JPlex with BeanShell Tutorial

Henry Adams
henry.adams@colostate.edu
December 26, 2011

NOTE: As of 2011, JPlex is being phased out in favor of the software package Javaplex, which is also written in Java and has a tutorial. We suggest that JPlex users switch to Javaplex. The Javaplex code is available at `http://appliedtopology.github.io/javaplex/` and the tutorial is available at `http://www.math.colostate.edu/~adams/research/javaplex_tutorial.pdf`.

Contents

1. Introduction 2
1.1. JPlex 2
1.2. Accompanying files 2
1.3. Installation for BeanShell 2
2. Math review 3
2.1. Simplicial complexes 3
2.2. Homology 3
2.3. Filtered simplicial complexes 3
2.4. Persistent homology 3
3. Streams 3
3.1. Class SimplexStream 3
3.2. Subclass ExplicitStream and homology 3
3.3. Subclass ExplicitStream and persistent homology 5
3.4. ExplicitStream details 6
4. Point cloud data 7
4.1. Class PointData 7
4.2. Subclass EuclideanArrayData 7
4.3. Subclass DistanceData 7
4.4. Subclass Torus 8
5. Streams from point cloud data 8
5.1. Subclass RipsStream 8
5.2. Landmark selection 10
5.3. Subclass WitnessStream 10
5.4. Subclass LazyWitnessStream 11
6. Example with real data 12
7. Remarks 14
7.1. Scripts and functions 14
7.2. Representative cycles 14
7.3. Java heap size 14
Appendices 15
Appendix A. Dense core subsets 15
Index of JPlex and BeanShell commands 16
References 17

1. Introduction

1.1. **JPlex.** JPlex is a Java software package for computing the persistent homology of filtered simplicial complexes, often generated from point cloud data. The authors are Harlan Sexton and Mikael Vejdemo Johansson. This is a tutorial for using JPlex with BeanShell. There is a similar document for using JPlex with Matlab. Please email Henry with questions about this tutorial. Some of the exercises are borrowed from Vin de Silva's *Plexercises* at `http://comptop.stanford.edu/u/programs/Plexercises2.pdf`. Other sources of information about JPlex are the javadoc tree for the JPlex library (`http://comptop.stanford.edu/programs/jplex/files/javadoc/index.html`) and the CompTop website (`http://comptop.stanford.edu/programs/`).

The Matlab version of this tutorial contains more examples than the current BeanShell version. These examples use methods which are not part of the JPlex software package. These examples include: selecting points from a noisy figure eight or from a noisy torus in §4.2, selecting sequential maxmin landmark points in §5.2, plotting landmark points in §5.2, changing basis in §6, plotting point cloud projections in §6, an Euler characteristic demonstration, and computing dense core subsets in Appendix A. I hope to one day add these examples to the BeanShell tutorial.

If you are interested in JPlex, then you may also be interested in two other software packages that compute persistent homology. The first is Dionysus (`http://www.mrzv.org/software/dionysus`) by Dmitriy Morozov. The second is Javaplex (`http://appliedtopology.github.io/javaplex/`) by Andrew Tausz, Mikael Vejdemo Johansson, and Henry Adams. Andrew Tausz and I have written a similar tutorial for Javaplex, available at `http://www.math.duke.edu/~hadams/research/javaplex_tutorial.pdf`.

1.2. **Accompanying files.** This tutorial should be accompanied by the following files, available at the TMSCSCS website above.

(1) `commandListBsh.rtf`
(2) `exerciseAnswersBsh.rtf`
(3) `exerciseAnswersFigures.pdf`
(4) `PlexBshTutorial.pdf`
(5) `pointsRange.txt`
(6) `sampleScript.bsh`

PDF file (4) is this tutorial. Text file (1) lists all commands in this tutorial, so that you may copy and paste commands into the BeanShell window or into a script file. Text file (2) has answers for the exercises in this tutorial, and PDF file (3) contains some figures to accompany the answers. Text file (5) contains data. Bsh file (6) is a sample script of commands.

