Computational and applied topology, tutorial.

Paweł Dłotko
Swansea University
(Version 3.14, and slowly converging.)

July 24, 2018

Chapter 1

Introduction.

If you are reading this, it probably means that you got interested in applied and computational topology and perhaps you want to use it to solve your problems. Or perhaps you are searching for inspiration to your lectures. In any case, I hope you will find this resource useful. Some of the examples and ideas have been taken from my friends in the filed. In each of those cases I tried to make it clear in the test. Please note that the script is under constant development and it is not (nor have ever been) intended to be a complete, rigorous book. It is rather a collection of evolving ideas, anecdotes and examples to illustrate what is *applied and computational topology*. Or at least what s my view on it. I hope you will enjoy it!

Completion of this lecture will require you to get your hands dirty with data and implementation. Most of the concepts are illustrated with python (or C++) examples you may play with. They are all based on *Gudhi* library. Please follow the compilation instructions available in this webpage: `http://gudhi.gforge.inria.fr/`. There are at least a few ways you can install Gudhi - you can go through the whole compilation (require cmake, Boost, Eigen3, Cgal,...) or use a anaconda package installation. If you have any problems please email me as well as Gudhi mailing list *gudhi-users@lists.gforge.inria.fr*

The archive with exercises accompanying this script can be downloaded from here `https://www.dropbox.com/s/n5mwv3bntslwpsw/code.zip?dl=0`

So far I have presented some version of this material at the one week long lecture series I gave at EPFL in February 2015 and a day tutorial in Topological Data Analysis in Paris region in June 2018. This document is designed as a supporting material for tutorial. You will find various comments clearly indicating it throughout the text. You will miss all the presentation part, but I strongly encourage you to go ahead and play with this tutorial. If you have problem that no one else can help, and if you can find me, then perhaps I will be able to help.

Enjoy!

Paweł Dłotko, Swansea University, July 2018.

Chapter 2

Why not topology?

Topology study global structure of spaces. It integrates local information and provide concise summaries of data. Non local, provable properties are very important in contemporary data analysis, since *local* methods are very popular. Analysing data locally reassembles a process of reading done by a child who only learn to read. In this process a child read letters bit-by-bit and trying to put-all-of-them-together into words so that something that make sense is obtained. At the other hand, a person who is more experienced in reading do not look at the single letters, but at more *global* linguistic structures. Especially if you are fluent with skim reading. What is the structure we are referring to? Please read quickly the text below proposed by Matt Davies from MRC Cognition and Brain Sciences Unit at Cambridge.

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a total mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

Despite some local errors, the message is easy to understand. But when you try to read it as a child, by combining single letters together, you will fail. The reason why you were able to read and understood the text is because the *shape* of words are preserved. Permutations of letters correspond to some, quite considerable, noise. Analysing shape of words is a base of skim reading. That being said, we get to our credo: shape matters! Even in presence of noise shape determines the meaning. Through the lecture we will give you more and more examples to support this and methods to analyse shape.

Related to the previous example Have you noticed that most of legal agreement and licences ARE WRITTEN WITH CAPITALS? WHY IS THAT? WELL, NOW THE WORDS DO NOT HAVE SHAPE, SO YOU CANNOT SKIM READ THEM. AND THIS IS WHAT THEY WANT: THEY TYPICALLY DO WANT TO DISCOURAGE YOU FROM READING MANY PAGES OF A TEXT LIKE THIS. Apparently even the legal people are aware that shape matters.

This is one of the simplest examples of a situation when global structure allows us to understand the data. But any local analysis will, most probably, fail. So, here we are, in the world of topology. But, what topology is? Let me give you a few examples.

Topology is about aggregating local information to global output. Think about a great number of cheap, gps-free sensors which are distributed in a vast area they are to monitor and detect a fire. The boundary of this area is fixed with a chain of sensors that can communicate with the neighbouring ones. Each sensor can communicate with other sensors lying not further than a radius r away. It also can detect fire if only it break not further away than $\frac{\sqrt{3}}{2}r$ from that sensor. In this setting, can the sensor network determine if the whole area or interest is observed? It may seems very unlikely, since sensors do not have localization information. They do not have metric information (apart from binary knowledge if a sensor that they can 'hear' is in the distance smaller than r, or not). But, the answer is: yes, it can, check [18] for details! And it is all thanks to topology and its ability to integrate local connectivity information to global data.

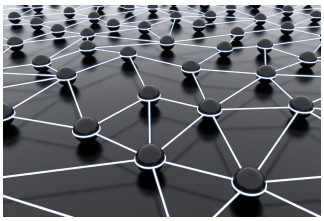


Figure 2.1: By Robert Ghrist.

Topology gives a rigorous dimension reduction technique. Think about a massive turbulent flow dynamics simulations the dynamics of which we want to characterize, see Figure 2.2. For the simulations the domain is divided into 12000^3 grid, 2048^3 sample of which is depicted in the right. An important factor is how *turbulent* the flow is. Typical measure of turbulence is the *enstophy*. In the picture, blue regions denote high, black low, enstrophy.

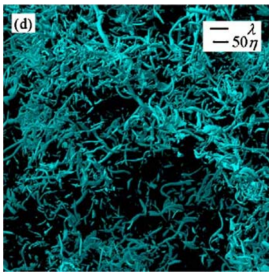


Figure 2.2: By Takashi Ishihara.

A typical characterization of the flow is an *average enstrophy*. That is a single scalar value to characterize this large and complicated dynamics. If the number is high (or locally high) then the simulation, or its region is consider highly turbulent. Low in the other case. But we can look at this picture through lenses of topology. Now we see cubical domain in $\mathbb{R}^3$ with a scalar value function defined on it. In this course we will discover tools to characterize it better than with a single scalar value.

Topology gives us an information about obstacles sometimes called *cuts*. Suppose we are given a non directed graph G consisting of set of vertices V and edges E (we will have more precise definition of a graph in one of the next sections). Let P be a set of all paths in G. We want to define a function $f : V \to \mathbb{R}$ (often called a potential) such that for every path $p \in P$ and for two constitutive vertices v_1, v_2 in this path, $f(v_1) > f(v2)$. It is easy to find out that such a function exist if and only if G do not contain *cycles*.

Quite often in engineering we face less trivial scenario when a similar situation happens. Let me focus here on one specific example dealing with solutions of Maxwell's equations. There are various ways of solving them, but there is one particularly interesting due to its speed and low memory consumption. It is called $T - \Omega$ approach to Maxwell's equations. It aims in defining a discrete counterparts of Maxwell's law given a three dimensional *mesh* $\mathcal{K}$ of a considered electric circuit. The mesh $\mathcal{K}$ decomposes into sub meshes $\mathcal{K}_a$ and $\mathcal{K}_c$, representing insulator and a conductor. One of the intermediate step require defining a potential–like function in the insulator $\mathcal{K}_a$. As in the case of graphs, this is not possible if $\mathcal{K}_a$ has some sort of cycles – the ones that do not bound anything. We will deal with those cycles a lot when talking about homology groups. In the case of $\mathcal{K}_a$ being non trivial, some topological correction have to be added to the system. In engineering we refer

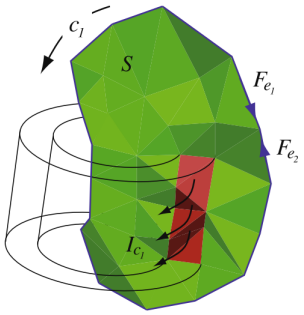


Figure 2.3: By Ruben Specogna.

to them as *cuts* and they are helpful to solve the problem better, faster and with less memory resources, see Figure 2.3.

Chapter 3

From graphs to complexes.

Graphs are object that, often independently, appeared in different disciplines of science. They are typically used as a abstraction of a set-up of some real-word problem. One of the first problems formulated in the language of graphs was a *Seven Bridges of Königsberg* problem formulated by Leonhard Euler in 1735. This problem, according to many scientists, give a foundation for graph theory and subsequently topology. Let us have a look at it.

Figure 3.1: From Wikipedia.

The problem is to find a walk through the city of Königsberg. A person have to cross each bridge exactly once and come back to the starting point. Euler's contribution was to show that such a walk is not possible. But, not the solution to this particular was most important, but a technique to reach it. Euler has abstracted away all the unnecessary information about the map of the city. He used only the important parts, which were:

1. The regions of the city that can be reached without crossing any bridge were contracted to single vertices - those gives rise to so called vertices in Euler's graph representing the problem.

2. The bridges, which connected the regions from the point (1). They give rise to the edges connecting vertices from point (1).

This reduction gave a very concise way of representing the problem. It is also a very good example of a process of *abstraction* which allow to give up all unnecessary details and construct a simplistic model that inherits all the properties required to solve the problem.

So why such a walk do not exist? I encourage you to experiment with this representation by yourself and see why. Really, this is just a simple counting. You can find the short description of the reason in the comments to this section.

Let us define graphs formally. An *abstract undirected graph* G is a pair (V, E) such that V is a set of vertices and E is a set of edges, each of which is a pair of vertices.

A $n-$*clique* in a graph is a set of vertices $v_1, \ldots, v_n$ such that there is an edge from v_i to v_j for every $i, j \in \{1, \ldots, n\}$, $i \neq j$.

A *path* between two vertices $u, v \in V$ is a sequence of vertices $u = v_1, \ldots, v_n = v$ such that there is an edge between v_i and v_{i+1} for every $i \in \{1, \ldots, n-1\}$.

Having this set of definition we can discuss the first topological concept of this lecture - a concept of *connected components*. A graph is connected if there exist a path between any pair of its vertices. If there exist two vertices for which such a path do not exist, then the graphs is *not connected*. We can define a relation of connectivity between two vertices: two vertices are in the relation if a path between them exist. So called *classes of abstraction* of that relation are connected component of a graph. A concept of a connected component is a classical in graph theory and in topology. But there are more common concept in the intersection between that two. Let us talk about *cycles*. A path having the same begin and end point $u = v_1, \ldots, v_n = u$ is called a *cycle*. For simplicity we will add an additional assumption that there are no repetitions among vertices $v_1, \ldots, v_n$ in a cycle. Simply speaking, a cycle is a closed path in a graph. Not all the graphs admits cycles. A graph that is connected and do not have a cycle is a *tree* (if it has unique connected component), or a forest in a more general case.

Given a connected graph G, we can construct a maximal acyclic sub graph of G. Any such a subgraph is often refereed to as a *spanning tree*. Note that for a given graph, there are many possible spanning trees. For a given graph G, let T be such a spanning tree, see an example in the Figure 3.2. It is clear that we cannot

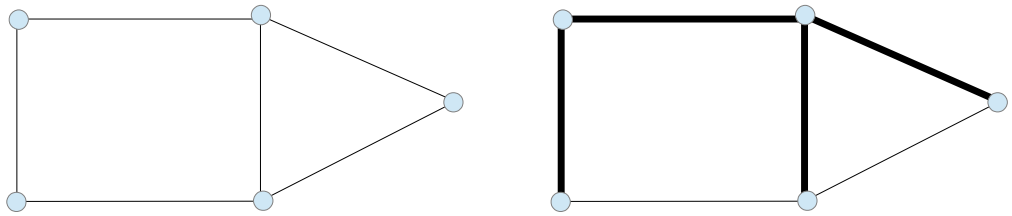


Figure 3.2: On the left, the initial graph G, on the right, with bold, the edges that belong to a spanning tree T of G.

add any more edges to T without introducing a cycle therein. Let me call every edge that is not a tree edge a *cotree* edge. Every cotree edge gives rise to a cycle. The collection of all the cycles obtained from all cotree edges forms a *cycle basis of a graph*. A cycle is yet another concept in common between graph theory and topology. One can interpret cycles as one dimensional holes in the structure of the graph.

Basic topological description in dimension zero is the one about connected components. Think of them as holes in the space. Various clustering methods can be found to capture connected components. The question you may be asking right now is: do we really need more? It turns out that in some cases we do. Have a look at the Figure 3.3. All the sets over there are different connected components, but they are very different. Topology allows to discriminate between them.

Given a cycle, we can *grasp* such a graph using that cycle to cage it. This cycle is like a 1-dimensional hole.

This rise a question: are there two, or higher dimensional holes? The answer is, yes, there are. Not in graphs, but in more general objects called complexes that are generalization of graphs. We will study those objects in the next sections. To talk about them, we need to leave safe harbour of graph theory into uncharted sea of algebraic topology. But, we shall always remember where we start from: we should remember that complexes and homology theory which we will discuss, is a generalization of a well know concept in graph theory and we can get a lot (but not all!) motivation from graph theory. Coming back to this basic intuition turns out to be very useful.

Before we start our voyage to the second star from the right, let us have a look at yet another nice intersection of classical graph theory and topology. Let us talk about *flows and cuts* and max flow min cut *duality*. To introduce this concept, we need *directed graphs*. They are almost as standard graphs. The only difference is that the set of edges $E \subset V \times V$ is formed by ordered pairs i.e. the edge (v_1, v_2) and the edge

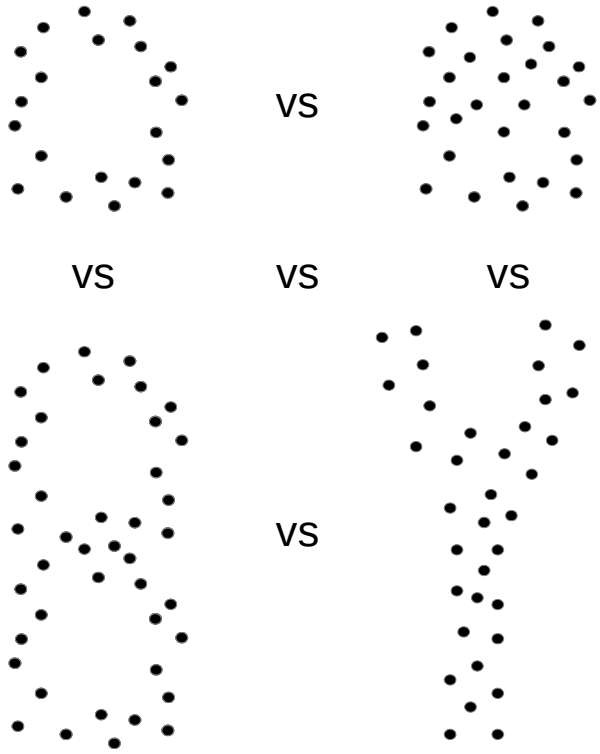


Figure 3.3: The same connected components different meaning. One of toy examples showing why we may need higher order topological information.

(v_2, v_1) are two different edges (pointing to the opposite directions sort of speak).

Often for various problems we are given some functions defined on vertices and edges of the considered graph. For the flow problem let us consider a *capacity* function on the graph's edges $c : E \to \mathbb{R}$. The capacity of an edge e tells us how much stuff can be pushed through e in a unit of time.

A *flow* is a map $f : E \to \mathbb{R}$ subject to the following two constraints:

1. Capacity constraint, for every $(u, v) \in E$, $f(u, v) \leq c(u, v)$. Simply speaking, we cannot push through an edge more stuff than its capacity.

2. Conservation of a flow: for every vertex $v \in V$ the amount of flow that enters v, exit from v. This is one of Kirchoff's law from the circuit theory.

Let us now pick two vertices $s, t \in V$. The s stands for *source*, t stands for *target*. A typical question in graph theory is: how much flow can we send from a source to target? An example of a weighed graph with a capacity function is presented in the Figure 3.4. This flow problem can be easily changed to a circulation problem by adding a directed feedback edge, let's call it F, from target to source of an infinite capacity. Now, I would like to consider only the cycles in a graph that involve the edge F. One can think that all the *local* cycles in the graph are trivial/non important.

A celebrated result in computer science says, that maximal flow (or circulation) is equal to the minimal cut. Flow lines are cycles crossing F. *Cuts* are the edges whose removal kills all flow. Mathematicians may already

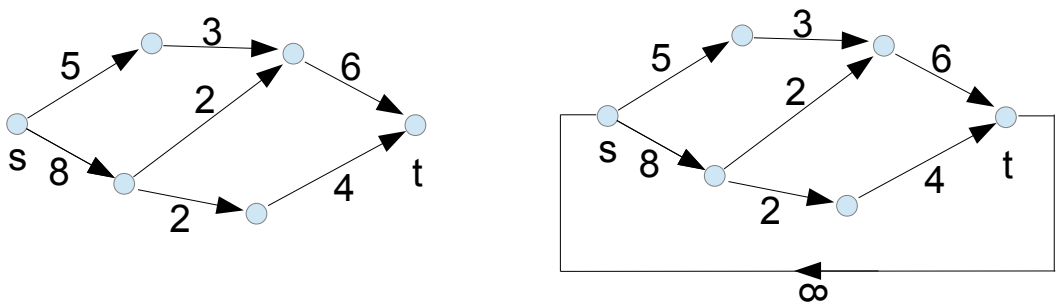


Figure 3.4: On the left, the initial directed graph with source and target and capacity function. On the right, the same graph with an extra feedback edge of an infinite capacity.

see an analogy to a well known theorem in topology. Flows are like first homology generators. Cuts are the dual generators of the first cohomology group. And this analogy is true. Max flow min cut theorem is nothing more than a special version of well known Poincare duality. The cuts we have been talking about are just like the cuts in the graphs. The only difference is that they are embedded in a higher dimensional structures we will be discussing in the next section.

