $LW_\nu(Z, L, t)$ 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 $LW_\nu(Z, L, t)$ is defined as follows.

- the vertex set is L.
- for vertices a and b, edge $[ab]$ is in $LW_\nu(Z, L, t)$ 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 $LW_\nu(Z, L, t)$ if all of its edges are.

Note that $LW_\nu(Z, L, t) \subset LW_\nu(Z, L, s)$ 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 §4.1, corresponding to PointData instance `pdataHouse`. 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 $W(Z, L, t)$ is the same as the 1-skeleton of a lazy witness complex $LW_2(Z, L, t)$. As a consequence, $LW_2(Z, L, t)$ is the flag complex of $W(Z, L, t)$.

The following sequence of commands is typical.

```
>> L = WitnessStream.makeRandomLandmarks(pdata, numLands);
                                   % Or, sequential maxmin landmarks
>> R = WitnessStream.estimateRmax(pdata, L);
>> laz = Plex.LazyWitnessStream(δ, dmax, tmax, ν, L, pdata);
>> intervals = Plex.Persistence.computeIntervals(laz);
>> Plex.plot(intervals, 'laz', t_max)
```

Again, t_{max} is often chosen in proportion to `R`. In the next section we build a lazy witness stream on a dataset of range image patches.

6. Example with real data

We now do an example with real data. Double check that the files `pointsRange.mat` and `dct.m`, which accompany this tutorial, are in your Matlab directory.

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 length 25 vector 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 create a PointData instance using subclass EuclideanArrayData. We pick 50 sequential maxmin landmark points, find the value of `R`, and build the lazy witness stream with parameter $\nu = 1$.

```
>> pdataRange = EuclideanArrayData(pointsRange);
>> L = maxminLandmarks(pointsRange, 50, 'e');
>> R = WitnessStream.estimateRmax(pdataRange, L)
R = 0.7757                  % Generally close to 0.75
>> lazRange = Plex.LazyWitnessStream(0.001, 3, R/3, 1, L, pdataRange);
>> lazRange.size
ans = 20937                 % Generally between 10000 and 25000
>> intervals = Plex.Persistence.computeIntervals(lazRange);
>> Plex.plot(intervals, 'lazRange', R/3)
```

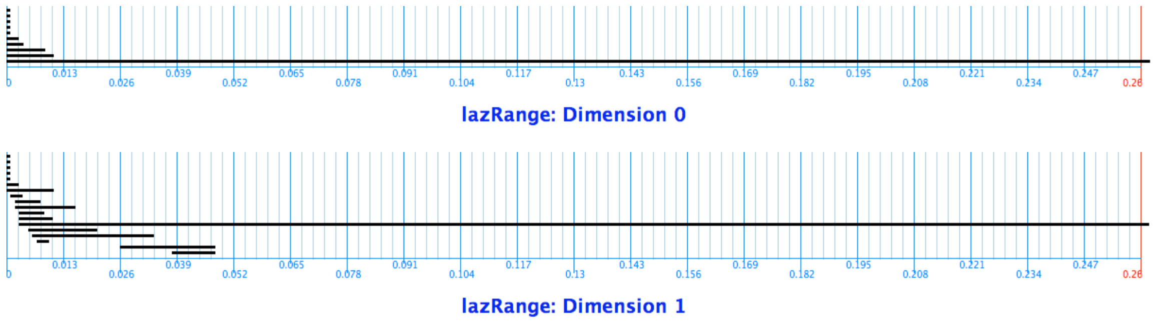


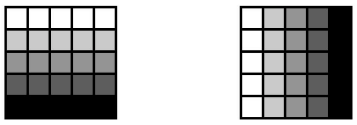
FIGURE 3. Betti intervals for `lazRange`, built from $X^5(300, 30)$

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 3. The method `dct.m` returns the DCT change-of-basis matrix for square patches of size specified by the input parameter.