1.3. **Installation for BeanShell.** You must have a Java Virtual Machine installed. Copy `plex.jar` to your home directory or a subdirectory thereof. In a command window or shell, enter the following command.

```
java -cp plex.jar JPlex
```

The Graphical User Interface (GUI) window that pops up is a modified version of BeanShell. The prompt in this window is `plex>`. Confirm that JPlex is working with the following command.

```
plex> Simplex.makePoint(1, 2);
<<(2) 1>>
```

All output that BeanShell prints will be wrapped in "`< >`". Some methods in this tutorial print superfluous output, say of the form `<[[I@b3236f>` or
`<[Ledu.stanford.math.plex.PersistenceInterval$Float;@916ab8>`. Such output is not included in the text of this tutorial, even though it will appear in your GUI window.

Note: for fairly trivial technical reasons, the version of `plex.jar` that you downloaded from the TMSCSCS website may not give you the correct answers for the circle and 6-sphere examples in §3.2. Please email

Henry at `henry.adams@colostate.edu` to get a different plex.jar file which will give you the correct answers for these (and all other) 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 JPlex, filtered simplicial 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. Streams

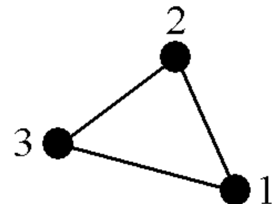
3.1. Class SimplexStream.

In JPlex, filtered simplicial complexes are called streams and are implemented by the class SimplexStream. The subclass ExplicitStream allows us to build a SimplexStream instance from scratch. In §5 we will learn about subclasses RipsStream, WitnessStream, and LazyWitnessStream, which construct SimplexStream instances from point cloud data.

3.2. Subclass ExplicitStream and homology.

Since JPlex is designed to compute persistent homology, it is not the most efficient software (notationally or computationally) for computing homology.

Circle example. Let's build a simplicial complex homeomorphic to a circle. We have three 0-simplices: [1], [2], and [3], and three 1-simplices: [12], [23], [31]. To build a simplicial complex in JPlex we simply build a stream in which all filtration times are zero. First we get an empty ExplicitStream instance.




```
plex> ExplicitStream s1 = new ExplicitStream();
```

Next we add simplices using the method `add`. Two inputs are required: one of type `double [][]` listing the simplices to be added, and one of type `double[]` listing the corresponding filtration times. We choose all filtration times to be zero as we are building a simplicial complex instead of a stream.

```
plex> double[][] cells = {{1},{2},{3},{1,2},{2,3},{3,1}};
plex> double[] filt = {0,0,0,0,0,0};
plex> s1.add(cells, filt);
```

Let's inspect what we've built. The object `s1.dump(`k`)` contains information about the k-simplices of `s1`. We display the 1-simplices and their filtration times.

```
plex> cells1 = s1.dump(1).C();
plex> for (int i = 0; i < cells1.length; i++) {print(cells1[i]);}
int []:  {1,2,}
int []:  {1,3,}
int []:  {2,3,}
plex> filt1 = s1.dump(1).F();
plex> for (int i = 0; i < filt1.length; i++) {print(filt1[i]);}
0.0
0.0
0.0
```

Our `s1`, like any ExplicitStream instance, has two states: open or closed. The state is automatically set to open whenever we edit or view the simplices of `s1`. We must manually close `s1` before computing its homology.

```
plex> s1.close();
```

The strange command names for computing homology will make sense later when we compute persistent homology.

```
plex> intervals = Plex.Persistence().computeIntervals(s1);
plex> Plex.FilterInfinite(intervals);
<BN{1, 1}>
```

The result `BN{1, 1}` means that `s1` has $Betti_0 = 1$ and $Betti_1 = 1$, which are the Betti numbers of a circle. (If you instead get BN{1}, email Henry at `henrya@math.stanford.edu` to get a different plex.jar file.)