3.1 Comments

1. When talking about Euler's problem I have promised you an explanation why a described walk in Königsberg do not exist. Have a look at the graph. There are odd numbed of edges incidental to every vertex. Suppose by contrary that such a walk exist. If it exist, we can assume that we start at any region we like, let me call it A. Then in order to move to the next region, we need to cross a bridge. Pick any bridge and cross. Once the bridge was crossed, it cannot be crossed any more from the formulation of the problem. Now, there are even number of bridges we can use when going through A. When getting in and getting out, we will make two of the remaining even number not usable. Therefore we can never get back to A having used all the attached bridges. In his work Euler gave a deep analysis and proved theorems about existence of a path, which we nowadays call an *Euler path* in graphs.

2. A Moore's law is an observation saying that the processing power of our computers double every 18 months. That means that now for 1000$ we can buy twice as powerful computer as we could 18 months ago. There are huge amount of money put by the processors manufactures to maintain such a speed up. But, far fewer people are aware that there is a Moore's law present also in algorithms and in general in science. In particular, the development of algorithms to compute maximal flows and minimal cuts extends Moore's law even though the money invested to developing algorithms are small fractions of money invested to build chips. So, if you have a DeLorean which allows you to travel back in time and you want to make a lot of money by doing max flow min cut calculations 20 years ago, but can take either software or hardware, you should definitely take software with you.

3. The genesis on work on max flow and min cut is a cold war analysis of transport capability of Soviet Union and the dependent countries[1]. The very urgent question for the US Air Forces was – how much supply can be bring daily to the front lines and where to cut the rail network of the enemy with the minimal cost

[1]See USAF unclassified report available here: `http://www.dtic.mil/dtic/tr/fulltext/u2/093458.pdf`.

so that a supply for the front lines will be broken. Below, a original map of the railway network graph with weights on the edges:

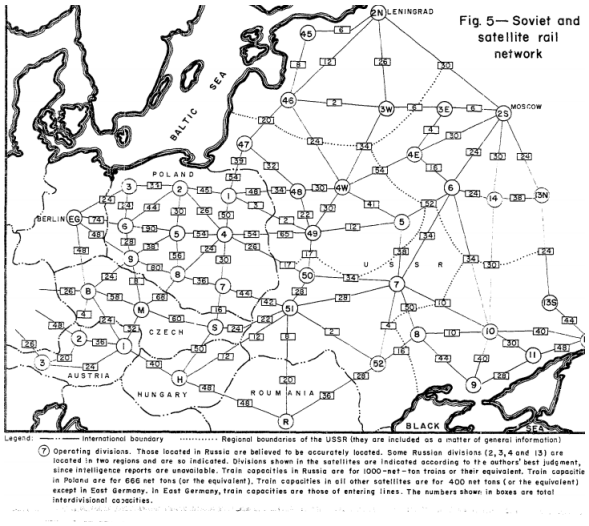


Figure 3.5: USAF Project Rand, research memorandum.

4. A classical algorithm to compute maximal flow and minimal cut is due to Ford and Fulkerson. The idea is very simple:

   (a) Try to find a path p from source to target such that flow do not use all available capacity of that path.

   (b) Increase the flow through p so that maximal capacity of p is reached.

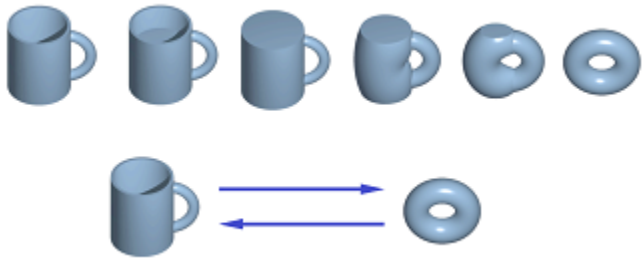
   (c) Repeat previous steps as long as such a path p can be found.

   In order to find a minimal cut, one need to start from the source and annotate all the vertices that can be reached with the edges e such that $f(e) < c(e)$. Minimal cut consist of all the edges which has one endpoint annotated and the other one not annotated.

Chapter 4

Why topology?

Topology is about *integrating local properties into global information*. Think of a set of beacons which can be heard from some distance. Imagine, that we have only a binary information: can I, or can I not hear the beacon *i*? Note that such a very local information typically will not change under a *continuous* deformation. Quite extreme example of such a continuous deformation is an equivalence of coffee mug and a donut, one you can see in the picture below.



In contract to that metric is about precise distances. In metric spaces we can determine exactly how far we are from a given object. How far we are from the Eiffel tower? How far it is from here to the nearest coffee shop? Metric give much more precise information, but at much greater costs. If we forget about metrics, we can have invariants that are very general - to the extend where they are the same for any pairs of shapes that can be continuously deformed to each other, like donut and coffee mug above.

Given this exposition you may wander, what is topology good for? Is there any realistic scenario when it can be used. The answer is positive, and we will have an example of this situation in this section.

Think of a collection of beacons. You do not know where they are. You do not know how far away from you they are. What you can measure is if you can hear it or not (and in some cases the strength of the signal). This is precisely the notion of proximity available via topology. There are some examples of beacons that uses Bluetooth technology - you can find them in some shops, and they allow to find various goods you are looking for (and perhaps even want to buy).

In various campuses and urbanized areas there are different types of beacons that we can use... If you happen to have osx or linux machine you will be able to take part in the exercise. Unfortunately at the moment I do not have a code that works on windows machines (sorry...) If you are linux user please download this code:
`https://www.dropbox.com/s/qpxocdk57x9cmaf/ssidcollect_linux.py?dl=0`

If you are osx user, please download this code:
`https://www.dropbox.com/s/bkf2yj6u1eza82w/ssidcollect_osx.py?dl=0`

Both are designed to recode ssid of wifi networks. Run them by using:

```
python ssidcollect_linux.py
```

And go for a walk. You may go around some obstacles (ideally of a considerable diameter). When you are back, stop the execution (you can do it by hitting CTRL+c or Command+C).

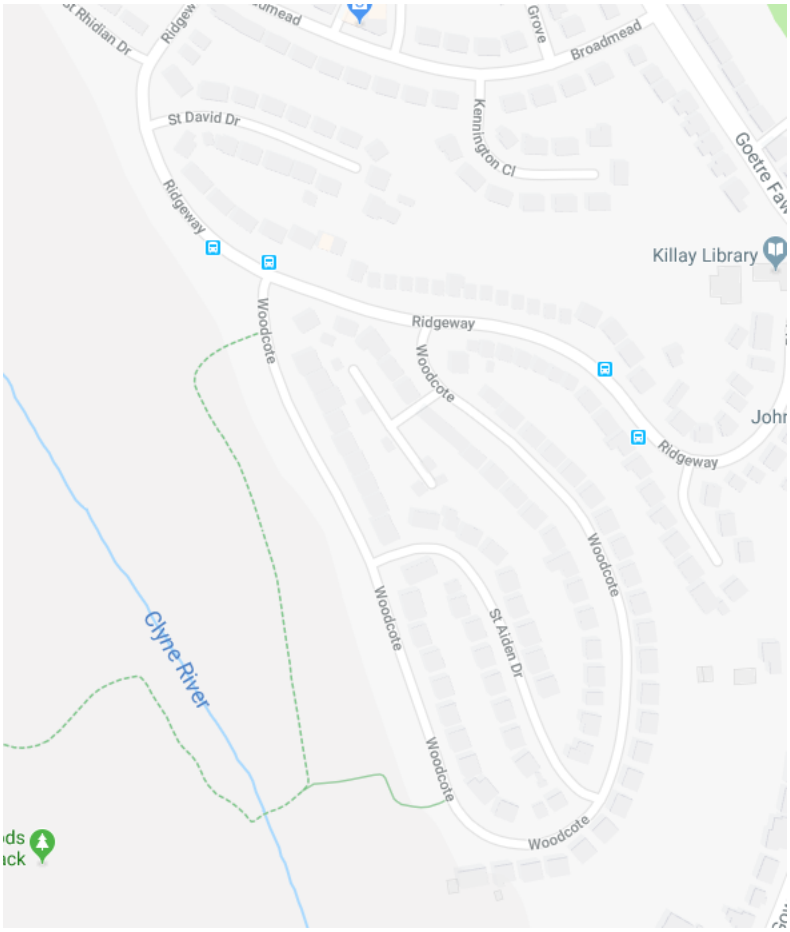
Then in the folder you have run the code the file *output* is produced. Have a look at it! The first column is a time (number of seconds since 1 January 1970), second, the ssid of the router, and third: the signal strength. We will turn this data now to a collection of points in $\mathbb{R}^N$. Number of dimensions, N, is equal to the number of distinct ssids we have in the file. Each sample in the time will correspond to a point. You can either implement this simple code, or get it from here: `https://www.dropbox.com/s/2euv2zahn2dyirs/ssidaccumulate.py?dl=0` To get the point cloud simply run:

```
python ssidaccumulate.py
```

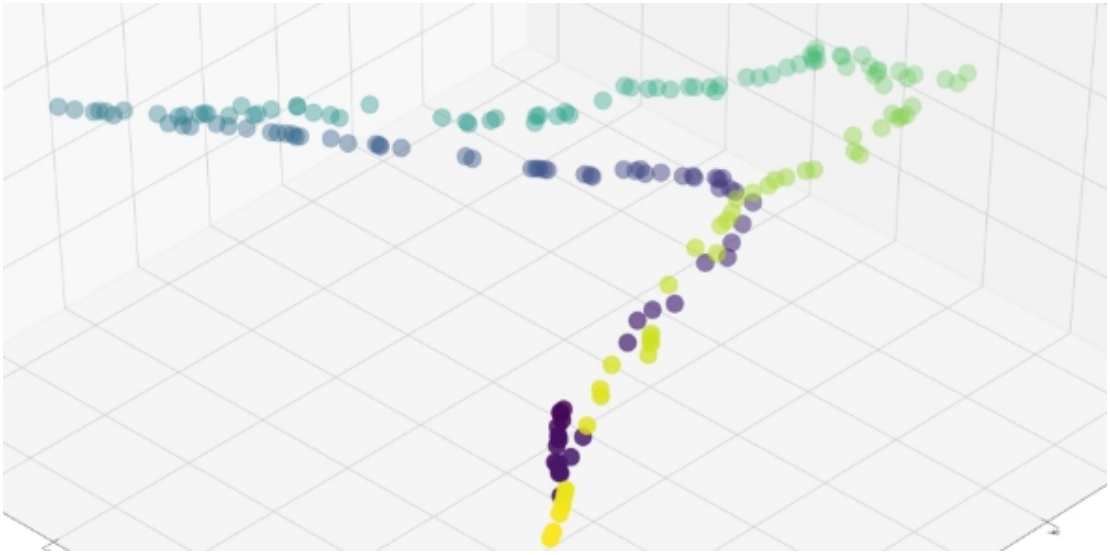
By doing so we will obtain a file *file_with_points.csv* Let us run the Principal Component Analysis on our data:

```
import matplotlib.pyplot as plt
import numpy
import csv
from sklearn import decomposition

X = numpy.genfromtxt('file_with_points.csv', delimiter=',')
fig = plt.figure()

ax = plt.axes( projection='3d' )

pca = decomposition.PCA()
pca.fit(X)
X = pca.transform(X)

size = [200 for n in range(len(X[:,0]))]
color = [100*n for n in range(len(X[:,0]))]
ax.scatter(X[:, 0], X[:, 1], X[:, 2],s=size,c=color)#
plt.show()
```

I have done this experiment when waling close my house in Swansea. Without revealing too much of my private data, I have done one of the circles from the map below:



Which results in the following result:



As you see, no metric features are preserved here, but we can notice that I have made a loop. Colouring indicate time. All this is based by detecting proximity to wifi routers. Later on in this lecture we will learn the ways of quantifying that there indeed is a cycle.

A historical note. I have done this exercise first attending a summer school in Topological Data Analysis organized by Gunnar Carlson, Rob Ghrist and Ben Mann in Snowbird in 2011. Majority of the code we have

used here is due to Mikael Vejdemo-Johansson (also known as MVJ). Big thanks to him!

Chapter 5

Cubical complexes

In the previous section we have discussed an idea of simplicial complexes. They are very important in computational topology. But there are other ways of representing topological spaces which are also highly useful. One of them is an idea of cubical complexes. As far as I know cubical complexes were introduced and used by Jean-Paul Serre [17].

I came across cubical complexes when studding rigorous dynamics and computed assisted proofs. They are closely related with *interval arithmetic* – a computational tool that allows to make calculations on computer rigorous [14]. The idea of the interval arithmetic is to put all errors of the computations into representation of a number. To do so, interval arithmetic define all the elementary operations (together with elementary functions) on a intervals instead of single numbers. ore precisely, by $\mathbb{D}$ let us denote a set of *double* numbers represented in a computer. Let us take $x, y \in \mathbb{D}$. By $low(x \diamond y)$ let us denote a largest number in $\mathbb{D}$ that is smaller than the result of operation $x \diamond y$. By $high(x \diamond y)$ let us denote a smaller number in $\mathbb{D}$ that is greater than the result of operation $x \diamond y$. We can obtain those numbers, since a processor get the result with higher precision than the one available in double numbers. Those operations guarantees that the true value of operation can be always enclosed in the resulting interval. Let us take two intervals $[a, b]$ and $[c, d]$ such that a, b, c, d are *representable* doubles. Then:

1. $[a, b] + [c, d] \subset [low(a + c), high(b, d)]$.
2. $[a, b] - [c, d] \subset [low(a - d), high(b - c)]$.
3. $[a, b] * [c, d] \subset [min\{ac, ad, bc, bd\}, max\{ac, ad, bc, bd\}]$.
4. I am skipping the formula for a division. It is analogous to the previous ones. One have to make sure that zero is not in the interval in the denominator.

Note that in the computations above the operations on the left hand side are rigorous mathematical operations defined on the intervals. The operations on the right hand side are made on computer representation of double numbers.

By using a basic Taylor expansion, we can also have an interval version of elementary functions. Given this, we can build mathematically rigorous numerical algorithms. This give rise to a big field of computational mathematics called rigorous dynamics.

In the dynamical systems, we often deal with a maps acting from $\mathbb{R}^n$ to $\mathbb{R}^n$. To do a rigorous computations in this $n-$dimensional setting, we are using $n-$dimensional cubes. This is the rigorous dynamics genesis of *cubical complexes*.

For now, we restrict only to cubical complexes for which cubes of a fixed dimension have the same size. By doing a rescaling we can assume that the endpoints of the involved intervals have integer coordinates. By an *elementary interval* we mean an interval $[n, n + 1]$ or an interval $[n, n]$. The first one is non degenerated, the second one is a degenerated one. A *elementary cube* is a Cartesian product of elementary intervals (degenerated of not). A few examples of elementary cubes are presented in the Figure 5.1.

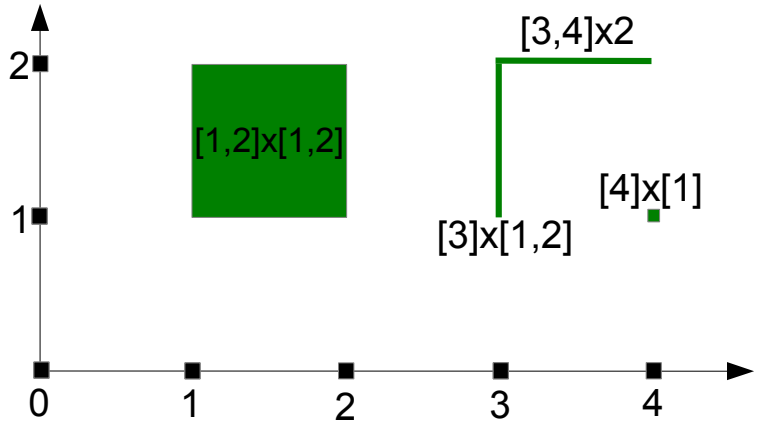


Figure 5.1: A cubical complex.

