Simplicial complexes for topological data analysis

Marc Glisse

I recommend reading the whole subject (including the last section) before starting to program.

1 Background

Topological data analysis is covered in CSC_51056_EP, 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 have been 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 thickened 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 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.

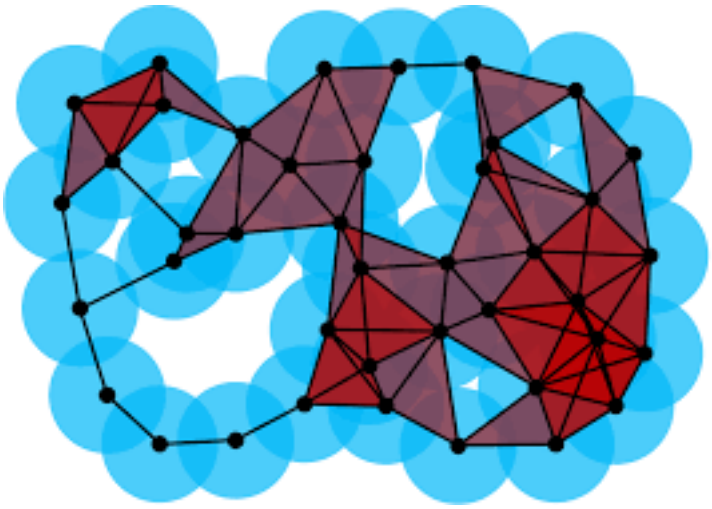

Figure 1: A union of balls, and its nerve the Čech complex.

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 $p_0, \dots, p_k$ for $k \leq d + 1$. One way to do this is to 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.

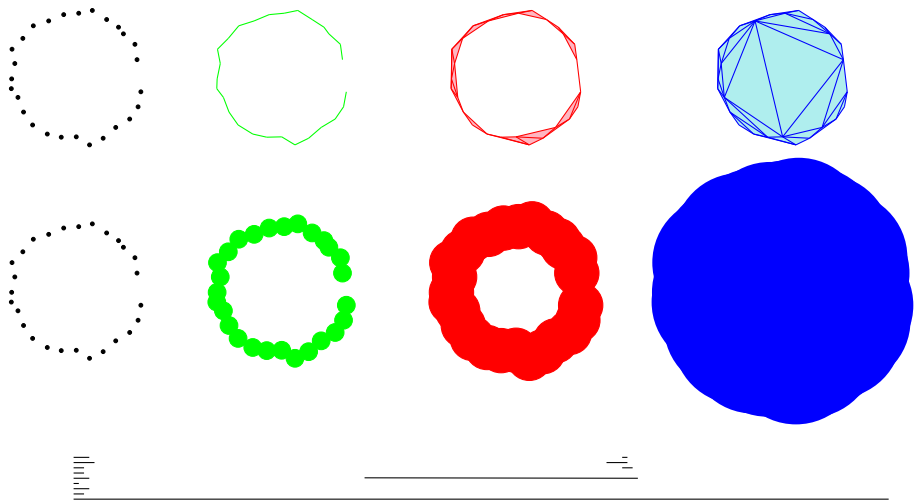

Figure 2: An α-complex filtration, the corresponding union of balls, and their persistence barcode. The bars starting at 0 represent connected components, quickly there is just one left. The other bars represent loops. When we are nearly filling the interior, 2 balls from opposite sides will touch, splitting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