6-sphere example. Let's build a 6-sphere, which is homeomorphic to the boundary of a 7-simplex. Adding the simplices manually as we did in the circle case would be very tedius: there are 8 vertices, 28 edges, 56 triangles, etc. Instead, we start with a 7-simplex that has no faces. We add its faces using the method `ensure_all_faces()`. Then, we remove the 7-simplex, leaving only its boundary.

```
plex> ExplicitStream s6 = new ExplicitStream();
plex> double[] cell = {1,2,3,4,5,6,7,8};
```

Note that we use a slightly simpler constructor for the `add` method here, with inputs of type `double[]` and `double`, since we add only one cell.

```
plex> s6.add(cell, 0);           % adds the 7-simplex
plex> s6.ensure_all_faces();
plex> s6.remove(cell);
```

How many 3-simplices did we avoid adding manually?

```
plex> s6.dump(3).C().length;
<70>                            % 70 3-simplices
```

We compute the homology.

```
plex> s6.close();
plex> intervals = Plex.Persistence().computeIntervals(s6);
plex> Plex.FilterInfinite(intervals);
<BN{1, 0, 0, 0, 0, 0, 1}>
```

We get nonzero Betti numbers $Betti_0 = Betti_6 = 1$. (If you instead get BN{1}, email Henry at **henrya@math.stanford.edu** to get a different plex.jar file.)

The following command tells us that we have been computing homology over the coefficient field $\mathbb{Z}_{11}$.

```
plex> Persistence.baseModulus();
<11>
```

We can instead compute over modulus 13 (or any other prime between 2 and 251).

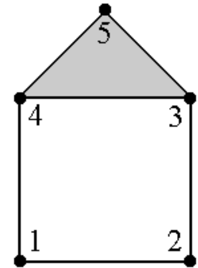
```
plex> Persistence.setBaseModulus(13);
```

Exercise 3.2.1. Build a simplicial complex homeomorphic to the torus. Compute its Betti numbers. *Hint:* You will need at least 7 vertices [7, page 107]. I recommend using a 3×3 grid of 9 vertices.

Exercise 3.2.2. 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 3.2.3. Build a simplicial complex homeomorphic to the projective plane. Find its Betti numbers over $\mathbb{Z}_2$ and $\mathbb{Z}_3$ coefficients.

3.3. **Subclass ExplicitStream and persistent homology.** 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.




```
plex> ExplicitStream house = new ExplicitStream();
plex> double[][] cells = {{1},{2},{3},{4},{5},{1,2},{2,3},{3,4},
{4,1},{3,5},{4,5},{3,4,5}};
plex> double[] filt = {0,0,0,0,1,0,0,0,0,2,3,7};
plex> house.add(cells, filt);
```

We compute the Betti intervals.

```
plex> house.close();
plex> intervals = Plex.Persistence().computeIntervals(house);
```

There are four intervals.

```
plex> intervals.length;
<4>
```

The intervals are indexed from 0 to 3. The fourth interval is a $Betti_1$ interval, starting at filtration time 3 and ending at 7.

```
plex> intervals[3].dimension;
<1>
plex> intervals[3].start;
<3.0>
plex> intervals[3].end;
<7.0>
```

Or, we can display the fourth interval all at once.

```
plex> intervals[3];
<[1:  (3.000000, 7.000000)]>
```

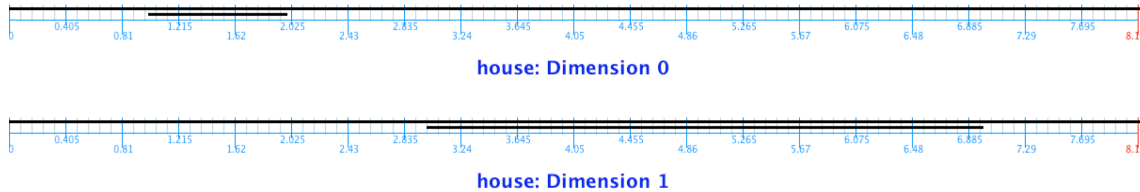
This 1-dimensional hole is formed by the three edges of the roof. It forms when edge $[4, 5]$ appears at filtration time 3 and closes when 2-simplex $[3, 4, 5]$ appears at filtration time 7.