Elementary cubes are building blocks of cubical complexes. Let us take a n-dimensional cube $C = [x_0, y_0] \times [x_1, y_1] \times \ldots \times [x_n, y_n]$. At the beginning let us assume that all the intervals involved in C are not degenerated. Then the boundary of C is a collection of cubes $[x_0, y_0] \times [x_1, y_1] \times \ldots \times [x_i, x_i] \times \ldots \times [x_n, y_n]$ and $[x_0, y_0] \times [x_1, y_1] \times \ldots \times [y_i, y_i] \times \ldots \times [x_n, y_n]$ for every $i \in \{0, \ldots, n\}$. If some intervals in the product are degenerated, they are skipped in this procedure. Note that every element in the boundary of C is a cube of a dimension equal to the dimension of C minus 1. See Figure 5.2 for an example.

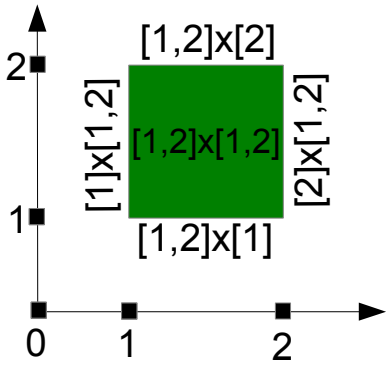


Figure 5.2: Boundary of an elementary cube.

For a typical kind of input there is one technical advantage of cubical complexes with respect to simplicial ones. They can be very efficiently stored in the computer memory. In the best case, we only need one bit (bit, not byte!) of information per cube. We do it by using so called *bitmap* representation. Instead of defining it formally, let us have a look at example. In the first case we will store only top dimensional cubes in the bitmap. In the second one we will show how to store the cubes of all dimensions.

In the Figure 5.3 an idea of a bitmap used to store cubes of fixed dimension is presented. In this complex, the cubes $[0, 1] \times [0, 1]$, $[2, 3] \times [0, 1]$, $[3, 4] \times [0, 1]$, $[1, 2] \times [1, 2]$, $[3, 4] \times [1, 2]$ are present. All cubes have unit length, and the range of out complex is from 0 to 4 in the x direction and from 0 to 2 in the y direction. We

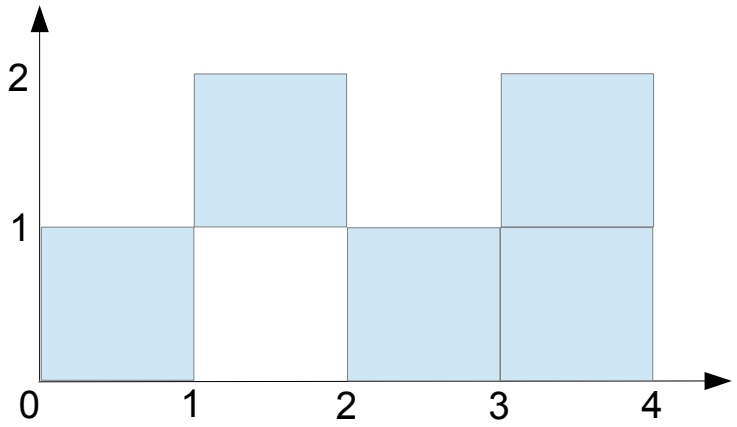


Figure 5.3: Bitmap of a fixed dimension.

can therefore enumerate all the possible cubes in that range and get the following 2 dimensional array:

$[0,1] \times [1,2]$	$[1,2] \times [1,2]$	$[2,3] \times [1,2]$	$[3,4] \times [1,2]$
$[0,1] \times [0,1]$	$[1,2] \times [0,1]$	$[2,3] \times [0,1]$	$[3,4] \times [0,1]$

To encode the complex, it suffice to encode which of the above cubes are present and which are not. Therefore, here is a representation of the complex:

$$\begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{matrix}$$

This array, given the wight and the height of a bitmap, can be encoded of one dimensional array of bits[1].

The idea of encoding the cells of all dimensions is very similar. Please consult the Figure 5.4. The obvious details are left for the reader.

Let me discuss one of the application of computational topology in rigorous dynamics. In this case, computational topology is used to compute a Conley index. The former one, if nontrivial, indicate existence of invariant part of a dynamical system (fix point, periodic orbit, etc.). An informal idea is presented in the Figure 5.5.

Let us make a simple exercise which will lead us to the topic of the next section - the filtrations. Let us define a two dimensional array of a size $2N \times 2N$. Let us assume that this grid cover the area $[-2,2]^2$. On each grid element let us define a value of a function which is a distance to the unit circle. Note that for this very specific function, the distance to the unit circle is simply a norm of the grid element. Let us display it and create a cubical complex based on it.

```
import numpy as np
import math
import gudhi as gd
from PIL import Image


N = 100
array = np.zeros((2*N+1,2*N+1))
xExtrem = 2;
```

[1]This is why we call it a bitmap.

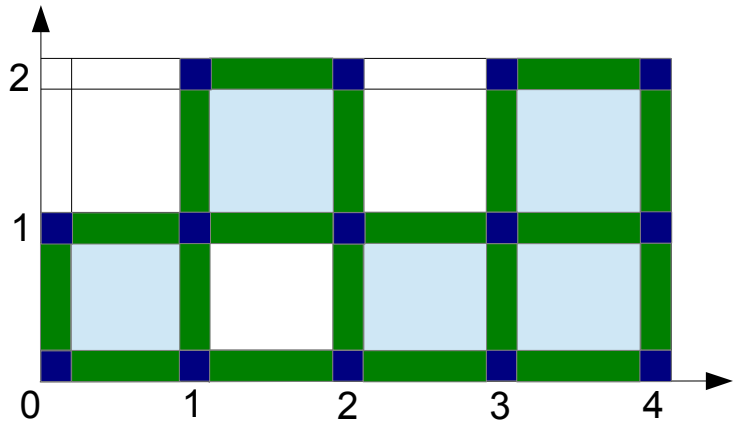


Figure 5.4: Bitmap containing cubes of all dimensions.

```
yExtrem = 2;

bitmap = []
for i in range(0,2*N+1):
        for j in range (0,2*N+1):
                x = i/(2*float(N)+1)*2*float(xExtrem)-xExtrem
                y = j/(2*float(N)+1)*2*float(xExtrem)-xExtrem
                norm = math.sqrt( x*x + y*y )
                norm =  math.fabs(norm-1)
                array[i][j] = norm
                bitmap.append(norm)


#Here we will display our creation:
plt.imshow(array, cmap='gray', vmin=np.amin(array),vmax=np.amax(array))
#plt.savefig('circle.png')
plt.show()

#Given the input data we can buld a Gudhi btmap cubical complex:
bcc = gd.CubicalComplex(top_dimensional_cells = bitmap, dimensions=[2*N+1,2*N+1])
bcc.persistence()
```

Here is a sample example of the picture you may get:

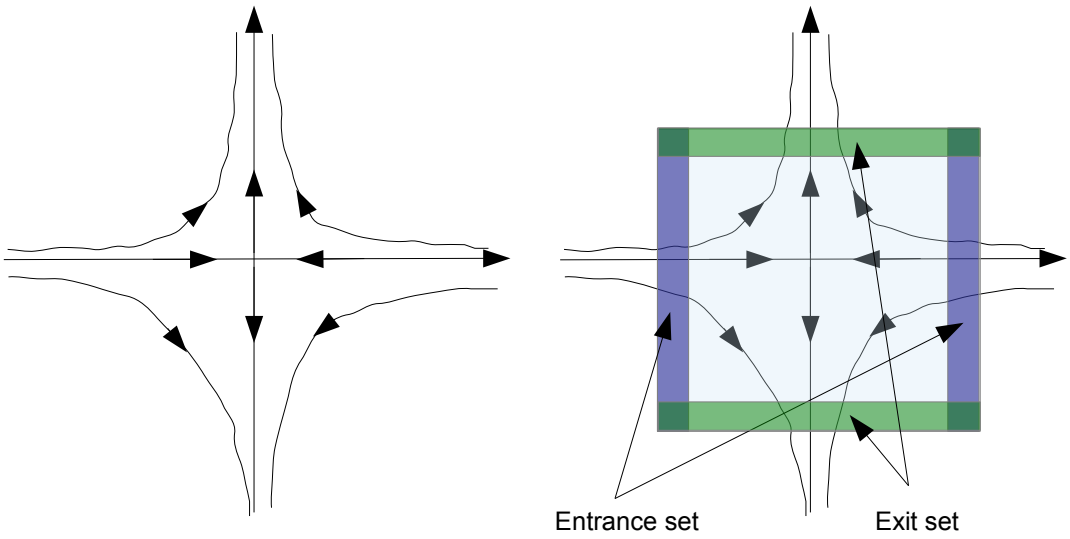
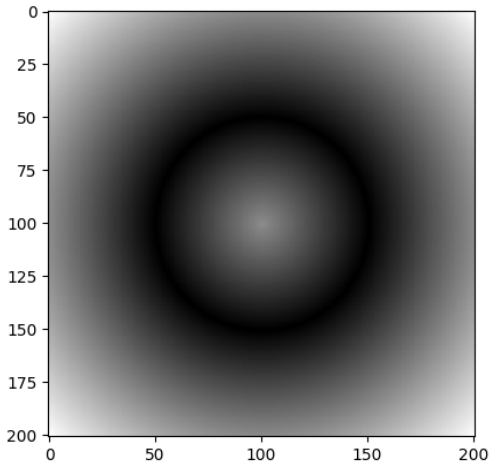


Figure 5.5: Idea of a Conley index, for further details please consult [12]. Let us have a simple phase space presented on the left. There is contraction in a horizontal direction and expansion in vertical one. Let us suppose we build so called *insulating neighbourhood* B as the box in the picture on the right. We assume that there are no fixed points in the boundary of B. We decompose the boundary of B into two sets: *exit set* is a set of all points that get out of B for any sufficiently short time (marked with green) and *entry set* which consist of all points in the boundary of B that enters B for any sufficiently small positive time (marked with blue). Now suppose we construct so called *quotient space*, where we contract the exit set into a point. To imagine the construction let us imagine first that the two horizontal stripes are contracted to a point. Later those two points have to be brought together. By dong so we will get a set having a homology of a circle. Celebrated theorem by Conley says that if there is an invariant set inside B, then topology of such a quotient space is not trivial. This is indeed the case over here.



Let us have a look at a similar example with a bit different way of obtaining a filtration on the maximal cubes (using Kernel Density Estimator).

Crater dataset (for cubical filtration) This example was created and made available by Bertrand Michel. Warm thanks to Bertrand! Please visit his webpage for more interesting tutorials `http://bertrand.michel.perso.math.cnrs.fr/Enseignements.html` The idea we want to illustrate over here is the following one. We start from a collection of points, and based on them, we create a continuous function: a kernel density estimator build based on those ponts. Having the continuous function, we discretise it on a grid, and computer persistent homology of that discretization. For that purpose, we will be using cubical complexes. Currently we use python native mechanisms to get this filtration. Gudhi–dedicated tools for that will be soon available. Please download the data from that location `http://bertrand.michel.perso.math.cnrs.fr/Enseignements/TDA/crater_tuto`

```
import numpy as np
import pandas as pd
import pickle as pickle
import gudhi as gd
from pylab import *
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
from IPython.display import Image
from sklearn.neighbors.kde import KernelDensity

f = open("crater_tuto")
#For python 3
#crater = pickle.load(f,encoding='latin1')
#For python 2
crater = pickle.load(f)
f.close()

plt.scatter(crater[:,0],crater[:,1],s=0.1)
plt.show()

#create 10 by 10 cubical complex:
xval = np.arange(0,10,0.05)
yval = np.arange(0,10,0.05)
nx = len(xval)
ny = len(yval)


#Now we compute the values of the kernel density estimator on the
#center of each point of our grid.
#The values will be stored in the array scores.
kde = KernelDensity(kernel='gaussian', bandwidth=0.3).fit(crater)
positions = np.array([[u,v] for u in xval for v in yval ])
scores = -np.exp(kde.score_samples(X= positions))

#And subsequently construct a cubical complex based on the scores.
cc_density_crater= gd.CubicalComplex(dimensions=
[nx ,ny],top_dimensional_cells = scores)
# OPTIONAL, persistent homology computations:
pers_density_crater =cc_density_crater.persistence()
plt = gd.plot_persistence_diagram(pers_density_crater).show()
```

5.1 Filtration on complexes

In this section we will define a filtration of a simplicial or cubical complexes. Traditionally in mathematics filtration is defined as a increasing sequence of subcomplexes. To make it as simple as possible, I prefer to define it in an equivalent way, by using so called *filtering function* defined on a finite complex. Let $\mathcal{K}$ be a finite simplicial or cubical complex. $f : \mathcal{K} \to A$, where A is a set with a total order, is a filtering function if for every $a \in \mathcal{K}$ and for every b in the boundary of a, $f(a) \geq f(b)$. In computational topology we often say that the boundary of a cell have to enter the filtration before the cell. In the Figure 5.6 please find a positive and negative example of a filtering function.

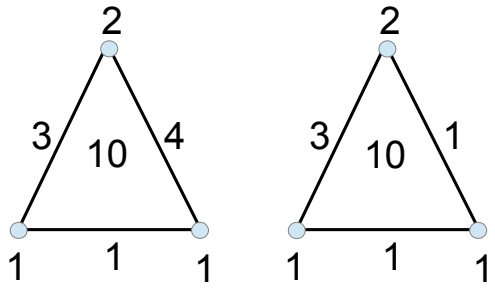


Figure 5.6: On the left, a filtering function on a simplicial complex. The function on the right is not a filtering functions, as some simplices appears before their boundaries.

There are a few standard ways of defining filtration in a complex $\mathcal{K}$. Let us have a look at them:

1. Given a filtering function f defined on vertices. This may happens for instance when a scalar function is provided along with geometric data. Then, given a simplex or a cube S which have vertices $v_0, \ldots, v_n$ we set the filtration value of S to $max_{i \in \{1,\ldots,n\}} f(v_i)$.

2. Given a filtering function f defined on top dimensional cells of a complex $\mathcal{K}$. This situation happens for instance when a $2d$ or a $3d$ image is given. The gray scale level of the R, G or B from RGB levels on pixels or voxels provide such a function. Then, the filtering function on a cell $S \in \mathcal{K}$ is defined as minimum of a filtering function values of maximal cells that have S in their boundary.

3. Suppose that we are constructing a Rips complex. Then a natural filtration is induced by the length of the edges of a graph. Then a filtration value of vertices is set to 0, filtration values of edges is their length and the filtration values of higher dimensional simplices is maximal value of the length of edges in the boundary of that simplex.

It is an easy exercise that those are filtering functions on a complex $\mathcal{K}$.

Chapter 6

Cycles and homology.

Algebra is an offer made by the devil to mathematician. The devil says "I will give you this powerful machine, it will answer any question you like. All you need to do is to give me your soul: give up geometry and you will have this marvellous machine"
Sir Michael Atiyah, 2002

In order to start talking about applied *algebraic* topology, we do need to dive a little bit deeper into algebra. Firstly we will define homology groups, which are almost exactly fit the definition of the devil's offer mentioned above. However later on we will explore the persistent homology, and by doing so we will re-gain our soul, as we will regain some part of geometric information.

The core idea of algebraic topology is to assign numbers to a geometrical elements. Taking a collection of $n-$dimensional simplices, by assigning numbers to them we obtain $n-$dimensional chains. They form a structure of a group[1] by a simple addition of elements component-wise (like polynomials). In this lecture we will restrict ourselves to the case of $\mathbb{Z}_2$ group, i.e. the elements of the group are 0 and 1, and the operations are defined *modulo 2*. In this case it is important to remember that:

$$0 + 0 = 0$$

$$1 + 0 = 0 + 1 = 1$$

$$1 + 1 = 0$$

A *group of n-dimensional chains* of a complex $\mathcal{K}$ will be denoted by $C_n(\mathcal{K})$. This group, for $n < 0$ or n greater than the dimension of the complex $\mathcal{K}$, is a trivial group, as there are no simplices to support the chains.

Right now, we have a graded structure of groups in different dimensions. To relate them, we need to define a *boundary operator*. Let us define it formally. In the lecture about complexes we have already encountered boundaries of simplices. Now we have a formal framework of chains, so we can define a boundary of a simplex as the following chain: Let us start from the case of simplices. Given a simplex $S = [v_0, v_1, \ldots, v_n]$, its boundary, $\partial S = \sum_{i=0}^{n} [v_0, v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n]$. Now, given a chain $c = \sum \alpha_i s_i$, where α_i's are scalars and c_i's are simplices, we can define a boundary of c as $\partial c = \partial \sum \alpha_i s_i = \sum \alpha_i \partial c_i$. In this way, the boundary operator extended to chains maps $C_n(\mathcal{K})$ onto $C_{n-1}(\mathcal{K})$. A *chain complex* is a sequence of chain groups joined by the boundary operator. It is a crucial property of the boundary operator that $\partial\partial = 0$. It suffice to prove this property for every simplex – we leave it as a simple exercise.

