

Topological Data Analysis

Computation of filtrations and barcodes for some topological spaces.

1. Lancer le code

Deux versions du code sont disponibles. La première, plutôt naïve, utilise une matrice pleine pour représenter l'application frontière. Pour lancer ce code, rendez vous sur le script ReductionAlgorithm.java, au niveau de la fonction main. Vous pouvez alors changer les arguments de ComputeBarcode pour désigner le fichier source qui contient la filtration de l'input, et le fichier dans lequel écrire les barcodes. Vérifiez, si jamais vous exécutez l'algorithme depuis une console, que l'input se trouve au même niveau que votre script .java. Si vous exécutez le code depuis un IDE tel qu'Eclipse, ce dernier devrait automatiquement gérer les dépendances et trouver l'input à la racine du projet.

La seconde version utilise plusieurs techniques d'optimisation, notamment une implémentation de matrice sparse adaptée à la situation. Pour lancer ce code, rendez vous sur le script SparseImplementation.java au niveau de la fonction main. Deux commentaires vous indiquent où placer le fichier d'input ainsi que celui d'output.

Pour générer les différents inputs des espaces classiques il suffit de lancer la fonction main de ClassicalSpaces. Celle-ci ne prend pas d'argument et génère automatiquement les fichiers ball_i.txt, sphere_i.txt (pour $i = 0 \dots 10$), torus.txt, mobius_band.txt, klein_bottle.txt et projective_plane.txt qui stocke les triangulations classiques. Ils sont déjà enregistrés mais vous pouvez évidemment les régénérer.

2. Boundary Matrix

Dans cette partie et les suivantes, nous distinguerons les algorithmes écrits pour la version dense de la matrice de frontière de ceux écrits pour l'implémentation sparse de celle-ci que l'on détaillera en section 6. L'avantage de la première est qu'elle fournit en retour un détail des opérations effectuées : tri de la filtration, boundary matrix et enfin boundary matrix réduite. Celui de la seconde est de pouvoir gérer des fichiers plus lourds.

La classe BoundaryMatrix est dotée d'un champ *sortedsimplices*, à savoir le détail du simplicial complex, d'un constructeur qui prend un entrée une filtration et trie les éléments de celle-ci selon la filtration, et enfin d'une méthode principale *creatematrix()* qui génère la boundary matrix. À ce stade, remarquons qu'il a été nécessaire de créer une class *Simplex* qui implémente l'interface *Comparable*.

La méthode *creatematrix()* vise à remplir une matrice *result* en maintenant l'indice de la colonne de la matrice associée à chaque simplex dans un Hashmap *indicesimplex* :

```

int[][] result = new int[n][n];

HashMap<TreeSet<Integer>, Integer> indicesimplex = new HashMap<TreeSet<Integer>, Integer>();

```

Pour tout simplex de dimension au moins une, après lui avoir dédié une nouvelle colonne, on calcule linéairement en sa taille (ou plutôt en $O(\text{taille} * \log(\text{taille}))$ via l'usage du TreeSet) chacune de ces frontières, déjà présentes dans *indicesimplex* (une filtration impose qu'une face d'un simplex apparaît toujours avant le simplex dans une filtration, en ajoutant à la relation d'ordre une condition sur les dimensions), puis on en déduit comment remplir la colonne courante.

3. Réduction par pivot de Gauss

La partie réduction de la matrice peut être effectuée via l'appel à la fonction *PivotGauss* du script *ReductionAlgorithm.java*. Notons que Deux fonctions utilitaires, *low* et *GetColumn*, précèdent *PivotGauss* dans ce fichier. Ces fonctions servent respectivement, étant donné une colonne à calculer son indice de composante non nulle maximale, et étant donné une matrice de récupérer une colonne d'indice donné.

La réduction procède comme suit :

- On initialise une matrice *result* qui contiendra le résultat.
- Pour chaque indice de colonne, on récupère la colonne correspondante dans la *BoundaryMatrix*.
- Tant que celle-ci est non nulle et qu'il existe une colonne de *result* d'indice plus faible, on récupère cette autre colonne et on l'ajoute à la colonne courante (ce qui revient à effectuer une opération xor composante par composante modulo 2).
- On rajoute cette colonne modifiée à *result*.