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

```
plex> Plex.FilterInfinite(intervals);
<BN{1, 1}>
```

The method `Plex.plot` lets us display the intervals as a Betti barcode. The three inputs are `intervals`, a string to appear as a label, and the maximum filtration time (which may be adjusted $\pm 10\%$) for the plot.

```
plex> Plex.plot(intervals, "house", 8);
```



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.

The method `Plex.scatter` displays the same information as `Plex.plot` but in a different format: an interval starting at *start* and ending at *end* is plotted as the point $(start, end)$ in $\mathbb{R}^2$, which is necessarily above the diagonal.

```
plex> Plex.scatter(intervals,"house",8);
```

The produced figure is fairly large and so it is not included in this tutorial. For more information on scatter plots, see [6], especially Figure 7.

3.4. **ExplicitStream details.** We mention two remaining details about subclass ExplicitStream.

The methods `add` and `remove` 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 subsimplex $\sigma' \notin X(t)$, meaning that $X(t)$ is not a simplicial complex. The method `verify(true)` returns `true` if our stream is consistent and returns `false` with explanation if not.

```
plex> house.verify(true);
<true>
plex> double[] cell = {1,4,5};
plex> house.add(cell,0);
plex> house.verify(true);
<false>
```

The explanation below is not printed in the GUI window; it appears in pop-up message window.

```
Simplex <1, 4, 5> is present, but its face <1, 5> is not.
Simplex <1, 4, 5> has value 0.0000, but face <4, 5> has value 3.0000.
```

In §5 we will create SimplexStream instances that, unlike `house`, are not also ExplicitStream instances. To display or edit such streams, we will first need to use the method `makeExplicit`. See Exercise 5.1.1.

4. Point cloud data

4.1. Class PointData. A point cloud is a finite metric space, that is, a finite set of points equipped with a notion of distance. In JPlex, point cloud data is implemented by the class PointData. We detail two subclasses, EuclideanArrayData and DistanceData, that build PointData instances from different representations of a point cloud. In §5 we will learn how to build streams from a PointData instance.

4.2. Subclass EuclideanArrayData. This subclass is for a point cloud in a Euclidean space.

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

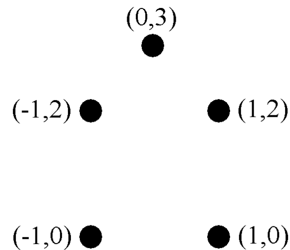


Figure 1. The house point cloud, stored in PointData instance `pdataHouse`

We create a PointData instance using these coordinates. The input to the EuclideanArrayData constructor, of type `double[][]`, lists the point coordinates.

```
plex> double[][] houseCoordinates = {{-1,0},{1,0},{1,2},{-1,2},{0,3}};
plex> pdataHouse = Plex.EuclideanArrayData(houseCoordinates);
```

Any PointData instance can display the number of data points and the distance between, say, points 1 and 3.

```
plex> pdataHouse.count();
<5>
plex> pdataHouse.distance(1, 3);
<2.8284271247461903>
```

The method `dimension` returns the dimension of the Euclidean space containing our points.

```
plex> pdataHouse.dimension();
<2>
```

The method `coordinate(`i`,`j`)` returns the j-th coordinate of point i. Points are indexed starting at one but coordinates are indexed starting at zero. We display the coordinates of point 5.

```
plex> pdataHouse.coordinate(5, 0);
<0.0>
plex> pdataHouse.coordinate(5, 1);
<3.0>
```

4.3. Subclass DistanceData. This subclass creates a PointData instance using a distance matrix. For a point cloud in Euclidean space, subclass DistanceData is generally less convenient than subclass EuclideanArrayData. However, subclass DistanceData can be used for a point cloud in an arbitrary metric space.