In order to define a homology groups, we need two subgroups of a group of chains – group of cycles and group of boundaries:

1. Cycles in homology theory are generalization of cycles we know from graph theory. A cycle in a graph is a closed path consisting of edges[2]. We can "pick up" the edges belonging to the cycle by defining a chain

[1] A group in algebra means a structure with associative operation that poses unit element and such that every element has its inverse.

[2] Let us assume for simplicity that no edge is repeated in the cycle.

c having values 1 for the edges in the cycle, and 0 otherwise. The chain c has a special property: every vertex of the cycle is incidental to even number of edges in the cycle. Consequently $\partial c = 0$, so a cycle, treated as a chain c, has empty boundary. That is a definition that generalize for higher dimensional complexes. A $n-$chain c is a cycle if $\partial c = 0$. From graph theory we have an intuition that a $1-$cycle is a closed path. What about a $2-$cycle? Well, think about a closed surface. A boundary of a tetrahedron of a cube for a start. In that case, every edge in the tetrahedron is incident to two 2-dimensional cells, and therefore it vanishes in the boundary.

The intuition of a 2-cycle is a closed surface. Note that this surface do not have to be a sphere. Think for instance of a boundary of a torus we have explored so far. This is again a 2-chain. With the sufficient stretch of our imagination this analogy can be carried on higher up with the dimension. Cycles form a subgroup of a group of chains. The group of $n-$dimensional cycles of a complex $\mathcal{K}$ is denoted by $Z_n(\mathcal{K})$.

It is good to think about cycles as a way to encapsulate information. In homology theory we are particularly interested in information about holes. given a complex with a hole, like the one in the Figure 6.1 we can pick a cycle to be the bound the hole. Many other cycles will encapsulate the same hole. There are also cycles which do not encapsulate anything but a solid region of the complex. In order to distinguish them, we introduce a concept of a *boundary*.

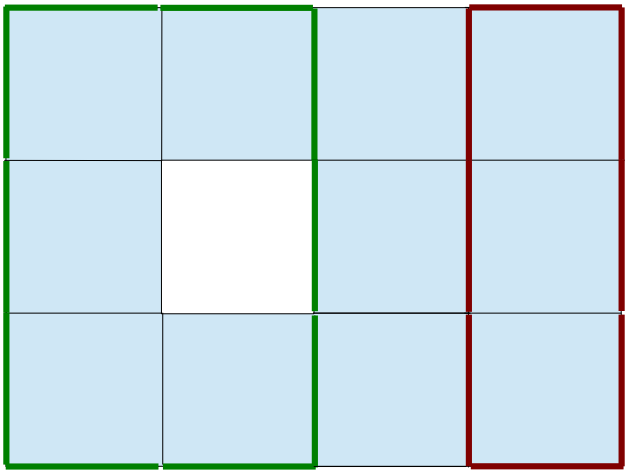


Figure 6.1: Examples of cycles that bounds and that do not bound a hole.

2. *Boundaries* are special kind of cycles that do not bound a hole. Let us have a look at the Figure 6.1 at the red cycle (the one on the right). This cycle is a boundary of a 2-chain consisting of all cubes surrounded by the cycle. Why? Well, the edges in between cubes cancel out, so only the external ones remain. Formally a *boundary* is a $n-$chain c such that there exist a $(n+1)-$chain d such that $\partial d = c$. Boundaries forms a subgroup of cycles. The group of $n-$dimensional boundaries of a complex $\mathcal{K}$ is denoted by $B_n(\mathcal{K})$.

One of the fundamental properties of the cycles and boundaries is the fact that every boundary is a cycle. There is a standard proof for simplices and cubes which can be figured out with paper and pencil in a short time. We will skip this proof here.

It is typical in algebraic topology to consider the cycles which bounds a hole important, and those which do not, not important. Moreover, Two cycles c_1 and c_2 such that $c_1 + c_2$ is a boundary are considered equivalent. See the Figure 6.2 for an illustration.

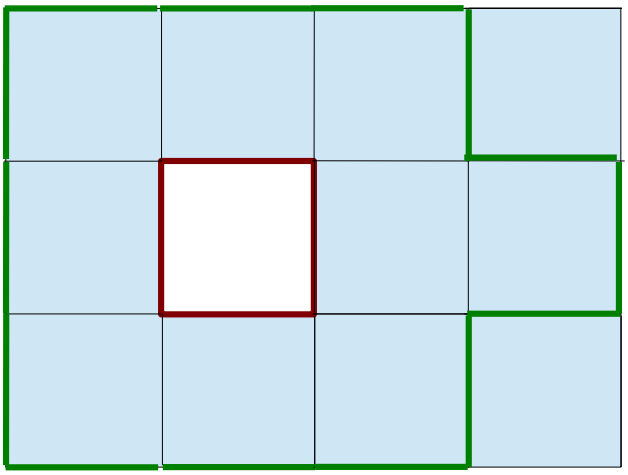


Figure 6.2: Two equivalent (homologous) cycles.

That intuition leads us to the definition of homology. Let us make it formal. Firstly we consider all the cycles that are boundaries trivial. Secondly, all the cycles that bound the same hole are considered equivalent. Since $\partial\partial = 0$, every boundary is a cycle, and the group of boundaries is a subgroup of the group of cycles. Given this, when quotient the group of cycles by the subgroup the boundaries, we will get the information about the holes in the complex, one cycle per hole. Those are given by so called homology groups. More formally:

Definition 1 *The p-th homology group is the p-th cycle group modulo the p-th boundary group.* $H_p(\mathcal{K}) = Z_p(\mathcal{K})/B_p(\mathcal{K})$

Let me stress the main idea which is to count only the cycles that surrounds a hole. Therefore, one can interpret a homology group as a tool to count holes in the space. Typically an integer homology is considered, which require bringing matrix into so called *Smith Normal Form*. In our simplified scenario we compute homology over $\mathbb{Z}_2$, what requires a simplified matrix reduction algorithm. We will present it in a version suitable for computations of persistent homology and re-use it in the next section.

In order to work with this algorithm we need to introduce a filtration (a.k.a. an ordering of elements) in the complex. For the sake of homology computations any filtration/ordering that preserves the odder of dimension works. Let us consider an example presented in the Figure 6.3. It consist of a complex being a two dimensional cube. The filtration we have chosen is the following:

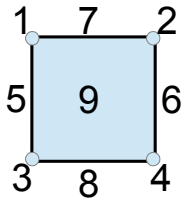
$$[1], [2], [3], [4], [1, 3], [2, 4], [1, 2], [3, 4], [1, 2, 3, 4]$$

Now, given the *sorted boundary matrix* we want to perform the following algorithm:

Algorithm 1 Matrix reduction.

Require: Sorted binary matrix M of a size $m \times m$;
  for i = 1 to m **do**
    while there exist $j < i$ such that low(i) = low(j) **do**
      Add column j to column i

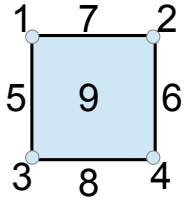
Where $low(i)$ is an integer being the lowest nonzero index in the $i-$th column. The idea of this algorithm is the following: We consider the columns of the matrix from left to right. And, if only the lowest one of the



	1	2	3	4	13	24	12	34	1234
1					1		1		
2						1	1		
3					1			1	
4						1		1	
13									1
24									1
12									1
34									1

Figure 6.3: Example of a complex with a filtering function on it that will be used in $\mathbb{Z}_2$ homology computations. Empty spaces denotes zero entries.

considered column is equal to the lowest one of one of the previous columns, we add the previous column to the considered one. We check the previous columns also from left to right. The reduced boundary matrix can be found on the Figure 6.4.



	1	2	3	4	13	24	12	34 +24 +23 +12	1234
1					1		1		
2						1	(1)		
3					(1)				
4						(1)			
13									1
24									1
12									1
34									(1)

Figure 6.4: Reduced boundary matrix. Lowest ones (which are unique after the algorithm terminates are marked with ovals.

Let us now talk about how to interpret the reduced matrix. I will use a vocabulary of persistent homology even though it is not formally defined yet. To get the idea it is important to follow both the matrix in the Figure 6.4 and the evolution of the filtration presented in the Figure 6.5.

Take the first column of the matrix (indexed by a vertex [1]). It is empty. Empty columns means that a homology class of a dimension equal to the dimension of the corresponding cell (in this case 0) has been created. This is exactly the one connected component on the Figure 6.5(a). The same holds for columns denoted by 2, 3 and 4. The corresponding vertices, introducing new connected components, are on the Figure 6.5(b-d). Then we have a column 13. It is nozero. In that case, we need to look at the lowest one. It is in row 3. It means, that whatever was *created* in the column indexed by 3 is *killed* in this column. Let us have a look at the Figure 6.5(e). An edge 13 appears and the number of connected components went down. So, indeed,

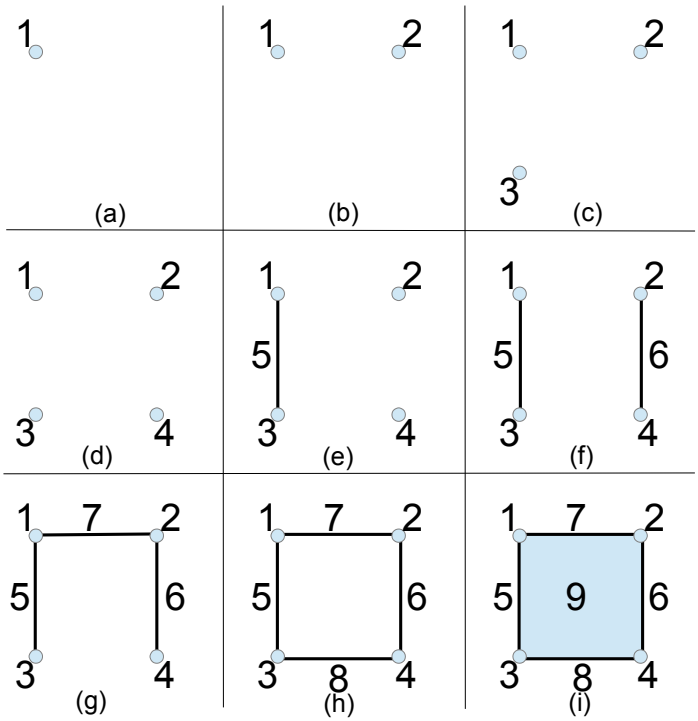


Figure 6.5: Filtration of a complex.

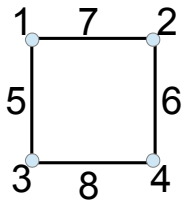
a homology class died. The same situation takes place in the column 24. The lowest one is in the row 4, and therefore whatever was created in the column dented by 4 dies in this column. Again, when we look at the Figure 6.5(f), we will see that adding edge 24 decrease a number of connected components. Analogous argument hold for the column 12. The next column, denoted as $34 + 24 + 23 + 12$ is zero. That means, that a homology class is created there. Please note that $34 + 24 + 23 + 12$ explicitly gives us a representation of the nontrivial cycle that is created. The last column, 1234 has a lowest one 34. This lowest one means that whatever was created in the column 34 (now denoted as $34 + 24 + 23 + 12$) is killed by 1234. Look at the Figure 6.5(i). Adding the two dimensional cell turns a homologically nontrivial cycle $34 + 24 + 23 + 12$ to a trivial one.

Okay, so how to get homology out of it? Simply check, which classes were not killed. In this case, the only class which was not killed is the one in dimension zero, started by the vertex 1. If we do not add a two dimensional cell 1234, we would also have a one dimensional class $34 + 24 + 23 + 12$. It should be also clear, that we can stop the reduction at any stage, and we with this procedure, we will get a homology up to this stage.

Okay, great. Magic. But the obvious question is – why it works? Let's consider a few basic examples. The first one will be just a shorter and simplified version of the situation from the Figure 6.3, see Figure 6.6. There are no conflict until we get to the last column.

Lets make a couple of observations. Firstly, every column in the matrix is either empty (all entries are zero), or it has lowest one. In the first case the boundary of the cell corresponding to this column can be generated from the the cells that are already present in the complex. One may think about this as adding the final cell that close a topologically nontrivial cycle. This is the moment of creation.

In the second case the lowest one in the nth column of the reduced matrix appears if the boundary of a cell corresponding to that column create a cycle that is not a linear combination of the cycles that have been

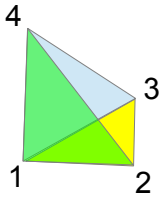


	13	24	12	34
1	1		1	
2		1	1	
3	1			1
4		1		1

Figure 6.6: Filtration of a complex.

generated so far. Since the boundary of a cell is a cycle itself, it means that this cycle cannot be generated as a linear combination of already available cycles, i.e. it is a nontrivial cycle. Addition of a cell make it trivial. So, this is the moment of destruction.

I am not going to make a proof over here, let us convince ourselves that this is the case by looking at the next example presented in the Figure 6.7. In this example we have exactly the same situation, but one dimension higher. A detailed analysis is left to the reader.



	123	124	134	234
12	1	1		
13	1		1	
14		1	1	
23	1			1
24		1		1
34			1	1

Figure 6.7: Filtration of a complex.

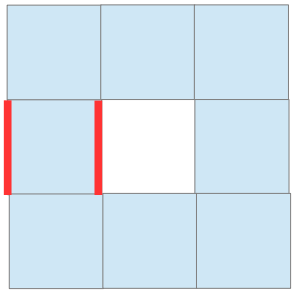
Let us summarize what we have learned. In order to get a dimension of $i-$ dimensional homology group, one need to take a number of zero columns corresponding to the cells in dimension i and subtract number of nonzero columns corresponding to simplices in dimension $i+1$. The first collection of columns corresponds to the group of cycles, the second – to the group of boundaries.

6.1 Comments

1. There exist a theory dual to homology theory called cohomology theory. If there is a time, it will be intuitively introduced at the end of this lecture. Computational cohomology is also useful in applied science. Generating *cocyles*, a concept dual to generating cycles in homology, are explicitly needed in some discrete formulation of Maxwell's equations. In a lot of cases homology and cohomology theory are dual. Do you remember the max-flow-min-cut duality we have been talking about in the first section? If there are no so called *torsions* present in the homology groups[3] then homology and cohomology groups are *dual* (see the Universal Coefficient Theorem for Cohomology [9]). There is a simple interpretation to this available in the picture of annulus below. In red a representative of cohomology generator is given.

[3]Torsions are generators of a finite order in the homology group.

Formally it is so called *cochain*. Think about it as a fence that have to be crossed by any homology generator. This is the basic interpretation of homology-cohomology duality.



This duality is a discrete version of Ampere's law. Let us suppose that the annulus above is a cross-section of a insulating domain. The hole therein is a part of a conducting domain (that close up to a solid torus in 3d). The Ampere's law states that magnetomotive forces on cycles surrounding the branch of a conductor s equal to the current passing through this conductor. It can be interpreted exactly as homology–cohomology duality.

2. Homology over $\mathbb{Z}_2$ are easy to define. In a standard lecture in algebraic topology we always start from homology defined over *integer* coefficients. Due to so called *universal coefficient theorem for (co)homology*, those are the one which carries the most information (see the comment above). The reason why we have chosen to use $\mathbb{Z}_2$ homology (or in general, a *field* homology) is that they are needed for *persistent homology* to be well defined. We will talk about persistent homology in the next lecture.

   I want to point out that the fact that we are using $\mathbb{Z}_2$ homology here do not mean that the (co)homology over integers are not used in applied topology. On the contrary, in some applications they are explicitly needed.

Chapter 7

Persistent homology

So far we have been talking about standard homology. This topic has been known in mathematics since the times of Henry Poincare. One can think about persistent homology as a version of a homology theory where the complex under consideration is not static, i.e. contain all the simplices which were mend to be there. But what happen if new cells are being constantly added to the complex? In this new setup, we are likely to have Betti numbers appearing and disappearing when new simplices appears. To summarize this whole process we will require a notion of "time slots" in which Betti numbers are active. Let us consider the Figure 7.2 as an example.

We will also look at persistent homology from the perspective of point clouds. In this case we will not keep the single, unique distance in the Rips complex, or radius of the ball in the Cech complex fixed. In contrary, we will let it grow, and we will use persistent homology to see how the number of holes of different dimensions are evolving once the radius is growing. Doing so we will obtain a multi-scale summary of the data.

You may ask, why we need a multiscale summary? The reason for that is because often the interpretation of the data depend on the scale, or granularity, you use to look at the data. Given this different interpretations may be equally correct. The Figure 7.1 shows a standard example of a case where an information in the picture cannot be summarized in a single spatial scale.