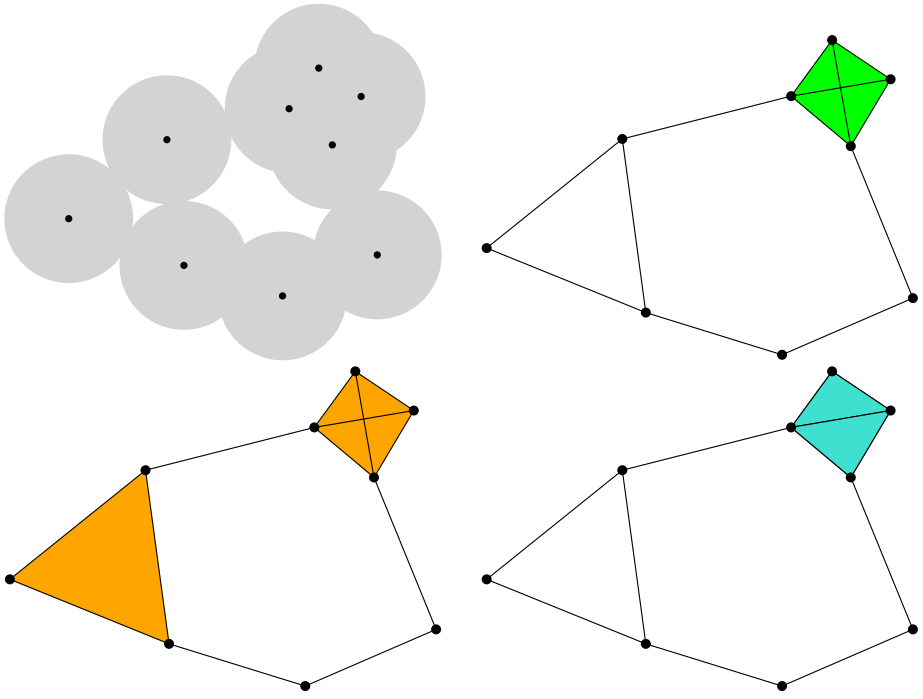

Figure 3: For the same point set and the same parameter r: the union of balls, the Čech complex, the Rips complex (a simplified variant not considered in this project), and the α-complex. Čech and Rips contain a tetrahedron.

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 may for instance use the property that if a simplex is not in the complex, no simplex containing it can be in the complex.

In order to visualize your output, you can use for instance sc.py (requires `matplotlib`) 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.

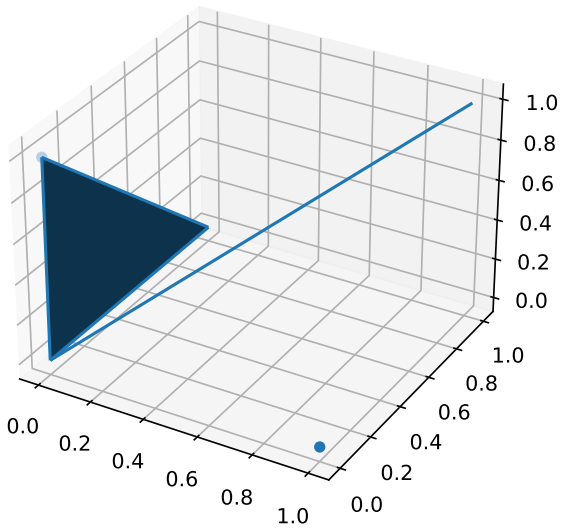

Figure 4: Graphic output of `sc.py`. You can rotate the view with the mouse.

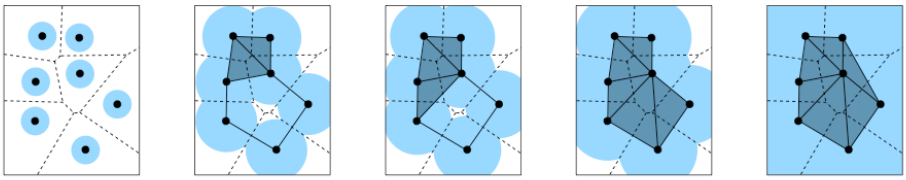

Figure 5: A sequence of α-complexes.

3 α-complex

As in the previous section, you may limit yourself to $\mathbb{R}^3$, although supporting arbitrary dimensions would be ideal.

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.*

Be wary of any shortcut you took with the LP-type algorithm for MEB, which may not be valid for this problem.

Task 5. *Given a set P 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.*

Note that the standard way to build the α-complex is to build the Delaunay triangulation, then compute the filtration value of each simplex, by decreasing dimension, and finally drop the simplexes whose dimension or filtration value is larger than what we need. Here, we purposedly deviate from this practice to avoid computing the full Delaunay triangulation, which is costly when the ambient dimension d is large (5 or 6 is already large and it is hard to go much beyond 10).