4. Complexité

Notons m le nombre de simplexes de la filtration. Les appels à *low*, *GetColumn* et l'addition de deux colonnes sont des opérations linéaires en m . Notons qu'une récurrence rapide montre que lorsque l'algorithme finit la réduction de la colonne courante, l'ensemble des colonnes déjà traité admet des pivots distincts deux à deux. Cette remarque engendre que la complexité totale de *PivotGauss* est cubique en m . En effet :

- Pour une colonne donnée, il y a un nombre inférieur à m de colonnes potentielles pour réduire la colonne courante.
- Une opération entre une colonne et une autre a lieu au plus une fois au cours du déroulement l'algorithme (en effet, dès lors que la colonne courante est réduite via une autre colonne, son pivot diminue et on ne réutilisera jamais l'autre colonne pour réduire à nouveau la colonne courante).
- Ainsi pour chaque colonne, on effectue au plus m examens des colonnes précédentes, durant lesquels un appel à *low* et à *GetColumn* a lieu.

5. Barcode

Également dans la classe *ReductionAlgorithm.java*, la fonction *ComputeBarcode* permet d'obtenir la persistance associée à Simplicial Complex. En réalité, cette fonction encapsule les précédentes :

- elle lit la filtration dans un fichier source, *Input*, via la class *ReadFiltration.java*.
- elle trie les simplexes et calcule la matrice de frontière.
- elle effectue la réduction via *PivotGauss*.
- elle génère une liste de Barcodes.
- elle l'écrit dans un fichier destination : *Output*

```
public static void ComputeBarcode(String Input, String Output)
```

Au passage, mentionnons la nécessité d'une classe *Barcode.java* auxiliaire pour remplir la liste des barcodes:

```
public class Barcode {  
    int dim;  
    float birth;  
    float death;
```

Par défaut la valeur -1 pour *death* signifie une mort non programmé (la composante connexe n'est jamais tuée, autrement dit le nombre betty associée à *dim* augmente), tandis que -2 signifie l'absence de significativité de ce Barcode (une colonne nulle tuée par une colonne ultérieure qui dupliquera une information).

6. Optimization

SparseImplementation.java récapitule les fonctions qui précédent de façon plus efficace :

```
public class SparseImplementation {  
    // 5 champs:  
    // -BoundarySparse retient pour chaque colonne(key) les indices non nuls(value).  
    // -IndexToSimplex permet d'accéder à partir d'une colonne(key) au simplex correspondant(value).  
    // -SimplexToIndex effectue l'opération réciproque.  
    // -Pivots retient pour chaque colonne(key) non nulle, l'indice non nul le plus élevé.  
    // -Barcodes permet de retenir les résultats de naissance(key) et de mort(value) lus sur la matrice réduite.  
  
    HashMap<Integer, TreeSet<Integer>> BoundarySparse;  
    HashMap<Integer, Simplex> IndexToSimplex;  
    HashMap<TreeSet<Integer>, Integer> SimplexToIndex;  
    HashMap<Integer, Integer> Pivots;  
    HashMap<Integer, Integer> Barcodes;
```

Outre la représentation sparse de la matrice de frontière, l'accès aux barcodes et aux pivots est très rapide grâce aux structures de *HashMap*. Considérons tout d'abord la fonction auxiliaire suivante :

```
PivotGauss(TreeSet<Integer> CurrentColumn, TreeSet<Integer> PreviousColumn)
```

chargée de réduire *CurrentColumn* via *PreviousColumn* en modifiant les indices de *CurrentColumn*. Si l'on note n et m les longueurs de ces deux colonnes, remarquons que l'opération de réduction est ici symétrique, -il s'agit d'un XOR des deux colonnes-, si bien que son coût est en O(mlog(n)), où le logarithme provient de l'appel à la méthode *remove* la structure de *TreeSet*, mais pourrait s'effectuer en O(nlogm) en inversant le rôle des colonnes. On voit là une légère amélioration théorique possible de la complexité.

La fonction *Reduction*, étant donnée une colonne, appelle *PivotGauss* autant de fois que nécessaire pour réduire la colonne courante, comme dans la section 3. Ici néanmoins, puisque les Barcodes sont contenus dans un *HashMap*, ces derniers sont calculés en même temps que la matrice est réduite.