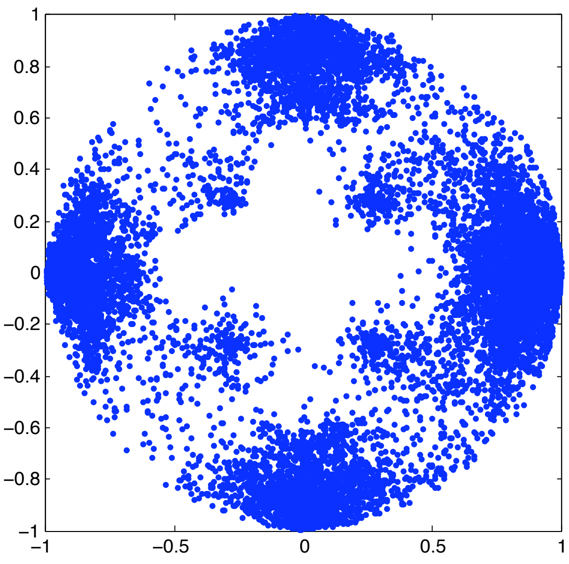
```
>> pointsRangeDct = pointsRange*dct(5);
```

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

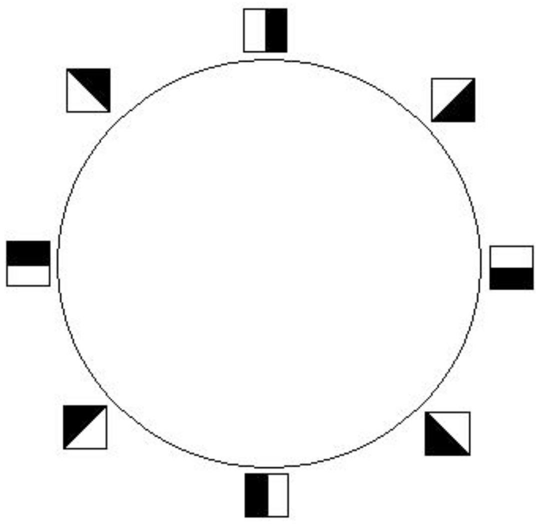


We plot the projection of `pointsRangeDct` onto the linear gradient DCT basis vectors.

```
>> plot(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 in Figure (b).

7. Remarks

7.1. **Scripts and functions with JavaPlex commands.** In order to include JavaPlex commands in an m-file script or function, include the command `import edu.stanford.math.plex.*;` as the first command of the script. For m-file functions, include this command as the second line - that is, as the line underneath the function header.

Writing scripts and functions is very useful. One example is the m-file `eulerCharacteristic.m`, which accepts an ExplicitStream as input and demonstrates two different methods for computing the Euler characteristic.

```
>> eulerCharacteristic(s6)
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 homology class. The current version of JavaPlex does not return representative cycles, though development versions of JavaPlex can. There is no guarantee that the produced representative will be geometrically nice.

7.3. **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 computer has a heap limit of approximately 128 megabytes. To increase your limit to, say, 256 megabytes, create a file named `java.opts` in your Matlab directory which contains the text `-Xmx256m` and then restart Matlab.

7.4. **From Java variables to Matlab variables.** Some JavaPlex commands return Java variables, when instead one might want output in the form of a Matlab variable, such as a matrix of numbers. One way to transform Java variables into Matlab variables is by using an m-file. For instance, in §**??** we used the m-file `interval2mat.m` to take an array of type PersistenceInterval into a Matlab matrix. Similarly, the m-file `betti2mat.m` accepts input of type `Plex.BettiNumbers` and returns a Matlab vector of integers. These commands are typically used as shown below, where `intervals` is an array of type PersistenceInterval.

```
>> intervalMatrix = interval2mat(intervals);
>> bettiVect = betti2vect(Plex.FilterInfinite(intervals));
```

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 §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 is 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 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. I have not included it 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 me at `henrya@math.stanford.edu` if you're interested in using `kDensity.m`.

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. Carlson, 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.