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. Don't forget you can copy and paste commands from `commandListBsh.rtf` into the GUI window, or enter them using a script!

```
plex> double[][] distances =
```

```
{{0,2,Math.sqrt(8),2,Math.sqrt(10)},
{2,0,2,Math.sqrt(8),Math.sqrt(10)},
{Math.sqrt(8),2,0,2,Math.sqrt(2)},
{2,Math.sqrt(8),2,0,Math.sqrt(2)},
{Math.sqrt(10),Math.sqrt(10),Math.sqrt(2),Math.sqrt(2),0}};
```

We create a PointData instance from this matrix.

```
plex> pdataHouseDD = Plex.DistanceData(distances);
```

Check that the methods `count` and `distance` return the same output with `pdataHouseDD` as with `pdataHouse`. The methods `dimension` and `coordinate` are not functional with the DistanceData subclass.

4.4. **Subclass Torus.** The following command constructs a 20×20 grid of points on the 2 dimensional unit torus. This torus is the identification space obtained by taking the square $[-1, 1] \times [-1, 1]$ and identifying the left and right and the top and bottom edges in an orientation-preserving manner.

```
plex> pdataT20 = Plex.Torus(20, 2);
```

Methods `count` and `distance` return the expected information. Method `dimension` returns the manifold dimension.

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 $t = 0$, δ, 2δ, 3δ, ..., $N\delta$, 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.

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 $\mathrm{Rips}(Z, t)$ is defined as follows:

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

Note that $\mathrm{Rips}(Z, t) \subset \mathrm{Rips}(Z, s)$ 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*.

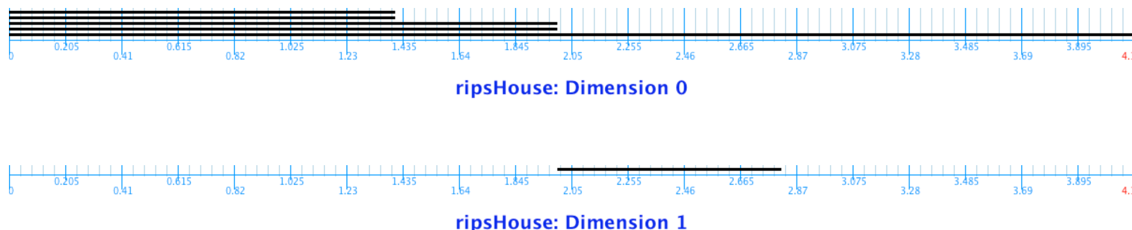
Let's build a Rips stream instance `ripsHouse` from the PointData instance `pdataHouse`. Note this stream is different than the ExplicitStream `house` we built in §3.3.

```
plex> 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.

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



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

```
plex> 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 points a torus and build the RipsStream `ripsT20`. The fourth command returns the total number of simplices in `ripsT20`.

```
plex> pdataT20 = Plex.Torus(20, 2);
plex> ripsT20 = Plex.RipsStream(0.001, 3, 0.5, pdataT20);
plex> ripsT20.size();
<4000>
plex> intervals = Plex.Persistence().computeIntervals(ripsT20);
plex> Plex.FilterInfinite(intervals);
<BN{1, 2, 1}>
plex> Plex.plot(intervals, "ripsT20", 0.5);
```

We do not include figures of the Betti barcodes because they are very tall: there are 400 intervals in dimension zero and 401 in dimension one. There is unrealistic uniformity among these intervals, due to the uniformity of our synthetic point cloud.