La fonction principale est ici le constructeur de la classe, qui initialise les champs, et prend en entrée un ensemble de simplexes. La fonction construit alors la *boundaryMatrix* en la réduisant simultanément ; quand une colonne est ajoutée, elle est réduite via les colonnes précédentes, et on retient le barcode éventuellement engendré.

Une dernière fonction utilitaire,

```
public void WriteBarcode(String File)
```

utilise le champ *Barcodes* afin d'écrire dans *File* l'output au format désiré.

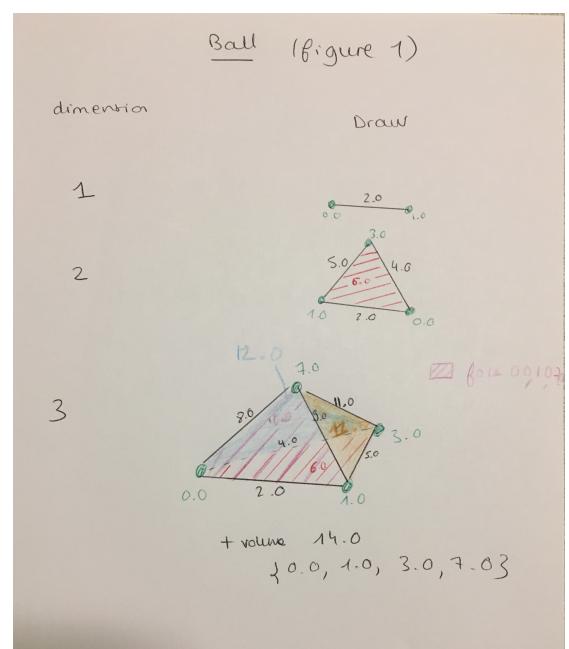
Pour générer les différents inputs des espaces classiques il suffit de lancer la fonction main de *ClassicalSpaces*. Celle-ci ne prend pas d'argument et génère automatiquement les fichiers ball_i.txt, sphere_i.txt (pour i =0...10), torus.txt, mobius_band.txt, klein_bottle.txt et projective_plane.txt qui stocke les triangulations classiques. Ils sont déjà enregistrés mais vous pouvez évidemment les régénérer.

7. ClassicalSpaces

La class *ClassicalSpaces.java* génère les filtrations des espaces classiques. Tout d'abord présentons la façon dont générer le *Vector<Simplex>* représentatif de la triangulation.

a. BALLS

Balls(int d) qui génère un *Vector<Simplex>* représentatif de la boule de dim d grâce à l'appel à la fonction récursive *RecursiveBall(Vector<Simplex> CurrentFiltration, int i)* qui permet d'ajouter une dimension. Elle ajoute le simplex i en le reliant à tous les simplexes existants. Ceci est représenté sur la figure 1.

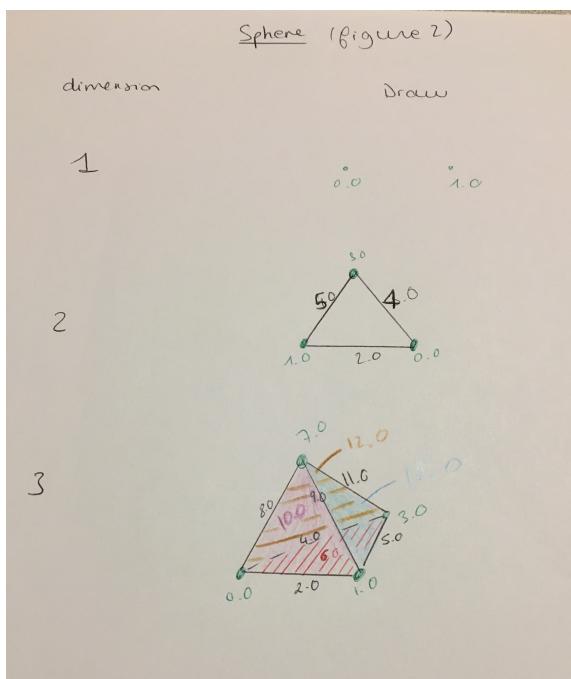


Cette fonction respecte bien la règle de la fonction valeur puisque à chaque augmentation de la dimension, on considère d'abord le point, puis les arrêtes qu'il construit avec les points précédents, puis les surfaces et on incrémente la valeur de f à chaque fois (elle vaut la taille courante de la triangulation à chaque ajout). Il s'agit d'un invariant conservé tout au long de la fonction.