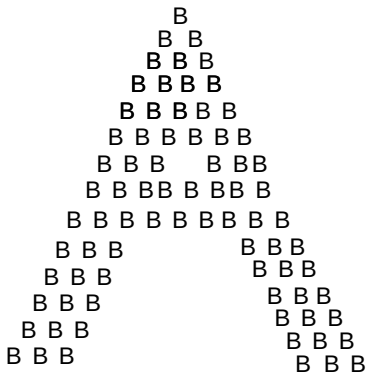


Figure 7.1: Scale matters!

Let us have another look at the example in the Figure 7.2. In this case, we are tracking the connected components of the sublevelsets of a function $S_a = \{x \in \mathbb{R} | f(x) \leq a\}$

In the example in the Figure 7.3 we see an important convention which is often refereed to as *elder rule*. In that case, when two connected components are merged, then the one which is *youngest* die. The reason for that is because we want to keep track of the topological features that lives for longest.

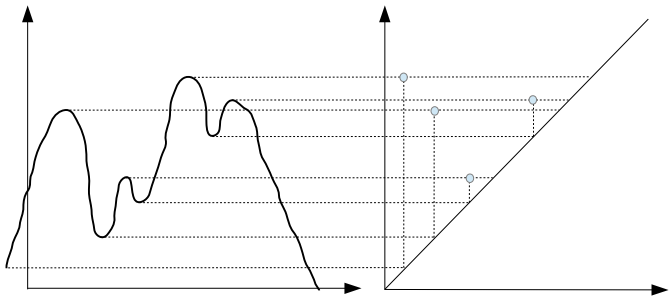


Figure 7.2: Illustration of a basic idea of persistent homology. Let us consider the plot of the function on the left. Let us imagine that this is a surface of a one dimensional terrain and that there is a rain falling over that terrain. We assume that the amount of water that falls at every point is the same no matter where the point is. When the plot of the function is floated, we check how many summits do we see. In that case, obviously the lowest passes will be floated first. We *pair* each summit with the pass the floating of which causes that we stop to see the summit. We draw the differences of the heights in the diagram on the right. In a terms of functions those pairings give us an idea how about a minimal perturbation of a function (in the terms of maximum norm) that causes the minimum and the corresponding maximum to disappear. Those points, also called intervals, that we get, are closely related with the concept of persistent homology.

What we saw in the Figure 7.2 and Figure 7.3 is exactly what the persistent homology is about in the dimension 0. Clearly, in the case of presented functions $f : \mathbb{R} \to \mathbb{R}$ we cannot have more complicated homology than the one in dimension 0. But, this can certainly happens for functions $f : \mathbb{R}^n \to \mathbb{R}$. Let us specify that.

Let $\mathcal{K}$ be a simplicial complex. Let us have $f : \mathcal{K} \to \mathbb{R}$, a filtering function. Let us remind that this implies, that for every $a \in \mathbb{R}$, $f^{-1}(a)$ is a subcomplex of $\mathcal{K}$.

Please note that Vietoris-Rips and Cech complexes comes naturally with a filtration on them which comes from a changing parameter.

In this case a filtration of a complex is a nested sequence of subcomplexes:

$$\emptyset = \mathcal{K}_0 \subset \mathcal{K}_1 \subset \ldots \subset \mathcal{K}_n = \mathcal{K}$$

Given a filtration, we can define a filtering filtering function, by setting $f(a) = min_{i \in \{0,\ldots,n\}} a \in \mathcal{K}_i$. Filtering function on a finite complex also define filtration of this complex in a natural way.

Given the sequence of complexes one included to the following one, we have natural sequence of homomorphisms in homology:

$$0 = H_p(\mathcal{K}_0) \to H_p(\mathcal{K}_1) \to \ldots \to H_p(\mathcal{K}_n) = H_p(\mathcal{K})$$

At each step, when we go from $\mathcal{K}_i$ into $\mathcal{K}_{i+1}$ some of the homology classes may become trivial, and some new ones may appear. It turns out that, given such a sequence, we can choose a basis which is compatible for all the steps of the filtration. An algorithm to compute persistent homology is a constructive proof that such a basis can be picked up. Given such a basis we can track the lifespan of a homology classes. When we first see a class, we say that the class is *born*. When the class became trivial, or identical to other class born earlier, we say that the class *dies*. Let us have a look at the example from the Figure 7.4.

Given this intuition we are now able to provide a formal definition of persistence.

Definition 2 *Let $f_p^{i,j} : H_p(\mathcal{K}_i) \to H_p(\mathcal{K}_j)$ be a homomorphism induced by inclusion. The p-th persistent homology groups are the images of $f_p^{i,j}$ for 0 $leqi \leq j \leq n$.*

A *lifespan* of a homology class is encoded by a persistence interval $[b, d]$, such that $b < d$. The collection of all persistence intervals is usually encoded as so called *persistence diagram*. It is a finite set of points. For

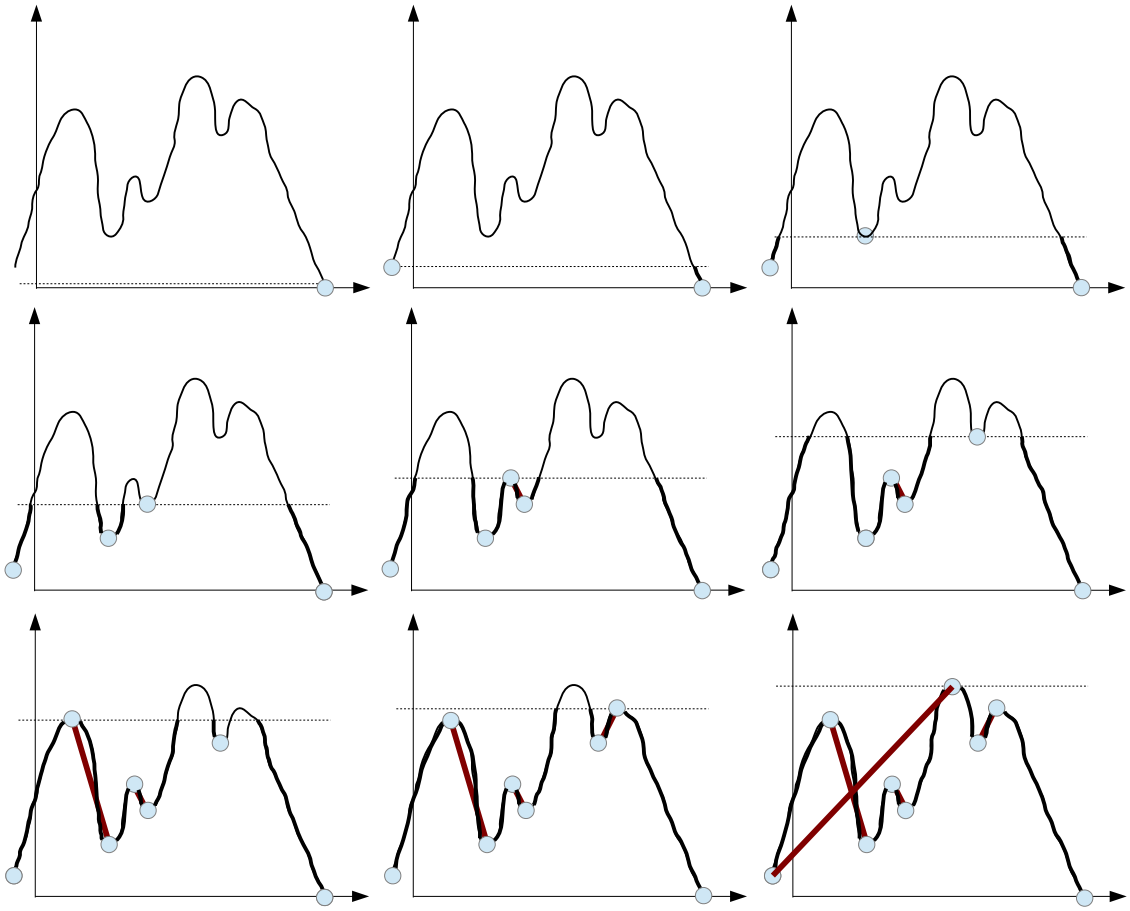


Figure 7.3: The evolution of connected components. Explicit matchings between minima and corresponding saddles are given.

instance, for the intervals $[1, 2]$, $[2, 4]$ and $[3, 4]$ the corresponding diagram is depicted in the Figure 7.5. Let us discuss now the algorithms to compute persistent homology. It turns out that it is analogous to the algorithm to compute standard homology we already know. Let us consider the example presented in the Figure 7.6 for further details. Note that pairings, which are topological invariant, are directly implied by lowest ones. It require a proof to show that those are canonical. For that one, consult a book by Edelsbrunner and Harer, page 154.

7.1 Comments

1. By using ideas from rigorous numerics one can compute persistence not only of a finite complex, but also for a continuous function defined on a compact subset of $\mathbb{R}^n$.

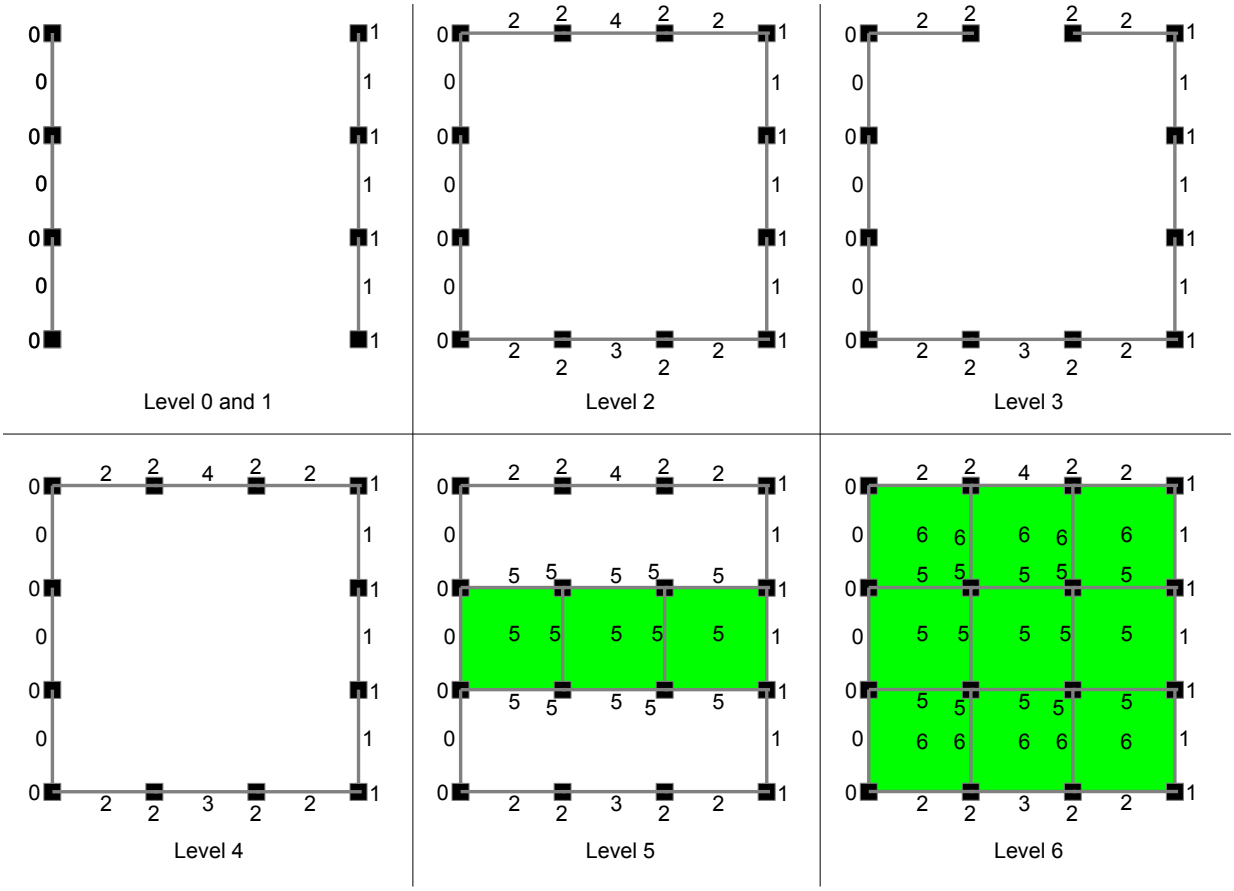


Figure 7.4: The evolution of connected components. In the first picture (level 0 and 1) we have two connected components being born. First in the level 0, second one in the level 1. In he level 2 there is no change in topology. In the level 3 one of the connected component dies. In that case, due to the elder rule, the connected component which was born in the time 1 dies. Its lifespan is therefore $[1, 3]$. In the level 4 the number of connected components remains unchanged, but we have a new $1-$dimensional cycle is created. In the level 5 another $1-$dimensional cycle is created. In the level 6 both one dimensional cycles are killed, and their life span is $[4, 6]$ and $[5, 6]$.

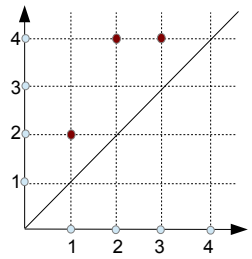
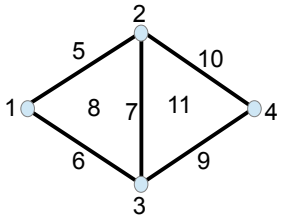


Figure 7.5: Persistence diagram of intervals $[1, 2]$, $[2, 4]$ and $[3, 4]$.



	12	13	23	123	34	24	234
1	1	1					
2	1		1			1	
3		1	1		1		
4					1	1	
12				1			
13				1			
23				1			1
34							1
24							1

Before reduction

	12	13	23 12 13	123	34	24 34 13 12	234
1	1	1					
2	1						
3		1			1		
4					1		
12				1			
13				1			
23				1			1
34							1
24							1

After reduction

Figure 7.6: Example of the matrix reduction algorithm used to obtain persistence. The filtration of verices of the considered complex is 1, 2, 3 and 4. We use this filtration to name the edges and vertices. Numbers by edges and vertices denote their filtration. The first conflict appears on the column 23. It is resolved by adding column 12 and 13 to the column 23. In this way, a empty column is created. It corresponds to a cycle in the subcomplex. The next conflict appears in the column 24 and it is resolved by adding columns 13, 34 and 12 to the column 24. On the right, the reduced matrix is presented. Let us go through it from left to right to read off the persistence. Note that not all the columns are presented there (for sake of space we have dropped the zero columns representing the boundary of vertices). The column 12 has the lowest one 2. That means, that whatever was created in the vertex 2 is killed by the edge 12. That pairing gives rise to the interval $[2, 5]$. Analogously the column 13 gives rise to the interval $[3, 6]$. Column 23 is empty. It means, that a $1-$dimensional cycle is created by adding 23. Column 123 kills the cycle created in the column 23 and gives rise to the interval $[7, 8]$. Column 34 gives rise to the interval $4, 9$ in dimension 0. Column 24 is empty and therefore gives rise to the cycle supported in $12 + 13 + 34 + 24$. Column 234 kills that cycle and give rise to the interval $[10, 11]$ in dimension 1.

7.2 Standard metrics and stability results.

When analysing noisy data we always need to make sure that the result for a noise-less data is not far from the results obtained for a noisy data, i.e. that the method is stable. One of main reasons why persistent homology is useful in applied science is precisely this property: persistence is stable, it is robust. Roughly it means that small change in the filtering function induces a small change in persistence diagram. In order to make this statement precise, we need to introduce a metrics in a space of functions, and more importantly, in a space of persistence diagrams. In the space of functions we will use standard L^∞ metric. Let us define now a metric in a space of persistent diagrams.

Recall that a persistence diagram is a finite multiset of points in the plane $\mathbb{R}^2$ (with points at infinity). For convince, for this finite multiset of points we are adding the points in the diagonal, each with infinite multiplicity. They are needed for the technical reasons. They represent points that are born and dies in the same time. Think of them as virtual pairs of particle and anti-particle. You can get them by infinitesimally small perturbation of the filtration.

Let X and Y be two such persistence diagrams. Let us consider a bijections $b : X \to Y$. The *Bottleneck* distance is defined in the following way:

$$W_\infty(X,Y) = inf_{b:X\to Y} sup_{x\in X} ||x - b(x)||$$

In other words, we consider all possible bijections between (infinite) multisets of points X and Y and we are searching for the one that minimize the maximal distance between a point x and a point $b(x)$ matched with x via this bijection. This concept is closely related with the earth mover distance. It only cares about the maximal distance. There is another distance which is sensitive to all changes. It is a q-Wasserstein distance and it is defined in the following way:

$$W_q(X,Y) = (inf_{b:X\to Y} \sum_{x\in X} ||x - b(x)||^q)^{\frac{1}{q}}$$

Please consult the Figure 7.7 for an example of a matching.

