Simplicial complexes for topological data analysis

Marc Glisse

This project is too long, you do not have to do everything. Section 2 is guided. Section 3 is completely independent and much more open. Preferred languages are Python and C++, but you can feel free to use any other language, including Java, and use a simple file format of your design for input and output. If the language is not Python, C++ or Java, make an effort so the code is readable by someone who does not know the language. Some Python scripts are provided for visualization, you are free to modify them, and you do not have to use them at all.

1 Background

Topological data analysis is covered in INF556, but here are some basics so you can realize this project. A lot of the text in this document is context, interesting to understand why we are doing this, but not at all necessary to perform the tasks.

You are already familiar with graphs, which contain a vertex set and an edge set. A *simplicial complex* is a generalization in higher dimension. In addition to vertexes (dimension 0) and edges (dimension 1), it also contains triangles (dimension 2), tetrahedra (dimension 3), and more generally k-simplexes defined by $k+1$ points. The *faces* of a k-simplex are the simplexes defined by subsets of those $k+1$ points (the faces of a tetrahedron are 4 triangles, 6 edges and 4 vertexes). Just as in a graph the extremities of an edge must be in the vertex set, in a simplicial complex the faces of a simplex must also be in the complex. There are no self loops (repeated vertices), no hyperedges with multiplicity, no orientation (edges AB and BA are the same), etc.

Persistent homology is a tool from algebraic topology that allows to track the evolution of the topology (connected components, loops, cavities) of a growing object. For an input set of points $p_1, \dots, p_n$, we can consider the balls of radius r: $B(p_1, r), \dots, B(p_n, r)$. If the points were sampled on some continuous object, say a circle, the hope is that for a well chosen r the union of balls will look similar to a thicked version of the circle (an annulus) that is *equivalent* to the circle (it deformation retracts to it). However, we will not fix r here and instead track the evolution as r grows: first many connected components, that progressively merge when the disks start overlapping, at some point a loop appears, and later this loop disappears as its interior is filled, see Fig. 2. Working directly with a union of continuous objects is hard, so we replace it with its *nerve*, that is a simplicial complex with a vertex for each ball, an edge between 2 vertexes if the 2 balls intersect, and more generally a k-simplex when the corresponding $k+1$ balls have a point in common (or equivalently if there exists a ball of radius r that contains those $p_{i_1}, \dots, p_{i_{k+1}}$, where the center of this ball is the

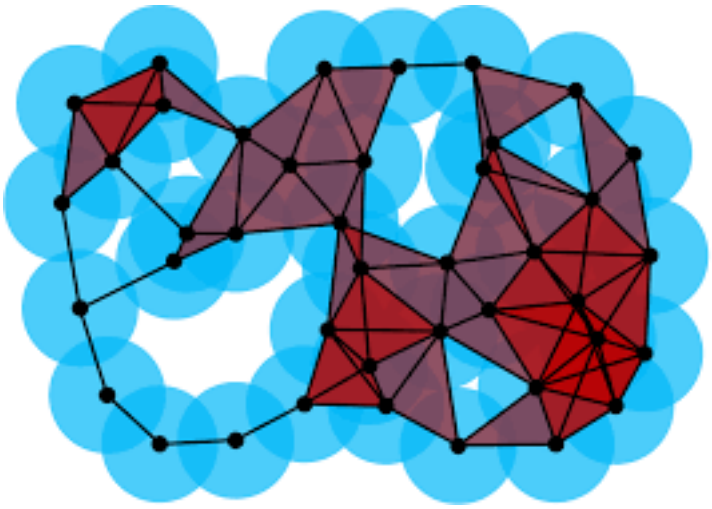


Figure 1: A union of balls, and its nerve the Čech complex.

common point). This simplicial complex is call the *Čech complex* of parameter r, see Figs. 1 and 3. In order to represent this complex for all values of r in some interval $[0, R]$, we actually build the largest complex (parameter R, possibly infinite) and associate to each simplex an *insertion time* or *filtration value* which is the smallest r where it appears. This is called a *filtered simplicial complex* and can then be fed to some algorithm to compute persistent homology.