The diameter of the two dimensional unit torus is $\sqrt{8}$, so choosing $t_{max} = 0.5$ 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 profile $Betti_0 = 1$, $Betti_1 = 2$, $Betti_2 = 1$ 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. 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 5.1.3. 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 50 random landmarks from Torus instance `pdataT20`.

```
plex> L = WitnessStream.makeRandomLandmarks(pdataT20,50);
plex> L.length;
<51>
```

Output `L` of type `int[]` always has first entry zero. The remaining 50 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].

JPlex with BeanShell does not yet have a command for sequential maxmin landmark selection, though this can be done in Matlab.

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 WitnessStream or LazyWitnessStream instance.

Exercise 5.2.1. Let Z be the point cloud in Figure 1 from §4.2, 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 $\max\{d(l_0,z), d(l_1,z), ..., d(l_k,z)\} \leq t + m_k(z)$.

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.

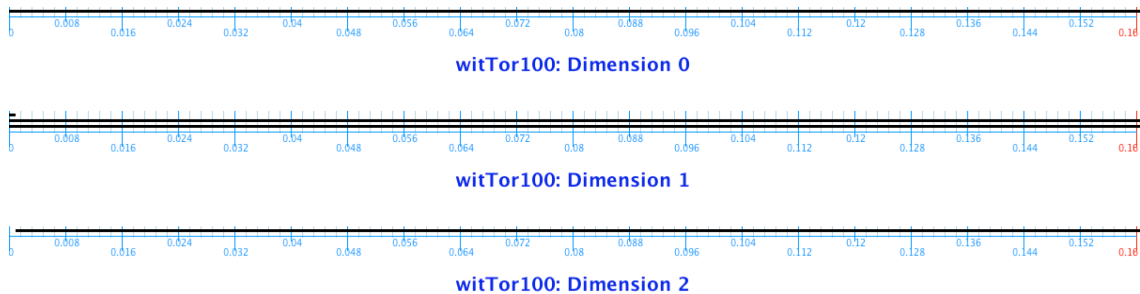
Exercise 5.3.1. Let Z be the point cloud in Figure 1 from §4.2, corresponding to PointData instance `pdataHouse`. Let $L = \{(1,0),(0,3),(-1,0)\}$ be the landmark subset. Find by hand the filtration time for the edge between vertices $(1,0)$ and $(0,3)$. Which point or points witness this edge? What 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 torus, with 50 random landmarks. The third command returns the landmark covering measure `R` from §5.2. The fourth 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`.

```
plex> pdataT100 = Plex.Torus(100,2);
plex> L = WitnessStream.makeRandomLandmarks(pdataT100,50);
plex> R = WitnessStream.estimateRmax(pdataT100, L);
<1.216492079160321>          % Generally close to 1.2
plex> witT100 = Plex.WitnessStream(0.001, 3, R/8, L, pdataT100);
plex> witT100.size();
<3053>                       % Generally close to 3000
```

We plot the Betti intervals.

```
plex> intervals = Plex.Persistence().computeIntervals(witT100);
plex> 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 4,000 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 $\max\big\{d(a,z),d(b,z)\big\} \leq t + m(z)$.
- 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 §4.2, 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.

```
plex> L = WitnessStream.makeRandomLandmarks(pdata, numLands);
plex> R = WitnessStream.estimateRmax(pdata, L);
plex> laz = Plex.LazyWitnessStream(δ, d_max, t_max, ν, L, pdata);
plex> intervals = Plex.Persistence().computeIntervals(laz);
plex> 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. Please copy the file `pointsRange.txt`, which should accompany this tutorial, into your home 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.

Text file `pointsRange.txt` is in fact $X^5(300, 30)$: each of the 15,000 lines contains 25 numbers. Feel free to open the file. The suffix `e-01` means that an entry is multiplied by 10^{-1}. It is not easy to visualize a circle by looking at these coordinates!