In this section we will provide a stability theorem for the Bottleneck distance. There exists stability theorems for the Wasserstein distances, but they are conceptually more complicated. This stability theorem is due to David Cohen-Steiner, Herbert Edelsbrunner and John Harer [4]. There are much more general stability theorems that work using very mild condition [5] - for a case of Rips complexes with similarity measure.

Theorem 1 *Let $\mathcal{K}$ be a cell complex and $f, g : \mathcal{K} \to \mathbb{R}$ be two filtering functions. For each dimension p the bottleneck distance between the diagram of $\mathcal{K}$ with a function f (denoted as $Diag(\mathcal{K}, f)$) and the diagram of $\mathcal{K}$ with a function g (denoted as $Diag(\mathcal{K}, g)$) satisfy:*

$$W_\infty(Diag(\mathcal{K}, f), Diag(\mathcal{K}, g)) \leq ||f - g||_\infty$$

In other words, when perturbing the filtering function by ϵ, the points in the diagrams will not move further away than ϵ. This is a property which makes persistence useful in the applied science. If time permits we will show how it allows to compute persistence of a continuous function.

Let us put this theorem into a test! We already know how to generate triangulations with Gudhi. Let us generate one, and add a filtration to it:

For the purpose of this exercise we will pick one of the triangulations from the Section in which we have discussed simplicial complexes and assign a filtration (composed for instance from an increasing sequence of integers) to it:

```
import numpy as np
import gudhi as gd
import matplotlib
#Create a simplex tree
#Simplices can be inserted one by one
#Vertices are indexed by integers
#Notice that inserting an edge automatically insert its vertices
```

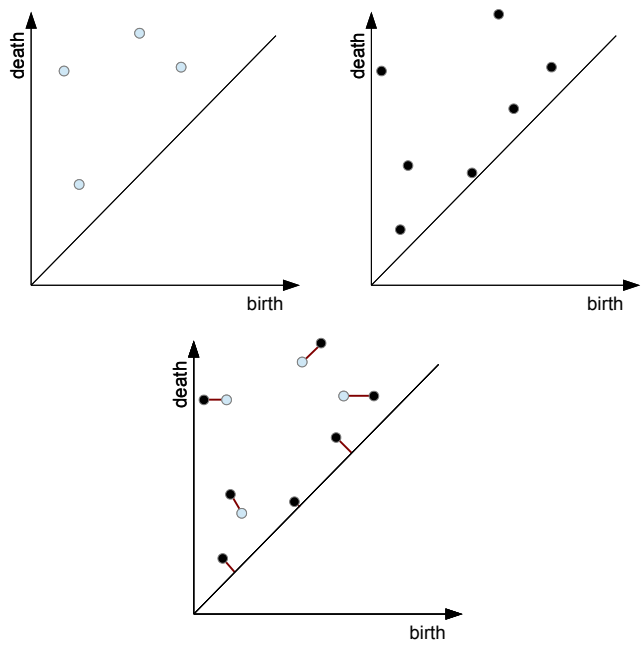


Figure 7.7: Example of matching between persistence diagrams. For illustration we put the persistence diagram from the top left and the top right into one diagram in the bottom, and we make a matching therein.

```
(if they were #not already in the complex)
st = gd.SimplexTree()
st.insert([1,4,8],0)
st.insert([1,2,8],1)
st.insert([2,6,8],2)
st.insert([2,3,6],3)
st.insert([3,4,6],4)
st.insert([1,3,4],5)
st.insert([4,5,9],6)
st.insert([4,8,9],7)
st.insert([7,8,9],8)
st.insert([6,7,8],9)
st.insert([5,6,7],10)
st.insert([4,5,6],11)
st.insert([1,2,5],12)
st.insert([2,5,9],13)
st.insert([2,3,9],14)
st.insert([3,7,9],15)
st.insert([1,3,7],16)
st.insert([1,5,7],17)

diagram = st.persistence(persistence_dim_max=True)

plt = gd.plot_persistence_diagram(diagram)
plt.show()
```

The numbers after comma (ranging from 0 to 17) indicate the filtration value of top dimensional simplices. This is propagated to lower dimensional simplices via so called lower star filtration (i.e. a lower dimensional simplex get a filtration value equal to minimum of filtration values of adjusted top dimensional simplices).

Now, to illustrate stability theorems, let us perturb the filtration and visualize it, and compute the Bottleneck distances between diagrams. Those bottleneck distances will be bounded by the magnitude of noise we add to the filtration. Below some hint on how to use random numbers generator to perturb the filtration.

```
import random as rd
rd.seed()
rd.random()
```

Also remember that in order to compute the Bottleneck distance between two diagrams use:

```
gd.bottleneck_distance(diagram, diagram1)
```

Please do it on your own! If there is any problem, you can find the whole code below (with the white ink, sorry... :) You can also find it in the attached directory tree.

Chapter 8

Experiments with Persistent homology

In this section we are considering a number of exercises involving persistent homology computations. Please go through them and let me know if you have any questions or problems.

8.1 Open and closed proteins

This example was entirely designed by Bertrand Michel and is available at his tutorial in here: `http://bertrand.michel.perso.math.cnrs.fr/Enseignements/TDA/Tuto-Part2.html`. It also bases heavily on the work of Peter Bubenik. Big thanks to both of them.

In ths example we wil work on the data also considered in the paper `https://arxiv.org/pdf/1412.1394`

The data required for this exercise are available here: `https://www.researchgate.net/publication/301543862_corr`

The purpose of this exercise is to understand if persistent homology information can determine the folding state of a protein. Please load the data, unpack them and put them all to the *data* folder. For a further detail please consult the paper. Here are the steps we are gong to perform in the calculations:

1. Read the appropriate correlation matrices from csv files (we will use Pandas for that).
2. Transform a correlation matrix to a pseudo–distance matrix by transforming each entry a_{ij} into $1 - |a_{ij}|$.
3. Compute the Rips complexes of the obtained distance matrices (note that in this case, we are constructing a Rips complex of a pseudo–metric space which is not a Euclidean space. Such a situations are very typical in this type of bio–oriented research).
4. Compute persistent homology of the obtained complexes.
5. Compute all-to-all distance matrix between the obtained persistence diagrams (do it dimension–by–dimension). For that purpose we will use Bottleneck distance implemented in Gudhi.
6. Use Multidimensional Scaling method implemented in the scikit-learn library to find two dimensional projection of the data.
7. As you can see, in this case we get sort of separation, but not a linear one. For a further studies of what can be done with those methods, please consult the original paper `https://arxiv.org/pdf/1412.1394`.

```
import numpy as np
import pandas as pd
import pickle as pickle
import gudhi as gd
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
```

```
from IPython.display import Image
from sklearn import manifold
import seaborn as sns

#Here is the list of files to import
files_list = [
'data/1anf.corr_1.txt',
'data/1ez9.corr_1.txt',
'data/1fqa.corr_2.txt',
'data/1fqb.corr_3.txt',
'data/1fqc.corr_2.txt',
'data/1fqd.corr_3.txt',
'data/1jw4.corr_4.txt',
'data/1jw5.corr_5.txt',
'data/1lls.corr_6.txt',
'data/1mpd.corr_4.txt',
'data/1omp.corr_7.txt',
'data/3hpi.corr_5.txt',
'data/3mbp.corr_6.txt',
'data/4mbp.corr_7.txt']
#Read the files:
corr_list = [pd.read_csv(u , header=None,delim_whitespace=True)
for u in files_list]
#And change correlation matrix do a distance matrix:
dist_list = [1- np.abs(c) for c in corr_list]

#Compute persistence in dimension 1 for all the files.
#Visualize the persistence diagrams for some of them
persistence = []
for i in range(0,len(dist_list)):
        rips_complex = gd.RipsComplex(distance_matrix=dist_list[i].values,max_edge_len
        simplex_tree = rips_complex.create_simplex_tree(max_dimension=2)
        simplex_tree.persistence()
        persistence.append( simplex_tree.persistence_intervals_in_dimension(0) )

#And compute all-to-all bottleneck distances. Note that this part
#will take a few seconds:
dist_mat = []
for i in range(0,len(persistence)):
        row = []
        for j in range(0,len(persistence)):
                row.append( gd.bottleneck_distance(persistence[i], persistence[j]) )
        dist_mat.append(row)

#We will now use a dimension reduction method to
#visualize a configuration in  R^2  which almost
#matches with the matrix of bottleneck distances.
#For that purpose we will apply a Multidimensional
#Scaling method implemented in the scikit-learn library.

mds = manifold.MDS(n_components=2, max_iter=3000, eps=1e-9,dissimilarity="precomputed"
pos = mds.fit(dist_mat).embedding_
```

```
plt.scatter(pos[0:7,0], pos[0:7, 1], color='red', label="closed")
plt.scatter(pos[7:len(dist_mat),0], pos[7:len(dist_mat), 1], color='blue', label="open
plt.legend( loc=2, borderaxespad=1)
plt.show()

#repeat this for diagram in dimension 0.
```

In one of the next sections we will be discussing persistence representations, and one example we will consider there uses permutation test. Feel free to get back then to this example and use permutation test to distinguish two types of proteins.

8.2 Classification of smart phone's sensor data

In this exercise we will gather accelerometer data of various activities using our mobile phones. For that we need an app to record data on our phones. The one I am using is an android app "Physical Toolbox". You can find it on a play store `https://play.google.com/store/apps/details?id=com.chrystianvieyra.physicstoolboxsuite&hl=en_US`.

Due to the hardware restrictions I was not able to test in practice the i-phone analogue, but I think that one of those may work for you if you are an apple user:

1. VibSensor,
   `https://itunes.apple.com/us/app/vibsensor-accelerometer-recorder-vibration-analysis/id932854520?mt=8`

2. Physical toolbox – presumably restricted version of the android app we have discussed above
   `https://itunes.apple.com/us/app/physics-toolbox-sensor-suite/id1128914250?mt=8`
   and
   `https://itunes.apple.com/us/app/physics-toolbox-accelerometer/id1008160133?mt=8`

3. Accelerometer app
   `https://itunes.apple.com/us/app/accelerometer/id499629589?mt=8`

With this, we need linear accelerometer, and start recording at the right moment. The aim is to record various activities, like walking on stairs, normal walking, push ups, jogging, and whatever else you want. Remember to stop recording as soon as you stop the activity. Given the data, the app will store it in a csv file, which we will use for a later analysis.

Please grab your phones and record accelerometer data when different activities are performed. At the end, export your file, name it with the name of activity, and store it. You shall get file the header of which look like this:

	A	B	C	D	E
1	time	ax	ay	az	aT
2	0.012	0.2371	-0.3765	-0.5529	0.71
3	0.013	0.2832	-0.2754	-0.7395	0.838
4	0.014	0.3295	-0.2251	-0.5975	0.718
5	0.014	0.3179	-0.127	-0.6852	0.766
6	0.027	0.3098	-0.1268	-0.597	0.684
7	0.031	0.3169	-0.1346	-0.8348	0.903
8	0.046	0.0968	-0.1718	-0.7343	0.76

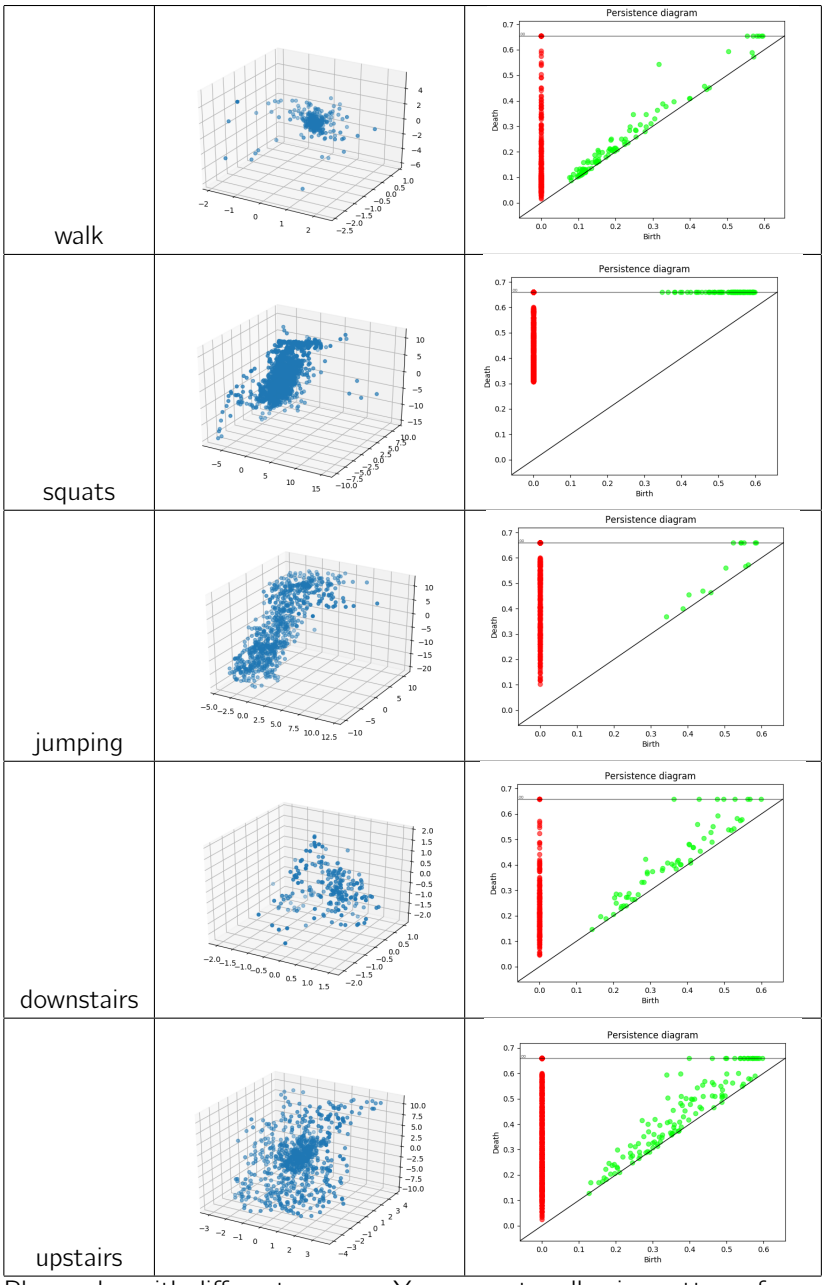
Remove the first row and the first and last column, and read it.

```
import numpy as np
import pandas as pd
import pickle as pickle
import gudhi as gd
from pylab import *
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
from IPython.display import Image
from sklearn.neighbors.kde import KernelDensity
import matplotlib


data_pd = pd.read_csv('activities/walk.csv',decimal=".",delimiter=",")
data = data_pd.values

data = np.delete(data, (0), axis=1)
data = np.delete(data, (3), axis=1)

#now we can visualize the data:
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
#plt.plot(data [:,0],data [:,1],data [:,2] )
ax.scatter(data[:,0],data[:,1],data[:,2] )
plt.show()

#And now we can compute persistent homology of the data and compare
#(visually at the moment) different activities.


Rips_complex_sample = gd.RipsComplex(points = data,max_edge_length=0.6 )
Rips_simplex_tree_sample = Rips_complex_sample.create_simplex_tree(max_dimension=2)

diag_Rips = Rips_simplex_tree_sample.persistence()
#diag_Rips
#diag_Rips_0 = Rips_simplex_tree_sample.persistence_intervals_in_dimension(0)
plt = gd.plot_persistence_diagram(diag_Rips)
plt.show()
```

Note that depending if you have python 2 or 3, you may get some problems with reading csv. This is something we need to work out on the flight.

Check if persistent homology give you a way to cluster the activities.

Here are a few examples of my activities, their point clouds, and persistence diagrams obtained for the parameters of the Rips construction as in the example above.



Please play with different sensors. You can get really nice patterns from magnetometer. Run it, rotate your phone, and see what you get!

Check it up for your case! Think of a characteristics of persistence diagrams that discriminate those activities. Think of alternative (to persistent homology) characteristics of point clouds that could differentiate the type of activity. Think about it next time you give an app a permission to access your phone sensor. And share your ideas and observations!

8.3 Back to the point clouds

So far we have generated a number of point clouds. Computing their persistent homology is the first task we can do in this section. In fact, computations of persistent homology is an optional part in all the codes available therein. The point that is missing is to visualize persistence diagrams. This can be achieved using the following code:

```
import matplotlib
#suppose that over here the simplex tree has been generated.
plt = gd.plot_persistence_barcode(pers)
plt.show()
```

for persistence barcodes or

```
import matplotlib
#suppose that over here the simplex tree has been generated.
plt = gd.plot_persistence_diagram(pers)
plt.show()
```

for persistence diagramss

Please carry on the experimentation with all the point clouds and check if the results are as predicted.

