Intro. to Gudhi

Tao Hou, CS 410/510

Sketch

- Overview
- Installation
- Python API

Gudhi overview

- From official website (https://gudhi.inria.fr/):
  - *a generic open source **C++ library**, with a **Python interface**,*
  - *for Topological Data Analysis (**TDA**) and Higher Dimensional Geometry Understanding.*
  - *offers state-of-the-art data structures and algorithms to construct simplicial complexes and compute persistent homology*
- Developed by Inria from France

C++ vs. Python

- C++ programs: Fast to execute, hard to develop
- Python programs: Slow in execution, easy to develop

C++ vs. Python

- C++ programs: Fast to execute, hard to develop
- Python programs: Slow in execution, easy to develop
- Gudhi was developed by C++, i.e., all the intensive computations are in C++; you can also use Gudhi directly from C++

C++ vs. Python

- C++ programs: Fast to execute, hard to develop
- Python programs: Slow in execution, easy to develop
- Gudhi was developed by C++, i.e., all the intensive computations are in C++; you can also use Gudhi directly from C++
- Provides Python interface
  - You can use the Gudhi C++ codes from Python
  - *Fast* to execute, *easy* to develop
  - Pybind11, Cython

Installation

- The official website recommended using Conda
  - https://gudhi.inria.fr/python/latest/installation.html
  - You can use both GUI and commandline to install packages on Conda

Installation

- The official website recommended using Conda
  - https://gudhi.inria.fr/python/latest/installation.html
  - You can use both GUI and commandline to install packages on Conda
- For installing Conda on linux (especially ubuntu, debian):
  - If sudo apt-get doesn't work (e.g., if you don't have access to the writing permission of the root directory)
  - Use wget to get the installation script and bash run the script:
    - wget https://repo.anaconda.com/archive/Anaconda3-2023.07-2-Linux-x86_64.sh
    - bash Anaconda3-2023.07-2-Linux-x86_64.sh
    - The link can be retrieved from: https://www.anaconda.com/download/

Gudhi python interface

Documentation

- https://gudhi.inria.fr/python/latest/

Categories of the API

- Cell complex data structures
  - Simplicial complex, Cubical complex
- Filtrations
  - Delauney filtration, Rips filtration, Witness complex, Cover complex (Mapper, Nerve and Graph Induced Complexes)
- Compute persistent homology
- Persistent barcode processing
  - Bottleneck distance, Wasserstein distance, Persistence representations (vectorizations and kernels for ML)
- Persistence graphical tools
  - display persistence barcode, diagram or density
- Others: Point cloud utilities (read from file, subsample, find neighbors), generating datasets, etc.

Simplicial complex (in math)

- A collection of sets, where each set contains vertices
  - Additionally, it should satisfy that subsets of each set in the collection also belongs to the collection
- A set *a* is a subset of a set *b*: *a* is a **face** of *b* and *b* is a **coface** of *a*

SimplexTree

- https://gudhi.inria.fr/python/latest/simplex_tree_user.html
- https://gudhi.inria.fr/python/latest/simplex_tree_ref.html
- “The simplex tree is an efficient and flexible data structure for representing general (filtered) simplicial complexes.”
- Here, a “filtered” simplicial complex is just a simplicial complex K equipped with a filtration $\mathcal{F}$ (aka. the filtration eventually “grows” into K)

Simplices in a SimplexTree

- A simplex is represented as a python list of integers:
  - A tetrahedron (3-simplex): [5, 9, 10, 17]
  - Every simplex returned from SimplexTree, and every simplex feed into SimplexTree as parameter is a list of integers
  - For a bunch of simplices, then it is a list of lists of integers

Simplices in a SimplexTree

- A simplex is represented as a python list of integers:
  - A tetrahedron (3-simplex): [5, 9, 10, 17]
  - Every simplex returned from SimplexTree, and every simplex feed into SimplexTree as parameter is a list of integers
  - For a bunch of simplices, then it is a list of lists of integers
- Advantage: Claritity, no confusion