2 Čech complex

While working with an arbitrary dimension would be ideal, you may limit yourself to $\mathbb{R}^3$ for simplicity.

2.1 General remarks

We denote C^k the subcomplex of the Čech complex where we keep only the simplexes of dimension at most k, and C^k_l where we keep only those whose filtration value is less than the limit l.

All tuples of points define a simplex of the Čech complex (of parameter ∞), and the filtration value of this simplex is the radius of its *minimal enclosing ball* (MEB), the smallest ball that contains these points.

2.2 LP-type problems

Task 1. *Implement the computation of the minimal enclosing ball (MEB) of a set of points using one (or more) of the algorithms from* `https://en.wikipedia.org/wiki/LP-type_problem`.

As a building block, you may need to compute the circumcenter of a non-degenerate set of points $p_0, \ldots, p_k$ for $k \leq d + 1$. One way to do this is to

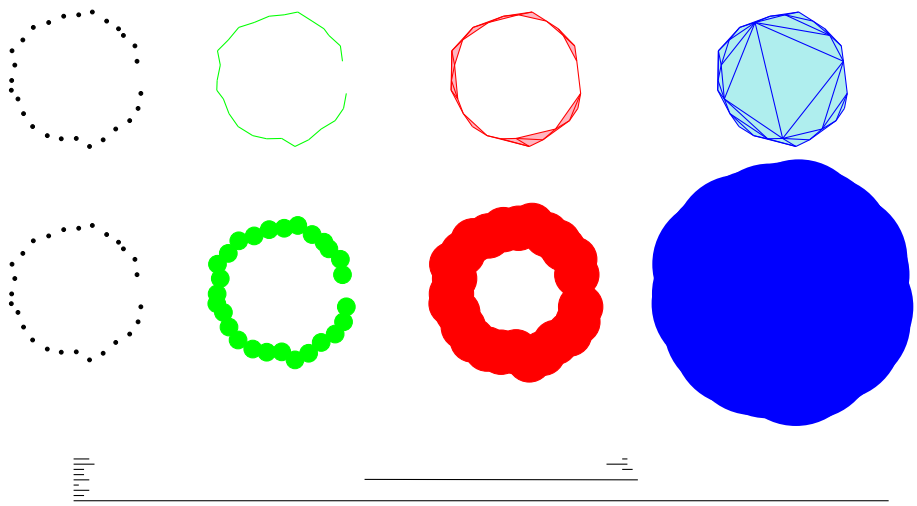


Figure 2: An α-complex filtration, the corresponding union of balls, and their persistence barcode. The bars starting at 0 represent connected components, quickly there is just one left. The other bars represent loops. When we are nearly filling the interior, 2 balls from opposite sides will touch, splitting the hole into 2 parts, so there are short-lived extra loops around that time.