Try to do the sampling a few times. Plot the persistence diagrams. Are they similar? Verify this by computing the distances between them. For that purpose you can use either python-based Wasserstein distance, or C++ based persistence representations. If only you take a reasonable number of points sampled from the set, you will see almost exactly the same persistence diagram. There is in fact a *stability result* saying that if so called *Hausdorff distnace* between point clouds is bounded by ϵ, then the persistence diagrams of that point clouds are bounded by the same ϵ subject to a few mild assumptions.

8.4 Random cubical complexes and generalized percolation

In this exercise we will create a number cubical complexes according to the following model: A top dimensional cube is present in the complex with probability $p \in [0, 1]$. Pick a few values of p, and generate a few instances of cubical complexes (choose dimension and size as you wish, but be reasonable as you do not want to blow up your memory). Compute persistent homology of those complexes. What can you observe? Can you track for evidences of phase transitions happening in the system? How this experiment is related to percolation theory.

Use the code below as a starting point in your experiments:

```
import matplotlib.pyplot as plt
import numpy as np
import random as rd
from mpl_toolkits.mplot3d import Axes3D
import csv
from sklearn import decomposition
import math
import gudhi as gd
from PIL import Image

#in this example we are considering two dimensional complexes
N = 100

#this is where you set the probability:
p = 0.6

bitmap = []
for i in range(0,N*N):
        x = rd.uniform(0, 1)
        if ( x < p ): bitmap.append(1)
        else: bitmap.append(0)

bcc = gd.CubicalComplex(top_dimensional_cells = bitmap, dimensions=[N,N])
```

```
diag = bcc.persistence()
#now we can check how many generators in dimension 0 and in dimension 1 we have:
dim0 = bcc.persistence_intervals_in_dimension(0)
dim1 = bcc.persistence_intervals_in_dimension(1)

print "Here is the first Betti number: ", len(dim0)
print "Here is the second Betti number: ," len(dim1)
```

You can also modify this exercise to create cubical complexes such that cubes get not only filtration 0 or 1, but the whole spectrum $[0, 1]$. Another interesting experiment aims in picking a grid $0 \leq p_1 < p_2 < \ldots < p_n \leq 1$, computing the Betti numbers for each probability, averaging them for each probability and plotting the results.

8.5 Ssid walks

If during your ssid–walk you have made a cycle, you can now go back to the data, and check if you can see the dominant interval in one dimensional persistent homology. How the persistence of this interval compare to the persistence of other (shorter) intervals that come from the noise.

Chapter 9

Persistence representations.

In this section we will take a look at various alternative ways of representing persistence diagrams which is suitable for statistical operations. For many years scientists are trying to provide a sound way of doing statistics on diagrams. The most obvious approach which has been used already many time is to compute some function of a diagram like:

1. Average length of an interval.
2. Median of interval's length.
3. Average of squares of length.
4. Sum of length's squares.
5. Length of maximal interval.
6. ...

All of this is done either to do a hypothesis testing, or to project the data from the space of persistence diagrams down into $\mathbb{R}$, where averaging is much easier to do.

Let us discuss more the problem with averages. Why one cannot define an average on diagrams? There exist a concept of a Frechet mean for a metric space. It can be used to define an average of two diagrams in the following way. Take two diagrams D_1 and D_2. Compute a minimal matching between them. For every $x \in D_1$ matched with $y \in D_2$, take a midpoint of a line segment between x and y as a element of the averaged interval. See Figure 9.1

But there is a problem with a concept of Frechet mean. There are situation when it is not unique. Consult the Figure 9.2 for an example.

I like to think about persistent diagrams as of natural numbers. In both cases, there is no clear way how to define averages. In the case of natural numbers a solution was to start using rational numbers. What is the corresponding rational number in the case of persistence diagrams? For that aim we construct an embedding into various functional spaces. We will present a few of them.

9.1 Persistence Landscapes

One option was proposed by Peter Bubenik[3] and implemented by me [16]. This is the idea of persistence landscapes.

Let us fix first the multiset of persistence intervals $\{(b_i, d_i)\}_{i=1}^{n}$. For every of the intervals let us define a function:

$$f_{(b,d)} = \begin{cases} 0 & \text{if } x \notin (b, d) \\ x - b & \text{if } x \in (b, \frac{b+d}{2}] \\ -x + d & \text{if } x \in (\frac{b+d}{2}, d) \end{cases} \tag{9.1}$$

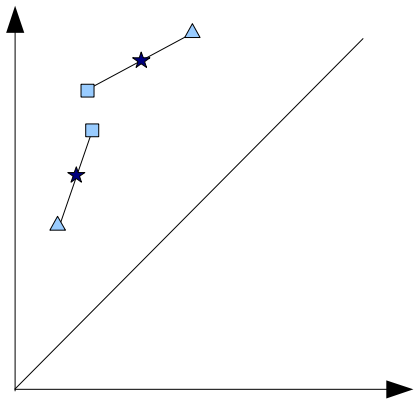


Figure 9.1: Frechet mean of two diagrams. Elements of the diagram D_1 are marked with squares. Elements of the diagram D_2 are denoted with triangles. Elements of an averaged diagram are marked with stars.

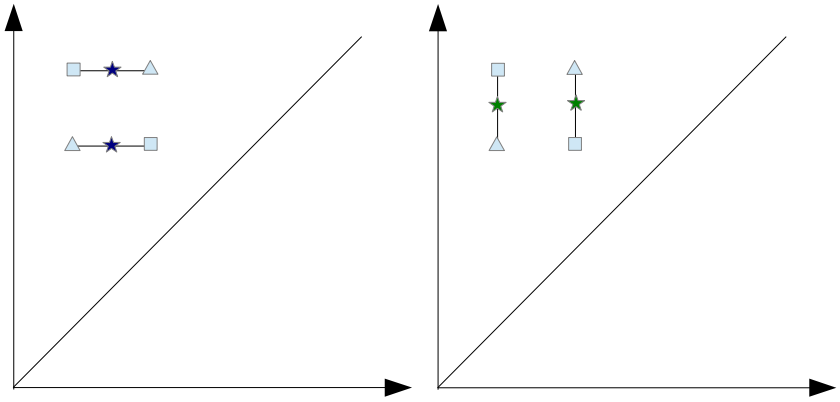


Figure 9.2: Problem with a Frechet mean. In this case we have two possible matching between diagrams. Using one of them gives us one possible average. Using another, gives an average which can be arbitrary far from the first one.

Then a *persistence landscape* is a set of functions $\lambda_k : R \to R$ such that $\lambda_k(x) =$ k-th largest value of $\{f(b_i, d_i)(x)\}_{i=1}^n$. This may be a little hard to digest at the first view. Let us take a look at the Figure 9.4 to get the idea.

Persistence landscape is an equivalent way of representing persistence diagram. There are algorithms to get one representation from anther. Also, given the intuition from the Figure 9.4 it is clear, that $\lambda_{n+1} \leq \lambda_n$.

Lets talk a little bit more about the idea of this representation. What if, at a given $x \in \mathbb{R}$, we have nonzero landscapes $\lambda_1, \ldots, \lambda_n$ and $\lambda_{n+1} = 0$? Well, it means that at the level x, our filtered complex have the rank of the homology in the considered dimension equal n. Moreover, the minimal amount of perturbation we need to make to kill at least one homology generator is $\lambda_n(x)$.

The obvious question is – how introducing persistence landscapes helps us to solve the problems we have

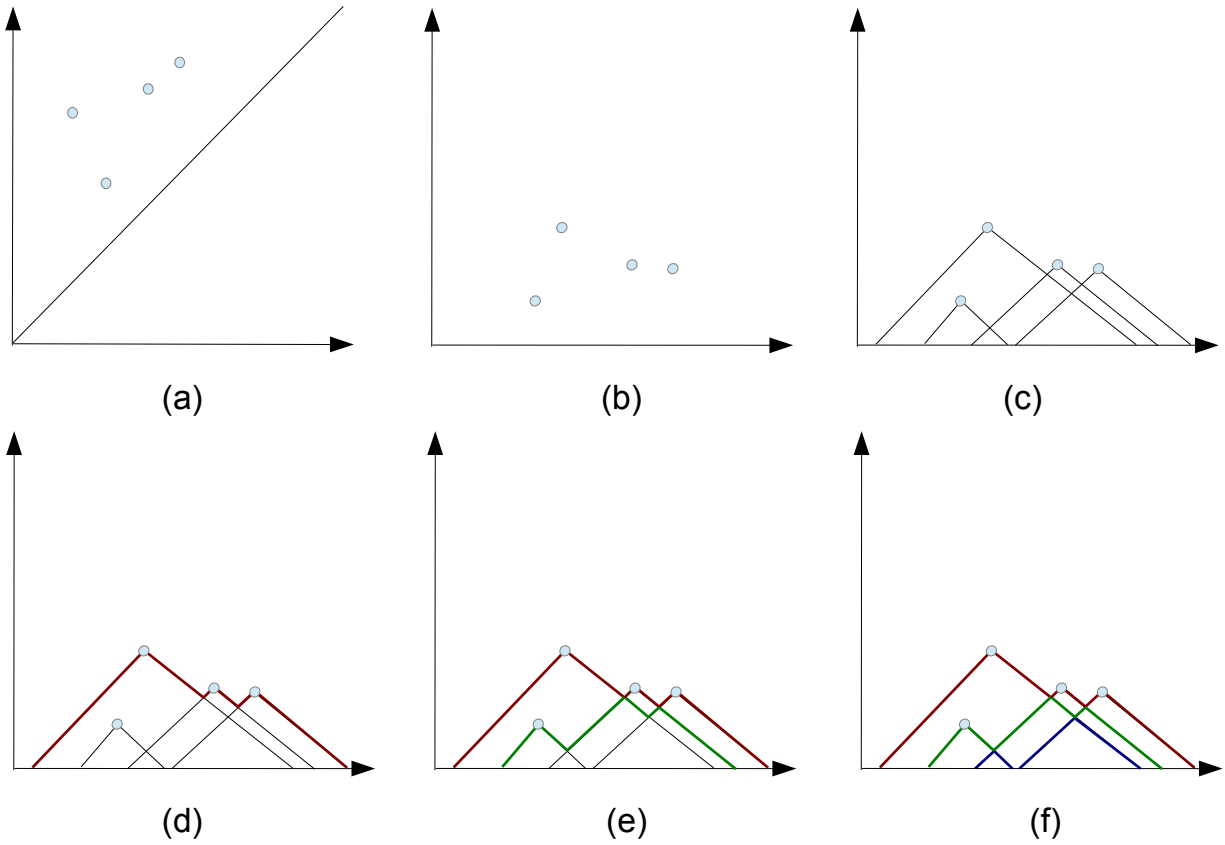


Figure 9.3: An idea of a persistence landscape. First, we "lie down" the persistence diagram (b). Formally this correspond to move from $(birth, death)$ coordinates into $(middlife, half\ life)$ ones. Then, for image of every interval (b, d) in the new coordinates, we draw a plot of the function $f_{(b,d)}$ as in (c). Then, the fists landscape function, λ_0 is depicted in the Figure (d), the λ_2 in (e) and λ_3 in (f).

with an average? Well, it does. An average of two functions $\lambda_i : \mathbb{R} \to \mathbb{R}$ and $\lambda_i' : \mathbb{R} \to \mathbb{R}$ is simply a point-wise average of them. So, the average is well defined, it is unique. The price to pay is that average of two landscapes is a partially linear function which typically do not come from any persistent diagram. This is something that make some people worry. In my opinion it is a very typical situation that in order to define an average of two nice objects, we need to allow ourselves a little more freedom, and move to a larger space. Only then one can state a definition that make sense. I think it is the case for persistence landscapes.

But, lifting the persistence diagrams to the $L^p(\mathbb{R} \times \mathbb{N} \to \mathbb{R})$ space, for $p \in [1, \infty]$ give us more. There is a natural distance function in this space: $d(f, g) = (\sum_{i \in \mathbb{N}} \int_{\mathbb{R}} |f_n(x) - g_n(x)|^p)^{\frac{1}{p}}$. Moreover those distances are stable.

The persistence landscape toolbox and the Gudhi implementation of persistence landscapes have the following functionalities:

1. Computations of a distance matrix.
2. Various plotting procedures.
3. Computations of averages and standard deviation.

9.2 Persistence Heat Maps

Many groups in different time have come up to an idea of building a functional descriptor of a persistence diagram by placing a kernel (typically Gaussian kernel) on each point of a diagram. In order to make this construction stable some type of *weighting* is necessary: otherwise on persistence points of very low persistence one should place the same kernel as the points of high persistence. As the first ones, can appear and disappear in any number even when a small amount of noise is introduced, they can change the resulting function by an arbitrary factor. To make this descriptor stable two strategies have been implemented:

1. Weight the kernel placed on a point (b, d) by the total persistence of the point $b-d$. This strategy has been used in so called Persistence Weighted Gausian Kernel proposed by Yasuaki Hiraoka and collaborators [6] as well as Persistence Images [8].
2. Weight all the kernels by the same number, but for any kernel placed in the point (b, d) place the symmetric kernel at the point (d, b) with the weight -1. This strategy has been proposed in the Persistence Stable Space Kernel [10].

An example of a kernel with no weight is presented in the Figure 9.4.

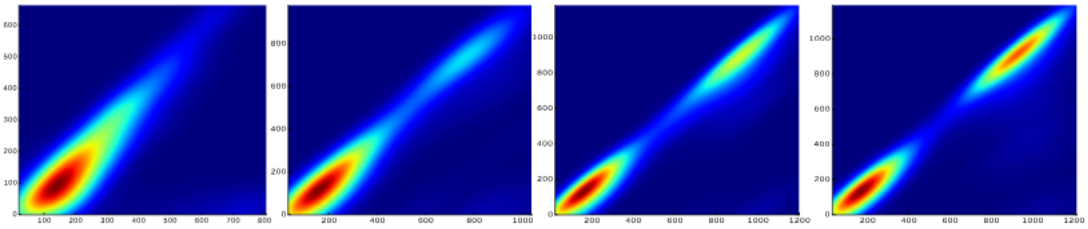


Figure 9.4: Persistence images obtained from diagrams describing neuronal morphologies, see [13] for details.

As much as Persistence Landscapes, various persistence heat maps enable all the main operations required in statistics and machine learning.

9.3 Persistence representations in Gudhi

All those persistence representations, and more, have been implemented in the package *Persistence Representations* available in Gudhi. At the moment unfortunately they are not available in python, but this will change soon.

In the next section we present an exercise we can do to familiarise ourselves with the concept. Unfortunately we do not yet have the persistence landscapes available in python, so to do that we need to make our hand dirty with C++. We are here to help you with that!

9.4 Exercise: detecting the dimension of a random point cloud examination of their persistence diagrams.

At the moment the exercise require, unfortunately working on the C++ level, since some functionalities related to averaging various representations of persistence, has not been cytonized yet. For that purpose, please download the Gudhi library from `http://gudhi.gforge.inria.fr/`, and subsequently use the attached

`https://www.dropbox.com/s/4js8g9k0evnzd1o/study_dimensions_with_persistence.zip?dl=0`

C++ file which need to be added to the Gudhi library and compiled with it. Please consult the linked file for a further technical details. For the purpose of the exercise, please generate a number N random point clouds for two different dimensions d_1 and d_2. Then generate a random point cloud in $\mathbb{R}^{d_1}, \mathbb{R}^{d_2}$. Using a permutation test, please check if there is a statistical difference between Persistence Landscapes of different dimensions. Any necessary details about this task will be given on demand, so please ask!

9.5 Permutation tests

Permutation test – Suppose we are given two sets A, B of persistence diagrams and we would like to show that they are essentially different. For simplicity let us assume that $|A| = |B|$. If we have some assumptions, we can use t-test (implemented in the PLT). If not, we can always use the permutation test. It works as follow:

STEP 1: Compute averages $\hat{A}$ = average of landscapes constructed based on the diagrams from set A, $\hat{B}$ = average of landscapes constructed based on the diagrams from set B. Let D =distance between $\hat{A}$ and $\hat{B}$. Let $z = 0$.

STEP 2: Put all persistence diagrams from A and B to a set C. Shuffle the set C, divide it into two new sets E and F. Compute averages $\hat{E}$, $\hat{F}$ and a distance $\hat{D}$ between them. Every time $\hat{D} > D$, increment the counter z.

STEP 3: Repeat step 2 a N times (typically 10000-1000000 times).