We create a PointData instance using subclass EuclideanArrayData from §4.2, except with a different constructor that accepts a filename as input.

```
plex> pdataRange = Plex.EuclideanArrayData("pointsRange.txt");
```

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

```
plex> L = WitnessStream.makeRandomLandmarks(pdataRange, 50);
plex> R = WitnessStream.estimateRmax(pdataRange, L);
<1.0096276221704195>             % Generally close to 1
plex> lazRange = Plex.LazyWitnessStream(0.001, 3, R/4, 1, L, pdataRange);
```

```
plex> lazRange.size();
<38936>                 % Generally between 20,000 and 200,000
```

The second command above is the only place where this example differs from [1], in which we select our landmark points via sequential maxmin.

This first command below takes up to one minute on my MacBook.

```
plex> intervals = Plex.Persistence().computeIntervals(lazRange);
plex> Plex.plot(intervals, "lazRange", R/4);
```

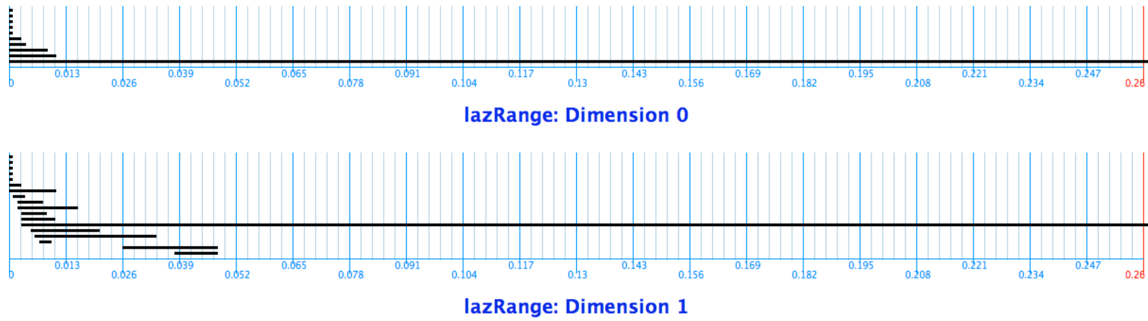
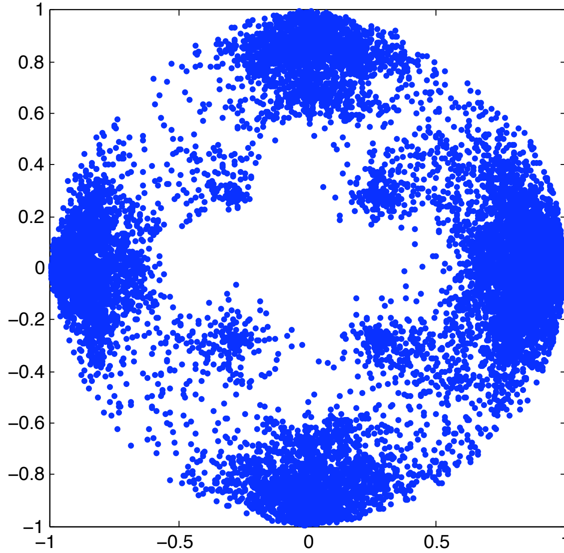


FIGURE 2. Betti intervals for `lazRange`, built from $X^5(300, 30)$

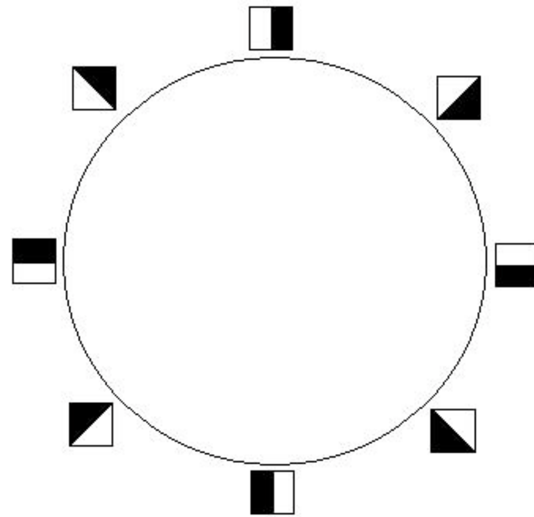
The plots above show that for a long range, the circle Betti numbers $Betti_0 = Betti_1 = 1$ are obtained. Your plot should contain a similar range. This is good evidence that the core subset $X^5(300, 30)$ is well-approximated by a circle.