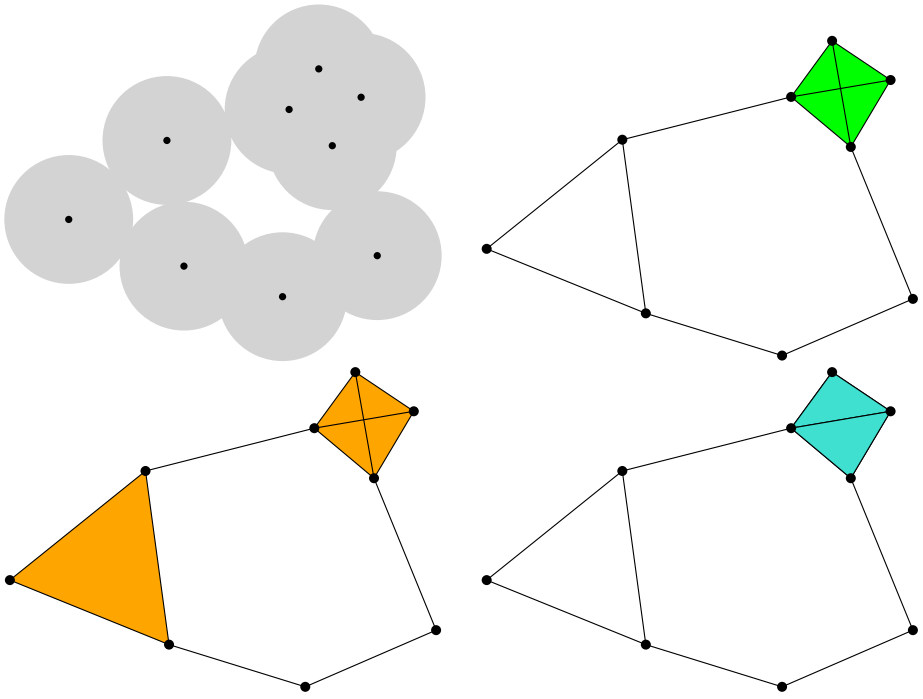


Figure 3: For the same point set and the same parameter r: the union of balls, the Čech complex, the Rips complex, and the α-complex. Čech and Rips contain a tetrahedron.

first write the circumcenter as a barycenter of the p_i with unknown weights, and notice that being equidistant from p_0 and p_i can be written as a linear equation. It then remains to solve a linear system (using an existing solver or writing your own) to find the barycentric weights, and finally the center and radius.

Because geometry is hard and floating-point arithmetic has limited precision, you may want to avoid testcases with non-generic position: no 3 points on the same line, no 4 points on the same circle, etc.

Testcases for MEB:

1. For a single point, the center is the point itself, with a radius of 0.
2. For 2 points, the center is the midpoint, and the radius is half of the distance between the points.
3. For 3 points with coordinates `(-10,0,0)`, `(10,0,0)`, `(0,1,0)`, the center is `(0,0,0)` and the radius 10.
4. For 3 points with coordinates `(-5,0,0)`, `(3,-4,0)`, `(3,4,0)`, the center is `(0,0,0)` and the radius 5.
5. For 4 points with coordinates `(5,0,1)`, `(-1,-3,4)`, `(-1,-4,-3)`, `(-1,4,-3)`, the center is `(0,0,0)` and the radius $\sqrt{26} \approx 5.09902$.

You can then experiment with adding more points inside the sphere and changing the order of the points, which should not change the MEB.

2.3 Construction

Task 2. *Given a set of n points in $\mathbb{R}^d$, implement a naive algorithm that enumerates the simplexes of C^k and their filtration values.*

For instance, for the 4 points of example 5 above, you should get (the printing format is irrelevant)

```
( 0 ) -> [0]
( 1 ) -> [0]
( 2 ) -> [0]
( 3 ) -> [0]
( 2 1 ) -> [3.53553]
( 1 0 ) -> [3.67423]
( 3 2 ) -> [4]
( 2 0 ) -> [4.12311]
( 3 0 ) -> [4.12311]
( 2 1 0 ) -> [4.39525]
( 3 2 0 ) -> [4.71495]
( 3 1 ) -> [4.94975]
( 3 2 1 ) -> [5]
( 3 1 0 ) -> [5.04975]
( 3 2 1 0 ) -> [5.09902]
```

Task 3. *Given a set of n points in $\mathbb{R}^d$, implement an algorithm that enumerates the simplexes of C^k_l and their filtration values. This algorithm should be less naive and not compute the MEB of all $(k+1)$-tuples of points, at least when l is small.*

You can use for instance https://geometrica.saclay.inria.fr/team/Marc.Glisse/tmp/inf421/sc.py to plot the vertices, edges and triangles of C_l^k. It is called `./sc.py --complex cplx.txt --coordinates coord.txt plot3d` where each line of `cplx.txt` looks like `5 12 14` to indicate a triangle connecting vertices 5, 12 and 14, and each line of `coord.txt` looks like `-1.5 2 0.7` for the coordinates of that point. For example, if `cplx.txt` contains

```
0 1
0 2
1 2
0 1 2
2
0 4
3
```

and `coord.txt` contains

```
0 0 0
0 1 0
0 0 1
1 0 0
1 1 1
```

we see Fig. 4.