Simplices in a SimplexTree

- A simplex is represented as a python list of integers:
  - A tetrahedron (3-simplex): [5, 9, 10, 17]
  - Every simplex returned from SimplexTree, and every simplex feed into SimplexTree as parameter is a list of integers
  - For a bunch of simplices, then it is a list of lists of integers
- Advantage: Claritity, no confusion
- This also makes mapping a simplex to a certain property a little tricky (e.g., you build a dual graph for a surface, and want to look up the vertex id corresponding to a triangle)
  - Would have to use a python dictionary (e.g., {‘A’: 1, ‘B’: 2})
  - The key for the dictionary is the simplex (the list of ints)
  - Have to convert the list into a tuple (`tuple(l)`) or a string (`str(l)`) first so that `l` can be used as a key for the dict

Simplices in a SimplexTree

- Some more info: in the C++ interface, each simplex has an integer id; you can manipulate a simplex simply by its id
- You can get its list of vertices by invoking a method
- This can be convenient in certain cases
- ~~The python interface can be a little tricky if you switch from it C++ version at first~~

SimplexTree

Example

```
import gudhi
st = gudhi.SimplexTree()
if st.insert([0, 1]):
    print("[0, 1] inserted")
if st.insert([0, 1, 2], filtration=4.0):
    print("[0, 1, 2] inserted")
if st.find([0, 1]):
    print("[0, 1] found")
result_str = 'num_vertices=' + repr(st.num_vertices())
print(result_str)
result_str = 'num_simplices=' + repr(st.num_simplices())
print(result_str)
print("skeleton(2) =")
for sk_value in st.get_skeleton(2):
    print(sk_value)
```

- https://gudhi.inria.fr/python/latest/simplex_tree_user.html

Building a SimplexTree

- Build from scratch using insert():
  - Declare a SimplexTree object
  - Insert a simplex to the complex: automatically add the faces of the simplices

Building a SimplexTree

- Build from scratch using insert():
  - Declare a SimplexTree object
  - Insert a simplex to the complex: automatically add the faces of the simplices
- You can assign a filtration value when you insert a simplex:

Building a SimplexTree

- Build from scratch using insert():
  - Declare a SimplexTree object
  - Insert a simplex to the complex: automatically add the faces of the simplices
- You can assign a filtration value when you insert a simplex:
  - Recall a simplex-wise filtration is sequence of simplicial complexes where each consecutive ones K_{i-1} and K_i differ by a single simplex denoted by σ_i

$$\mathcal{F} : \varnothing = K_0 \xhookrightarrow{\sigma_1} K_1 \xhookrightarrow{\sigma_2} \cdots \xhookrightarrow{\sigma_{m-1}} K_{m-1} \xhookrightarrow{\sigma_m} K_m = K$$

Building a SimplexTree

- Build from scratch using insert():
  - Declare a SimplexTree object
  - Insert a simplex to the complex: automatically add the faces of the simplices
- You can assign a filtration value when you insert a simplex:
  - Recall a simplex-wise filtration is sequence of simplicial complexes where each consecutive ones K_{i-1} and K_i differ by a single simplex denoted by σ_i

$$\mathcal{F} : \varnothing = K_0 \xhookrightarrow{\sigma_1} K_1 \xhookrightarrow{\sigma_2} \cdots \xhookrightarrow{\sigma_{m-1}} K_{m-1} \xhookrightarrow{\sigma_m} K_m = K$$

  - which can also be considered as a sequence of insertions of the simplices $\sigma_1, \sigma_2, \ldots, \sigma_m$

Building a SimplexTree

- Build from scratch using insert():
  - Declare a SimplexTree object
  - Insert a simplex to the complex: automatically add the faces of the simplices
- You can assign a filtration value when you insert a simplex:
  - Recall a simplex-wise filtration is sequence of simplicial complexes where each consecutive ones K_{i-1} and K_i differ by a single simplex denoted by σ_i

$$\mathcal{F} : \varnothing = K_0 \xhookrightarrow{\sigma_1} K_1 \xhookrightarrow{\sigma_2} \cdots \xhookrightarrow{\sigma_{m-1}} K_{m-1} \xhookrightarrow{\sigma_m} K_m = K$$

  - which can also be considered as a sequence of insertions of the simplices $\sigma_1, \sigma_2, \ldots, \sigma_m$
  - Also recall that each K_i in $\mathcal{F}$ is associated with an "α value", so we have that the "filtration value" of σ_i is the "α value" of K_i