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

Figure (a) below contains a projection of $X^5(300, 30)$ onto the two linear gradient DCT basis vectors. It shows the circle evidenced by Figure 2. This circle is called the range primary circle and is parameterized in Figure (b).



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



(b) Range primary circle

We change basis and plot Figure (a) in the Matlab version of this tutorial, but I am not sure how to do this yet in BeanShell.

7. Remarks

See the javadoc tree website given in §1.1 for more detailed BeanShell running instructions, including logging capabilities.

7.1. **Scripts and functions.** It is possible to write commands to a script and then call the list of commands at once. Save your script of commands to a file with ending `.bsh`, such as the sample script `sampleScript.bsh` which accompanies this tutorial. Put this file in your home directory. I know of two ways to run such a script. The first is from inside the BeanShell GUI window. Enter the following command.

```
plex> bg("sampleScript.bsh");
```

This will display pop-up figures. However, text output will only be produced for lines in your script enclosed by the command `print` (see `sampleScript.bsh` for an example). The second way to run a script is from the outer command window or shell. Enter the following command.

```
java -cp plex.jar JTerm < sampleScript.bsh
```

This will produce text output, even without using `print`. However, it will not display pop-up figures.

It is possible to write functions in BeanShell. I have not done very much of this yet. This is the main reason why there are fewer examples in the BeanShell version of the tutorial than in the Matlab version.

7.2. **Representative cycles.** The persistence algorithm that computes barcodes can also find a representative cycle for each homology class. The current version of JPlex does not return representative cycles, though development versions of JPlex can. There is no guarantee that the produced representative will be geometrically nice.

7.3. **Java heap size.** Depending on the size of your JPlex 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.

```
plex> Runtime.getRuntime().maxMemory();
ans = <66650112>
```

My computer has a heap limit of approximately 64 megabytes. To increase your limit to, say, 256 megabytes, close the GUI window. In the command window or shell, reopen BeanShell with the following modified command.

```
java -Xmx256m -cp plex.jar JPlex
```

Check in the GUI window that the heap size is now approximately 128 megabytes.

```
plex> Runtime.getRuntime().maxMemory();
ans = <266403840>
```

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.

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.

There is an example computing dense core subsets in the Matlab tutorial; I would like to add an analogous example to the BeanShell tutorial some day.

Index of JPlex and BeanShell commands

add, 3, 5, 6

bg, 14

close, 4, 5
coordinate, 7
count, 7

dimension, 5, 7
distance, 7
DistanceData, 7
dump, 4

end, 5
ensure_ all_ faces, 4
ExplicitStream, 3–5

java -cp plex.jar JPlex, 2
java -cp plex.jar JTerm, 14

length, 4, 5, 10

Persistence.baseModulus, 4
Persistence.setBaseModulus, 4
Plex.EuclideanArrayData, 7, 12
Plex.FilterInfinite, 4, 5, 9
Plex.LazyWitnessStream, 12
Plex.makeExplicit, 9
Plex.Persistence().computeIntervals, 4, 5, 8, 9, 11, 12
Plex.plot, 5, 8, 9, 11, 12
Plex.RipsStream, 8, 9
Plex.scatter, 6
Plex.Torus, 8, 9
Plex.WitnessStream, 11
print, 4

remove, 4
Runtime.getRuntime().maxMemory, 14

Simplex.makePoint, 2
size, 9, 11, 12
start, 5

verify, 6

WitnessStream.estimateRmax, 11, 12
WitnessStream.makeRandomLandmarks, 10–12

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.