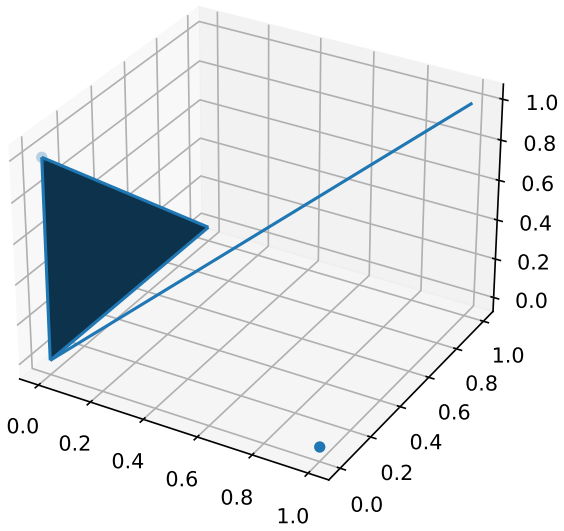


Figure 4: Graphic output of `sc.py`. You can rotate the view with the mouse.

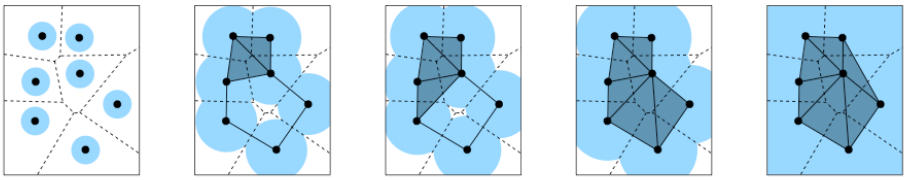


Figure 5: A sequence of α-complexes.

2.4 α-complex

The Čech complex grows too quickly to be used when l is not so small. However, there exists a smaller, *equivalent* (in a topological sense) alternative called the α-complex. Its definition is similar to the Čech complex, but instead of having a ball around each point, we only keep the part of the ball that is in the *Voronoi* cell of this point. Except at the boundaries, it removes the redundancies in the covering, without changing the union. We then compute the nerve of those truncated balls and call it the *α-complex*, see Fig. 5. More concretely, for a $k+1$-tuple of points, we consider the smallest ball that has those points on its boundary and no point in its interior. The radius of this ball is the filtration value of the corresponding k-simplex. If no such ball exists, the simplex is not part of the α-complex.

For example, for 3 points of coordinates $(0,5,0),(3,4,0),(-3,4,0)$, the triangle is in the α-complex, and its filtration value is 5. If we add a 4th point $(0,0,4)$, that triangle is still in the complex, but with a filtration value larger than 5. If we add a 5th point $(0,0,-4)$, that triangle is not in the complex anymore.

Task 4. *Reuse the LP-type algorithm with new parameters in order to determine if a simplex is in the α-complex and its filtration value. Note that this is less standard than for the MEB, you need to explain how this new problem fits in the framework.*

Task 5. *Given a set of n points in $\mathbb{R}^d$, implement an algorithm that enumerates the simplexes of dimension at most k and filtration value at most l of the α-complex and their filtration values.*

If you have reached this point, a plot showing the difference between the Čech complex and the α-complex would be a nice conclusion.

If we use a datastructure that can efficiently return the list of points that are contained in a ball (the nearest neighbors), can we use it to avoid some work? If you implement it, you can reuse any existing library for nearest neighbors (or range searching).