Building a SimplexTree

- More often, we would not build a SimplxTree (or a filtration) from scratch like above, but would rather build a filtration (i.e., an SimplexTree object) from a filtration building class (such as RipsComplex)

Info. of SimplexTree

- dimension()
  - returns the dimension of the simplicial complex
- filtration(simplex)
  - returns the filtration value for a given simplex
- find(simplex)
  - returns if the simplex was found in the simplicial complex or not
- get_filtration()
  - returns a generator with simplices and their given filtration values sorted by increasing filtration values
  - a generator is an object in python which you can loop over like a list
- get_simplices()
  - only difference from previous is that simplices are not sorted by filtration values
- num_simplices()
- num_vertices()

Adjacency info. in SimplexTree

- get_boundaries()
  - returns a generator with the boundaries of a given simplex
- get_cofaces()
  - get all the cofaces of a simplex (the star)
  - get the cofaces in a certain dimension for a simplex
- get_star()
  - same as get_cofaces() without parameters

Compute persistence barcode for a SimplexTree

- persistence():
  - computes and returns the persistence of the simplicial complex
  - you should **use 2 for homology_coeff_field**
  - **min_persistence** can be used for length “cut-off”
  - return type: list of pairs (dimension, pair(birth, death))
  - The returned pair is a pair of filtration value, e.g., for Rips filtration, it is the distance where a homology class is born and the distance where the homology class dies (not the index of the simplices in the simplex-wise filtration)
- compute_persistence()
  - similar to previous but doesn’t have any return

Compute persistence barcode for a SimplexTree

- persistence_intervals_in_dimension():
  - similar to persistence() but only return the persistence intervals in a specific dimension
  - requires compute_persistence() function to be launched first
- persistence_pairs():
  - another form of persistence barcode, where each pair is a pair of simplices that generates a persistence interval
  - requires compute_persistence() function to be launched first
  - however, we don’t really have the indices of the simplices in the simplex-wise filtration (this can be useful but I don’t find a way to do this in the current python interface of Gudhi)
  - note: you can definitely do this in the C++ interface

Other functions of SimplexTree

- betti_numbers()
  - returns the Betti numbers of the simplicial complex K (the last complex in $\mathcal{F}$)
  - betti_numbers function requires compute_persistence() function to be launched first
- write_persistence_diagram()
  - writes the persistence intervals of the simplicial complex in a user given file name

RipsComplex (class for building a Rips filtration in Gudhi)

- Rips filtration:
  - Given a set of points (point cloud) with pair-wise distance for the points, it returns you a filtration
  - The persistence of the filtration gives you an idea of the topology of the underlying space of the point cloud
- https://gudhi.inria.fr/python/latest/rips_complex_user.html

RipsComplex (class for building a Rips filtration in Gudhi)

- Rips filtration:
  - Given a set of points (point cloud) with pair-wise distance for the points, it returns you a filtration
  - The persistence of the filtration gives you an idea of the topology of the underlying space of the point cloud
- https://gudhi.inria.fr/python/latest/rips_complex_user.html
- You can create a RipsComplex in two ways in Gudhi:
  - By specifying coordinates for the points
  - By providing a distance matrix

RipsComplex (class for building a Rips filtration in Gudhi)

- Rips filtration:
  - Given a set of points (point cloud) with pair-wise distance for the points, it returns you a filtration
  - The persistence of the filtration gives you an idea of the topology of the underlying space of the point cloud
- https://gudhi.inria.fr/python/latest/rips_complex_user.html
- You can create a RipsComplex in two ways in Gudhi:
  - By specifying coordinates for the points
  - By providing a distance matrix
- What you eventually get from the class is a **SimplexTree object** representing the rips filtration (using the create_simplex_tree() method)

RipsComplex

- Constructor:
  - points (List[List[float]]) – A list of points in d-dimension.
  - distance_matrix (List[List[float]]) – A distance matrix (full square or lower triangular)
  - max_edge_length (float) – Maximal edge length. We do not consider those complexes in the filtration whose distance value is greater than this given max (for saving computing time)
  - sparse (float) – If this is not None, it switches to building a *sparse Rips filtration* and this parameter represents the approximation factor "ε"