The only output from this procedure we care about is the counter z. Given it, we compute the ration $\frac{z}{N}$. If it is small, that means that the clusters are well separated. If it is large, then just the opposite. In statistics the magic value below which the clusters are considered to be separated well is $\frac{5}{100}$. Also note that here, efficient computations of averages and distances is crucial.

9.6 Comments

The idea of persistence landscapes is in essence similar to the idea of *size functions* introduced by group of Massimo Ferri, Patrizio Frosini and Claudia Land from Bologna in early '90. Long before the concept of persistence was introduced, they had a persistence in dimension zero. In order to measure it, they have been using size functions.

Chapter 10

Sliding window embedding.

In the last sections I tried to convinced you that topology may be a good tool to look at in the applied science. Typically the inputs are in a form of point clouds, cubical or simplicial complexes or continuous functions. It turns out that topology may be useful also to analyse time series. In particular it is a handy tool to verify (semi)periodicity in data[1]. In that case, the input is a time series, i.e. a sequence of measurements, typically taken over constant time intervals. In this section we will give a raw intuition on what are the characteristics of (semi)-periodicity of time series that can be captured with persistent homology. For a further, more formal, treatment please consult [1].

Let us start with a motivating example, a platonic idea of a periodic function. The $sin(x)$, Figure 10.1.

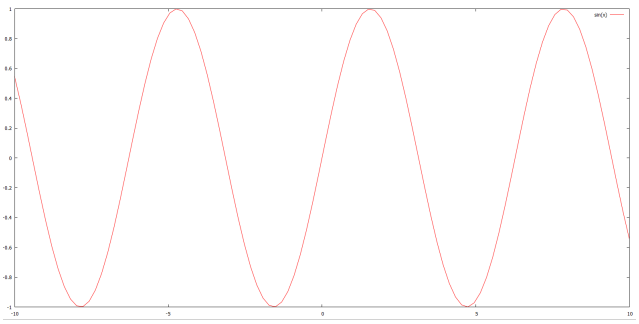


Figure 10.1: sin(x).

Let us start with a very simple experiment. You can use python, C++ or even exel to perform it. Let us generate a subdivision of an interval $[0, 2\pi]$ into a number of points $0, \epsilon, 2\epsilon, \ldots, 2\pi$. Given that, let us create a point cloud cloud by computing values $sin(x_i), sin(x_{i+1}$, where x_i and x_{i+1} are all the constitutive grid elements. By doing so we get:

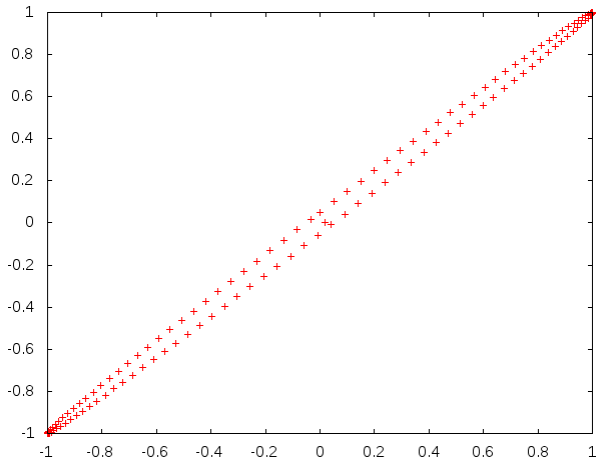
$$sin(0), sin(\epsilon)sin(\epsilon), sin(2\epsilon) \ldots sin(n\epsilon), sin((n+1)\epsilon) \ldots$$

Let us then draw this point cloud. What we see? We see some embedding of an ellipse. Quite flat one, but an ellipse, see Figure 10. It turns out that the point cloud get more *round* if the length of the sliding window embedding is comparable to the period of function [1].

This is the general idea of a sliding window embedding. We group N (2 in example above) values of the time series into a single point. By doing so we transform a time series into a point cloud[2]. Given this, we can measure the periodicity by checking how round this point cloud is – in a sense it give us a sort of periodicity (or cyclicity to be precise) measure. Instead of giving a proof of this statement, we will present an intuition. Suppose we are dealing with continuous, smooth function f we sample in the points $p_1, \ldots, p_n$ to obtain a time

[1] I will remain vague here what exactly I mean by periodicity and semi-periodcity.

[2] Note that we are losing the information about time when performing this construction – the order of points indicate the direction of time, but it is to some extend disregarded in this construction.



series. For the sake of argument let us assume that the period of f is ϵ. Then from smoothness of f a point $f(x_i), \ldots, f(x_{i+N})$ will be close to a point $f(x_i+\epsilon), \ldots, f(x_{i+N}+\epsilon)$. Most likely it will not be the same point, but due to the continuity of f it will be close. It will also be far away from the intermediate points in between $f(x_i), \ldots, f(x_{i+N})$ and $f(x_i+\epsilon), \ldots, f(x_{i+N}+\epsilon)$. Given this, starting from the point $f(x_i), \ldots, f(x_{i+N})$, moving forward we move away from it, and then come back when approaching the point $f(x_i+\epsilon), \ldots, f(x_{i+N}+\epsilon)$. This induce a circle topology which can be detected with persistent homology. Given a periodic, or semi periodic signal, we can use the lifespan of the corresponding persistence generator as a score of how periodic the signal is. If it is small, we have very little evidence to support such a claim.

But, from the picture we made we can clearly see that for platonic ideal of a periodic function the ellipse we get do not look very convincing. The reason for that the embedding we have chosen somehow random parameters and an embedding into a low dimensional space (embedding to $\mathbb{R}^2$).

Note that everything I wrote above is true under assumption that a signal is periodic. There are known cases when a signal is not periodic (even chaotic), but we see a clear cycle in persistent homology. An example of this situation is given below:

Let us consider a function $f(x) = sin(x) * w(x)$, where $w(x)$ is a random function constant on the intervals $[i\pi, (i+1)\pi]$ and equal 1 or -1 on those intervals (probability of each of them is $\frac{1}{2}$. Once we take sufficiently large domain in which the function $w(x)$ have both values, we will get almost exactly the same point cloud as from the sliding window embedding of a function $g(x) = sin(x)$. Yet, the function f is clearly not periodic. There are ways of detecting this case, but they will not be discussed in this lecture.

As an exercise, we will first generate the point cloud coming from the function $sin(x)$. Please do it yourself, and if needed, consult the code provided below.

```
import numpy as np
import gudhi as gd
import math
#construction of the grid points x, and the values therein, y = sin(x).
arg = np.linspace(0,10*math.pi,1000)
values = np.sin(arg)
#Size of the sliding window:
N = 200;
swe = []
for i in range(0,len(values)-N):
        point = []
        for j in range(0,N):
                point.append( values[i+j] );
        swe.append( point );
```

```
#Now we have the point cloud, and we can compute the persistent homology
#in dimension 1:


rips_complex = gd.RipsComplex(points=swe,max_edge_length=20)
simplex_tree = rips_complex.create_simplex_tree(max_dimension=2)
simplex_tree.persistence()
persistence = simplex_tree.persistence_intervals_in_dimension(1)
persistence
```

In the code above please experiment with N (the size of the sliding window), max_edge_length (note that for higher dimensions it should be increased to glue the cycle). Please convince yourself, that we see a dominant cycle in persistence, and that it comes from the fact that we are dealing with a periodic function here. For that compare the persistence of the cycle you get with the persistence of other classes (in dimension 0). Please be careful, since for some parameters the program may consume all your ram and force you to restart your computers.

Having this part of the exercise done, we shall add a little bit of noise to the function, and check if sliding window embedding still successfully detect cycle coming from the periodicity of the underlying function. For that purpose, when constructing sliding window embedding add a minimal random perturbation to the points. Note that in this case you are likely to get a lot of "noisy" circles, therefore use the gudhi tools to visualize them.

```
import numpy as np
import gudhi as gd
import random as rd
import matplotlib

import math
#construction of the grid points x, and the values therein, y = sin(x).
arg = np.linspace(0,10*math.pi,1000)
values = np.sin(arg)
#Size of the sliding window:
N = 200;
swe = []
rd.seed()
for i in range(0,len(values)-N):
        point = []
        for j in range(0,N):
                point.append( values[i+j]+rd.random()*1 );
        swe.append( point );
#Now we have the point cloud, and we can compute the persistent homology
#in dimension 1:


rips_complex = gd.RipsComplex(points=swe,max_edge_length=20)
simplex_tree = rips_complex.create_simplex_tree(max_dimension=2)
persistence = simplex_tree.persistence()
#persistence = simplex_tree.persistence_intervals_in_dimension(1)
plt = gd.plot_persistence_diagram(persistence)
plt.show()
```

Given this additional experience we can play now with some real periodic, or semi periodic data. Let us have a look at the webpage:
`https://datamarket.com/data/set/2324/daily-minimum-temperatures-in-melbourne-australia-1981-1990#`

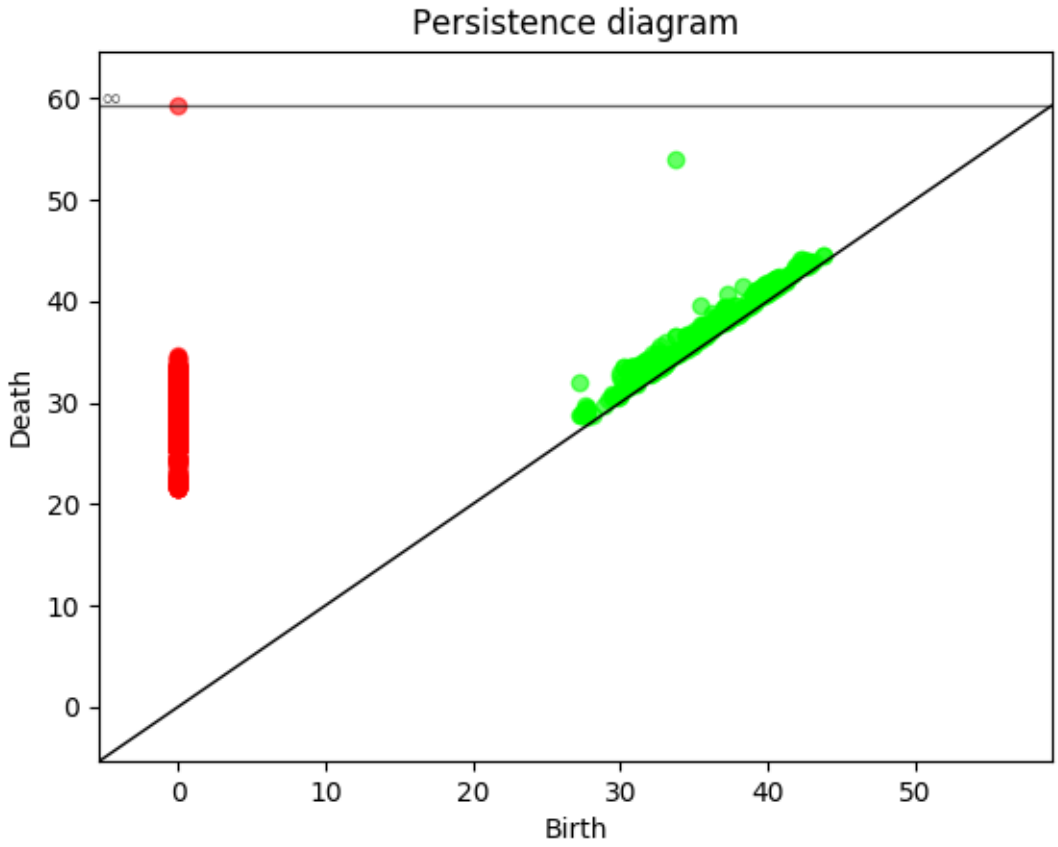
!ds=2324&display=line. You can get from there the spreadsheet with daily minimum temperatures from Melburne in the years 1981-1990. Those data should be almost periodic (almost, since it is not expected that temperatures yesterday, today and tomorrow will be exactly the same as last year those days). Let us try to use the tools we have just learned to see if a sliding window embedding generate a reasonably long living persistent homology class in dimension 1. Note that for some reason the last line of the csv files you have imported looks like having three positions, and pandas are having problem with it. Please remove the first and the last line before going to execute the code below. Also some of the temperature readings have a character ? in it. Please edit the csv files and remove this characters, since it will cause problems when reading the file.

Watch out, since this example tends to consume a lot of RAM! If you do not have too much of it in your machine, may want to restrict the data to some smaller time interval.

```
import numpy as np
import gudhi as gd
import random as rd
import matplotlib
import pandas as pd
import math

data_pd = pd.read_csv('semi_periodic/daily-minimum-temperatures-in-me.csv')
#We can now remove the first column:
data = data_pd.values
data = np.delete(data, (0), axis=1)

#convert it from multi dimensional array to one dimensional array:
data = data[:,0]

#in order to make the computations feasible, we should restrict the data a bit.
#experiment with various ranges, do not make it too large unless you want to restart
#your computer :)
#data = data[0:700]
#data = data[700:1500]
data = data[500:1500]

values = data.astype( np.float )

#Size of the sliding window. We will try to set it close to
#the predicted value of period, which is in this case, the
#number of days in a year.
N = 100;
swe = []
for i in range(0,len(values)-N):
        point = []
        for j in range(0,N):
                point.append( values[i+j] );
        swe.append( point );
#Now we have the point cloud, and we can compute the persistent homology in dimension


rips_complex = gd.RipsComplex(points=swe,max_edge_length=60)
simplex_tree = rips_complex.create_simplex_tree(max_dimension=2)
persistence = simplex_tree.persistence()
simplex_tree.persistence_intervals_in_dimension(1)
```

```
plt = gd.plot_persistence_diagram(persistence)
plt.show()
```

This is a typical output we can get. Note the dominant persistence interval in dimension 1:



Again, be careful not to pick up to large range. You do not want to kill your computer! :)

That is all for today. Thank you very much for taking this tutorial. Hope you liked it and that you have learned some new stuff you will be able to use later in your work. If you have any questions please let me know on p.t.dlotko@swansea.ac.uk

Please, do me one favour. Once you have completed the tutorial, please let me know (with anonymous email if you wish):

1. What you liked?
2. What you disliked?
3. What you think can be improved or added to the tutorial?

I will be very grateful for your feedback!

Bibliography

[1] Sliding windows and persistence: an application of topological methods to signal analysis. *Foundations of Computational Mathematics*, 2015.

[2] Jean-Daniel Boissonnat and Clément Maria. The simplex tree: An efficient data structure for general simplicial complexes. *Algorithmica*, 70(3):406–427, November 2014.

[3] P. Bubenik. Statistical topological data analysis using persistence landscapes. *Journal of Machine Learning Research*, 16:77–102, 2015.

[4] J. Harer D. Cohen-Steiner. H. Edelsbrunner. Stability of persistence diagrams. *Discrete and Computational Geometry*, 2007.

[5] Marc Glisse Steve Oudot Frederic Chazal, Vin de Silva. *The Structure and Stability of Persistence Modules.*

[6] Y. Hiraoka G. Kusano, K. Fukumizu. Persistence weighted gaussian kernel for topological data analysis. *arXiv:1601.01741*.

[7] Robert Ghrist. *Elementary Applied Topology*.

[8] T. Emerson E. Hanson M. Kirby F. Motta R. Neville C. Peterson P. Shipman L. Ziegelmeier H. Adams, S. Chepushtanova. Persistence images: A stable vector representation of persistent homology. *arXiv:1507.06217*.

[9] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002 - 544.

[10] U. Bauer R. Kwitt J. Reininghaus, S. Huber. arxiv:1706.03358v3. *A Stable Multi-Scale Kernel for Topological Machine Learning*.

[11] Sebastien Tavenas Jean-Daniel Boissonnat, Karthik C. S. Building efficient and compact data structures for simplicial complexes. *arXiv:1503.07444*.

[12] M. Mrozek K. Mischaikow.

[13] Martina Scolamiero Ran Levi Julian Shillcock Kathryn Hess Lida Kanari, Paweł Dłotko and Henry Markram. A topological representation of branching neuronal morphologies. *Neuroinformatics*, 16, 2018.

[14] R. E. Moore. *Interval Analysis*.

[15] M. Facello P. Fu E. P. Mucke C. Varela N. Akkiraju, H. Edelsbrunner. Alpha shapes: definition and software. *Proc. Internat. Comput. Geom. Software Workshop 1995, Minneapolis*.

[16] P. Dłotko P. Bubenik. A persistence landscapes toolbox for topological statistics. *Journal of Symbolic Computation*, 78:91–114.

[17] J.P. Serre. Homologie singuliere des espaces fibres: Applications. *Annals Math. 54(3) (1951), 425-505*.

[18] R. Ghrist V. de Silva. Homological sensor networks. 54(1).