3 Rips complex

3.1 Definition and motivation

(This section is for context and can be skipped)

As you may have noticed, computing the filtration value of simplexes of dimension more than 1 in a Čech complex is complicated. As an approximation, people often use the simpler *Rips complex*. It has the same vertices and edges as the Čech complex and the filtration value of the edges is still half of their length. However, for simplexes of higher dimension, this is a *flag complex* (or clique complex), that is, $k+1$ points define a k-simplex of the Rips complex of parameter r if and only if the complete graph on those $k+1$ vertices is in the complex. In other words, as for the Čech complex, all $(k+1)$-tuples of points define a k-simplex of the Rips complex for r large enough, but the filtration value is now half of the diameter of the simplex, something much easier to compute than the radius of the MEB. See Fig. 3 for a difference between the two complexes. While the Rips complex is "wrong" in some sense, it is an approximation of the Čech complex: the Čech complex of parameter r is included in the Rips complex of parameter r, which is included in the Čech complex of parameter $2r$. A nice property of the Rips complex is that its graph (its simplexes of dimension 0 and 1) is a compact way to represent it.

There exist efficient algorithms to compute the persistent homology of flag complex filtrations, which in particular do not need to store the whole list of simplexes in memory. However, they still need to iterate over all the simplexes, which can be a large number. Our goal here is, before sending the graph to some persistence code, to simplify the graph in a way that does not affect the result of the persistence computation, without computing all the higher dimensional simplexes. To give some intuition, the domination condition described below guarantees that adding the edge at that time is useless (the clique complex defined by the graph with the edge deformation retracts to the one without the edge), so we do not need to insert this edge so soon, we can delay it until a time where it may become useful again (and in particular not dominated). Removing just a few edges from the graph can have a dramatic impact on the number of simplexes in the simplicial complex: a full Rips complex has 2^n simplexes, while removing just 1 edge removes 2^{n-2} simplexes, 25% of the total!

3.2 Simplification

3.2.1 Graph optimization

The main object in this section is a graph $G = (V, E)$, with a function $f : E \to \mathbb{R}$ called the filtration value or insertion time (f is not necessarily injective). Both the input and the output are such a pair of a graph and a function. We can interpret it as a sequence of graphs $G_{t_1} \subset G_{t_2} \subset \ldots \subset G_{t_m}$ where $G_{t_i} = (V, E_{t_i})$, $E_t = \{e \in E \mid f(e) \leq t\}$ and $\{t_1, \ldots, t_m\} = f(E)$. For a vertex v, its neighborhood at time t is $N_t[v] = \{u \in V \mid uv \in E_t \text{ or } u = v\}$. Similarly, the neighborhood of an edge uv at time t is $N_t[uv] = N_t[u] \cap N_t[v]$. We say that an edge e is *dominated* by a vertex v at time t if v is not an extremity of e and $N_t[e] \subseteq N_t[v]$.

We are given a set of rules, allowed transformations on a graph:

1. If $f(e) = t_i$ and e is dominated at t_i, we can set $f(e) = t_{i+1}$ (t_{i+1} is the smallest value larger than t_i in $f(E)$).

2. If $f(e) = t_m$ (the largest value) and e is dominated at t_m, we can remove e from the graph.

Task 6. *The goal is to transform an input graph into a new graph that is smaller, without a huge running time or memory use. You need to come up with an algorithm, explain why it works, bound its complexity, implement it and test it. If you want, you can try to find a parallel algorithm and experiment to show how well it scales.*