More about the approximation factor "ε"

- A Rips complex can easily become huge, even if we limit the length of the edges and the dimension of the simplices. One way to further reduce the size is to use the sparse/approximation factor "ε"

More about the approximation factor "ε"

- A Rips complex can easily become huge, even if we limit the length of the edges and the dimension of the simplices. One way to further reduce the size is to use the sparse/approximation factor "ε"
- ε should be a value between 0 and 1 which controls how much the size is reduced and how similar the produced PD is to the one without size reduction
  - ε close to 0: the size of the Rips filtration is less reduced and the produced PD is similar to the unreduced one
  - ε close to 1: the size of the Rips filtration is much more reduced and the produced PD is a worse approximate to the unreduced one

More about the approximation factor "ε"

- A Rips complex can easily become huge, even if we limit the length of the edges and the dimension of the simplices. One way to further reduce the size is to use the sparse/approximation factor "ε"
- ε should be a value between 0 and 1 which controls how much the size is reduced and how similar the produced PD is to the one without size reduction
  - ε close to 0: the size of the Rips filtration is less reduced and the produced PD is similar to the unreduced one
  - ε close to 1: the size of the Rips filtration is much more reduced and the produced PD is a worse approximate to the unreduced one
- The approximate factor of PD: $1/(1-\varepsilon)$

Function of RipsComplex

The only non-constructor member function provided by RipsComplex:

- **create_simplex_tree**(*max_dimension=1*)
  - **max_dimension** (*int*) – we build the simplices for the complex only up to the given dimensioin
  - returns: A simplex tree encoding the Vietoris–Rips filtration
- This is the step that truly “builds” the filtration. The constructor only provides some parameter for the building

Examples of building Rips

- Build from a point cloud
- Build from a distance matrix
- Build from a correlation matrix
  - To the opposite of a distance matrix, which measures how “different” two objects are from each other, a correlation (a number from 0 to 1) measures how “similar” two objects are
  - Using 1 minus the correlation (similarity) we can have something similar to the distance

A critical advice to further reduce the size (and speed up)

- No matter the dimension of persistence you want to compute, you should always do following (to speed up):
  - when invoking create_simplex_tree() to build the Rips complex, only build up to dimension 1 (a graph) to get the SimplexTree
  - use collapse_edges() (of the SimplexTree) to reduce the size of this graph
  - call expansion() (of the SimplexTree) to get a filtration of a suitable dimension to compute its persistence

A critical advice to further reduce the size (and speed up)

- No matter the dimension of persistence you want to compute, you should always do following (to speed up):
  - when invoking create_simplex_tree() to build the Rips complex, only build up to dimension 1 (a graph) to get the SimplexTree
  - use collapse_edges() (of the SimplexTree) to reduce the size of this graph
  - call expansion() (of the SimplexTree) to get a filtration of a suitable dimension to compute its persistence
- This trick gives the same persistence diagram as one would get with a plain use of *RipsComplex*, with a complex that is often significantly smaller and thus faster to process (we will look at an example later)

Other ways to reduce the size

- Subsample the point cloud using methods in:
  - https://gudhi.inria.fr/python/latest/point_cloud.html

Displaying persistence barcode

- https://gudhi.inria.fr/python/latest/persistence_graphical_tools_user.html
- https://gudhi.inria.fr/python/latest/persistence_graphical_tools_ref.html
- **plot_persistence_barcode**()
- **plot_persistence_diagram**()
- **plot_persistence_density**()
- Noticeable parameters:
  - persistence
  - persistence_file
  - max_intervals
- If you want to explore some more advanced displaying options, you would have to know more about `matplotlib`

Code example

- `Rips.py`
- Try: max_intervals=100 for showing barcode
- Try: only dim 1 for barcode using “pd[1,:,:]”, and max_intervals=50
- Try: sparse=0.3
- Try: subsample the points (https://gudhi.inria.fr/python/latest/point_cloud.html)