If you have reached this point, a plot showing the difference between the Čech complex and the α-complex would be a nice conclusion. A benchmark comparing the running time and the number of simplexes for both complexes on some datasets could also be interesting.

3.1 Bonus

Please only consider this part after the rest of the project is well polished, with good tests.

The algorithm is rather brutal, we consider here some possible optimizations (you may add more if they make sense). Note that some optimizations may be incompatible. Since the goal of these changes is performance, they need to be compared to previous algorithms through some benchmark. It is ok if some optimization turns out not to help, although you should try to understand why.

1. Since the tests for a lot of simplexes are independent, if you already know how to use parallelism, try to compute many of them in parallel. If you have never used parallelism before, it may be best to skip this.

2. If we only care about simplexes of filtration value at most l, and we look for those that have the point p as a vertex, the points of P that are too far from p are irrelevant, they cannot be a vertex of such a simplex, and they cannot modify the existence or filtration value of such a simplex either. After specifying what "too far" means, you can use a range searching (often advertized as *nearest neighbors*) library or implement your own (you can go for a naive version in that case, that compares the distances to all the points), and apply the previous construction to a smaller set of points.

3. Each vertex often has few neighbors (other vertexes connected to this one through an edge), and we would like to take advantage of that. For a point p, let N (originally empty) denote the list of its neighbors. We process the points of $P \setminus \{p\}$ one by one. For a point q, we check if it is a neighbor of p in $N \cup \{p, q\}$. If it isn't, it won't be a neighbor in P either and we can forget it. If it is, it may still not be a neighbor of p in P, but we can use it to find a new neighor: we have a ball passing through p and q and centered at c that does not contain any point of N. Among the balls centered on the segment pc and passing through p, we find the largest one whose interior does not contain any point of P. On its boundary, it must have a new neighbor of p that we can add to N. The goal of this approach is to reduce the number of constraints in the LP-type problems we solve.

4 Boring generalities

Feel free to use any reasonably common programming language, but whatever the language, make an effort so the code is readable even by someone who does not know this particular language. Unless otherwise specified, you are welcome to use the basic datastructures and algorithms provided by your language (Java ArrayList, Python dictionaries, sorting, etc), including some external libraries (like NumPy, or some library providing a hash map if your language does not provide one by default). However, you should obviously avoid using a library

that solves the whole problem or even a large part of it, contact me if you have a doubt. Since the whole point of this project is for you to practice, you should also avoid looking at existing solutions.

Some Python scripts are provided for visualization, you are free to modify them, and you do not have to use them at all.

The exact interface of functions and classes is up to you. Coming up with interfaces that are convenient to use is an important part of programming. In particular, sharing some code between different tasks in helper functions would probably be better than copy-pasting large blocks of code. The exact datastructures are also up to you, if you are asked to store a list of points, you are free to use a sorted vector, a binary tree or a hash set instead of the informal "list", or to replace points with their index or their distance to the origin, as long as this allows you to compute the right result efficiently.

Testing is an essential part of any programming project. While the subject may suggest some tests, it is up to you to come up with good tests, for small intermediate functions and for the whole project, that help build confidence in the result. The tests definitely need to be part of your submission.

You have to put everything in a single archive (ZIP, or compressed TAR), instructions will come later on where to upload this. This archive must contain your report as a PDF file, all the sources (not just the main code, but also the tests, generation of datasets, etc), and a README helping me understand in which file I can find what, and possibly instructions on how to run the code (I should be able to re-run your experiments easily). If you have a lot of code in a single file, for instance a Jupyter notebook, make sure that it is well organized and easy to navigate, with sections separating the algorithms from the tests. The report should not contain any code (it can contain pseudo-code though), and in particular no picture of code.

Separately (you don't have to include it in the archive), you should prepare slides for a presentation of no more than 10 minutes (strict). The presentation (like the report) should focus on what you have done, the originality of your approach. It will be followed by questions.