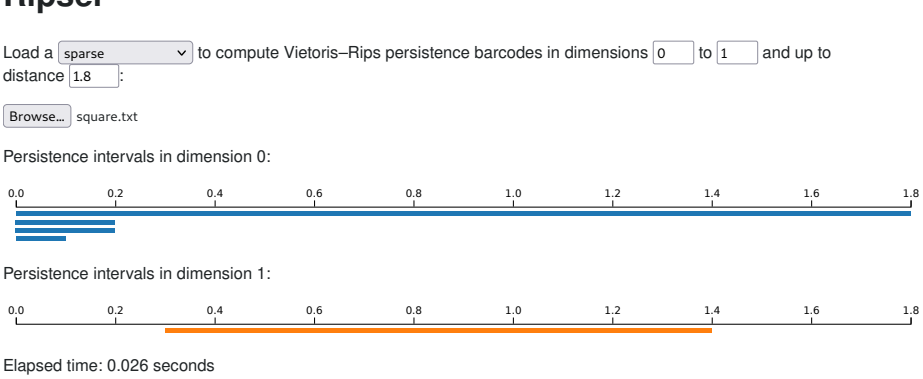
Note that the best datastructure depends on the algorithm you come up with, starting by implementing some specific graph datastructure before you have an idea how you are going to use it may prove a waste of time.

3.2.2 Testing

One way to test that the output filtration is valid is to check that the resulting persistence barcode is the same for the input and output. `https://geometrica.saclay.inria.fr/team/Marc.Glisse/tmp/ripser/`[1] with option "sparse" and "up to" a number larger[2] than t_m is a convenient way to compute it in your web browser[3] from a file where each line looks like `5 12 3.14` to represent an edge connecting vertex 5 to vertex 12 at time 3.14. For instance with

```
0 1 0.3
0 2 0.2
1 2 1.7
1 3 0.2
2 3 0.1
0 3 1.4
```

you should see



Now you can try removing the line `1 2 1.7` and notice that this smaller graph has exactly the same persistence barcode. Indeed, the edge $(1, 2)$ is dominated by vertex 0 (and by vertex 3) at time 1.7.

There are many ways to compute this from various languages (Gudhi for C++ or Python, javaPlex for Java or Matlab, Eirene for Julia, etc), but learning how to use them may take a lot of time.

[1] The original is at `https://live.ripser.org/` but is currently missing the "sparse" option which most closely matches our need.

[2] But not too large either since it determines the scale of the plot

[3] The computation happens entirely on your machine, the files are not sent to any remote server.

Here are some testcases on which you can try your code. If an optimal solution is mentioned, it is in no way required that your code be able to find it. You should start with small examples with few edges so you can follow what is going on step by step and manage to debug your code, and only move on to larger examples to test the speed once you are convinced it is valid.

1. The example with 4 vertices described above.
2. Complete graph: build a complete graph on k vertices, with the same filtration value for every edge. The optimal solution is to output a tree ($k-1$ edges). The graph has to remain connected.
3. Regular $2n$-gon. Place points at the vertices of a regular polygon with an even number of vertices and build the Rips filtration: complete graph, and the filtration value of an edge is its (Euclidean) length. The only edges that can be removed are the diameters, and you cannot remove all of them, only $n-1$.
4. Random points in a square. Generate $2n$ random numbers and use them as x and y coordinates for n points, compute the Rips filtration. You should be able to remove many edges here.

When you have coordinates, you can also use for instance https://geometrica.saclay.inria.fr/team/Marc.Glisse/tmp/inf421/scf.py to plot the edges as in Fig. 6. The edges (complex) have the same format as for Ripser, and the coordinates are one point per line, with whitespace-separated numbers. The same script, with option `intervals` instead of `plot2d`, and `--flag-dimension 2` (to indicate that it needs to compute the triangles), can also be used to get the persistence barcode in text format where the first number is the dimension.

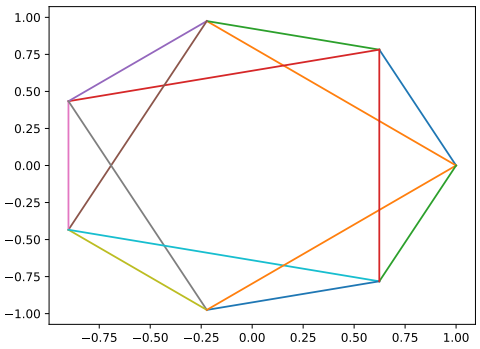


Figure 6: Plot of some edges with 2d coordinates using `./scf.py --complex edges.txt --coordinates point_coord.txt plot2d`