Other filtrations

- Delaunay complex
- Witness complex
- Cover complexes
- Tangential complex

Bottleneck distance

- https://gudhi.inria.fr/python/latest/bottleneck_distance_user.html
- Two implementations whose usages are similar
- Notice the “shape” of the PD array (only two indices for the np array without the PD dimensions)

Wasserstein Distance

- https://gudhi.inria.fr/python/latest/wasserstein_distance_user.html
- A generalization of bottleneck distance

Bottleneck distance

- Define a "**partial matching**" η:
  1. Select the same number of points from D_1 and D_2: let these points "perfectly" match to each other
  2. For the remaining points, we let them be "unmatched"

Bottleneck distance

- Define a "**partial matching**" η:
  1. Select the same number of points from D_1 and D_2: let these points "perfectly" match to each other
  2. For the remaining points, we let them be "unmatched"
- Define $cost(\eta)$ as the maximum of the two below:
  1. Maximum L_∞-distance of each two matched point p and q in D_1, D_2:

     $$\max_{x,y \text{ matched in } \eta}\{d_\infty(x,y)\}$$

     where $d_\infty(x,y) = \max\{|b_1 - d_1|, |b_2 - d_2|\}$ for $x = (b_1, d_1)$, $y = (b_2, d_2)$
  2. Maximum length $d - b$ (aka. distance to diagonal) of each unmatched point (b, d)

Bottleneck distance

- Define a "**partial matching**" η:
  1. Select the same number of points from D_1 and D_2: let these points "perfectly" match to each other
  2. For the remaining points, we let them be "unmatched"
- Define $cost(\eta)$ as the maximum of the two below:
  1. Maximum L_∞-distance of each two matched point p and q in D_1, D_2:

     $$\max_{x,y \text{ matched in } \eta}\{d_\infty(x,y)\}$$

     where $d_\infty(x,y) = \max\{|b_1 - d_1|, |b_2 - d_2|\}$ for $x = (b_1, d_1)$, $y = (b_2, d_2)$
  2. Maximum length $d - b$ (aka. distance to diagonal) of each unmatched point (b, d)
- The **bottleneck distance** is then defined as follows:

$$d_{\mathrm{B}}(D_1, D_2) = \min_{\eta \text{ over all partial matchings}}\{cost(\eta)\}$$

aka. the minimum cost of all partial matchings

Wasserstein distance

- Define a "**partial matching**" η:
  1. Select the same number of points from D_1 and D_2: let these points "perfectly" match to each other
  2. For the remaining points, we let them be "unmatched"
- Define $cost_p(\eta)$ as:

$$\sqrt[p]{\sum_{x,y \text{ matched in } \eta} \Big(d_p(x,y)\Big)^p + \sum_{(b,d) \text{ unmatched in } \eta} (d-b)^p}$$

where $\Big(d_p(x,y)\Big)^p = (b_1 - d_1)^p + (b_2 - d_2)^p\}$ for $x = (b_1, d_1), y = (b_2, d_2)$

- The p-th **Wasserstein distance** is then defined as follows:

$$d_p^W(D_1, D_2) = \min_{\eta \text{ over all partial matchings}} \{cost_p(\eta)\}$$

aka. the minimum cost of all partial matchings

Useful but shall not go over (for time being)

- Persistence representations (vectorizations and kernels for ML)
  - May go over this after covering machine learning

Datasets module

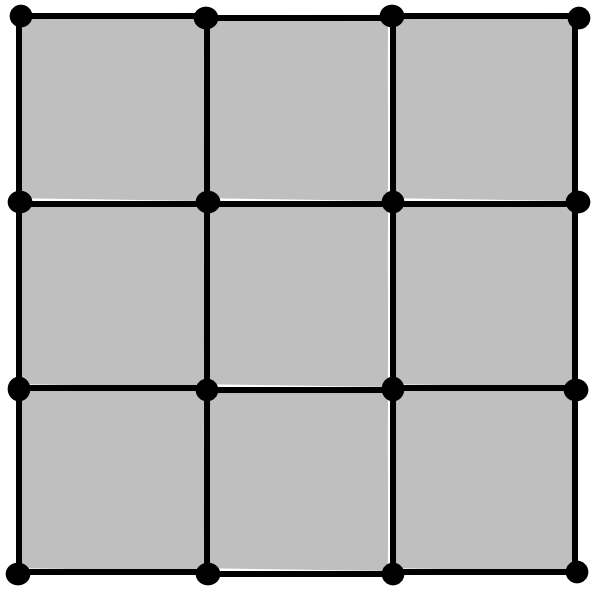
- https://gudhi.inria.fr/python/latest/datasets.html
- You can use the functions to directly generate sphere, torus, or the bunny

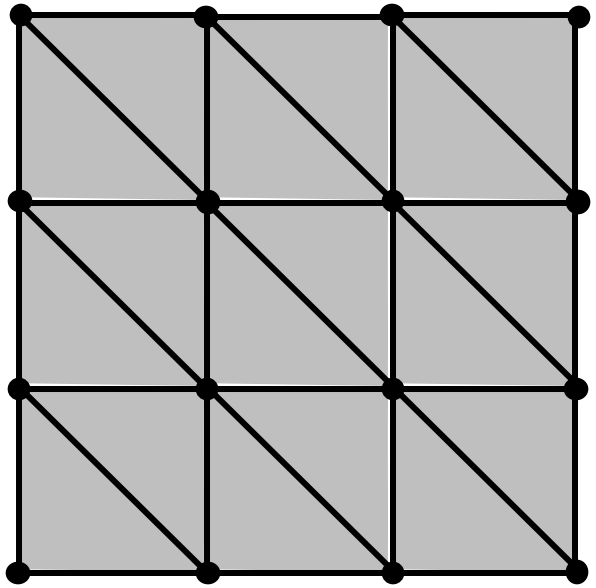
CubicalComplex

- A “cell” complex (generalization of simplicial complex) consisting of vertices (dim-0), edges (dim-1), squares (dim-2), cubes (dim-3)
- Very useful for processing images (2-complex) and 3D volume datasets (3-complex)
- A filtration is built from a **function** from a regular 2D/3D grid to the real values using the “sublevelset filtration”
- https://gudhi.inria.fr/python/latest/cubical_complex_user.html
- https://gudhi.inria.fr/python/latest/cubical_complex_ref.html

CubicalComplex

- Aka, we do not subdivide the grid





A very important (and tricky) point about CubicalComplex

- Two ways for constructing it:
  1. Specify an array named '**top_dimensional_cells**':
     - pixels (or voxels) are the squares (or cubes), aka. highest-dimensional cells
     - when building the sublevelset filtration, each square (or cube) is added one by one following the increasing order of their value, and their faces (e.g., edges or vertices) are added accordingly
  2. Specify an array named '**vertices**'
     - pixels (or voxels) are the vertex, aka. lowest-dimensional cells
     - the sublevelset filtration is one we have seen

A very important (and tricky) point about CubicalComplex

- Two ways for constructing it:
  1. Specify an array named '**top_dimensional_cells**':
     - pixels (or voxels) are the squares (or cubes), aka. highest-dimensional cells
     - when building the sublevelset filtration, each square (or cube) is added one by one following the increasing order of their value, and their faces (e.g., edges or vertices) are added accordingly
  2. Specify an array named '**vertices**'
     - pixels (or voxels) are the vertex, aka. lowest-dimensional cells
     - the sublevelset filtration is one we have seen
- The persistence of the two ways of interpreting a cubical dataset is not too different indeed because "shapes" of the spaces in the filtration are basically the same

Compute persistence for CubicalComplex

- Similar to SimplexTree, use persistence()
- We also have other similar functions as in SimplexTree

About the C++ interface of Gudhi

- Much more powerful
- Learning curve is steeper
- Besides the reference manual, you should also thoroughly study the examples provided

Some other libraries I know (or have used)

- Ripser (https://github.com/Ripser/ripser): good for computing rips filtrations
- Dionysus (https://mrzv.org/software/dionysus2/): also very versatile (e.g., vineyard); provide both C++ and Python interfaces
- Phat (https://github.com/blazs/phat): very fast in computing persistence; tricky to build
- Homcloud (https://homcloud.dev/index.en.html): Good for computing representatives for persistence
- FZZ (https://github.com/taohou01/fzz): fast for zigzag persistence; only in C++; uses phat
- More: https://cat-list.github.io/