b. SPHERES

La sphère de dimension d est obtenue dans la fonction $Sphere(int d)$ à partir de la boule de même dimension en retirant le simplexe de plus grande dimension. Ceci permet toujours de conserver la propriété de la fonction de triangulation puisque le sommet de plus grande dimension est le dernier à la liste et ne change donc rien aux fonctions valeurs des simplexes précédents.

Les triangulations désirées sont représentées sur la figure 2.

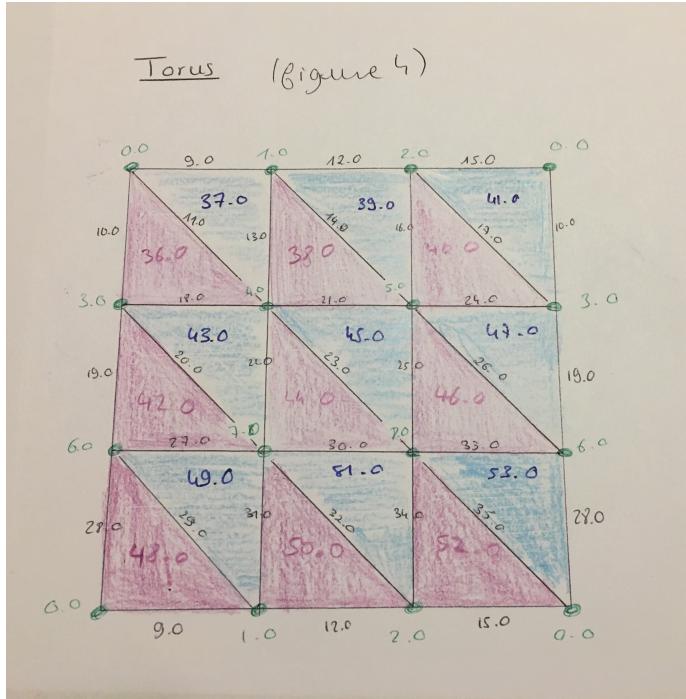


c. MOBIUSBAND, TORUS ET KLEIN BOTTLE

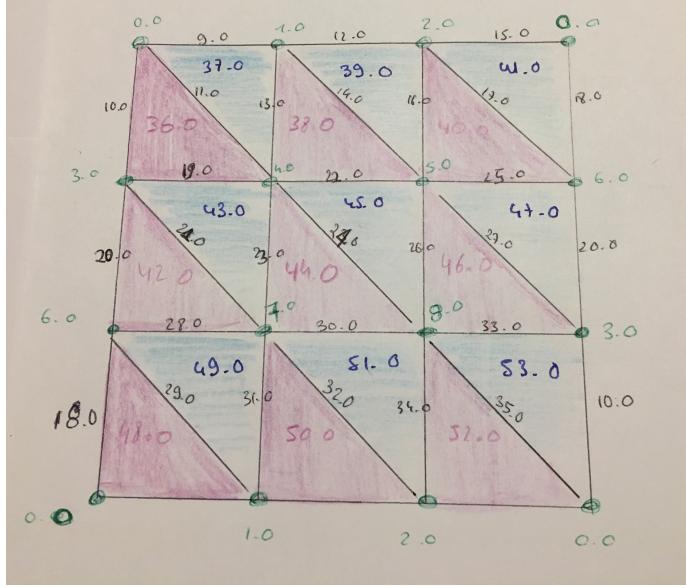
Ces 3 triangulations sont générées selon le mécanisme suivant :

- Une fonction $MatrixMobiusBand()$, $MatrixTorus()$, $MatrixKleinBottle()$ qui retourne la matrice représentative de la triangulation désirée. Ces matrices sont en accord avec les triangulations des figures 3, 4, 5.

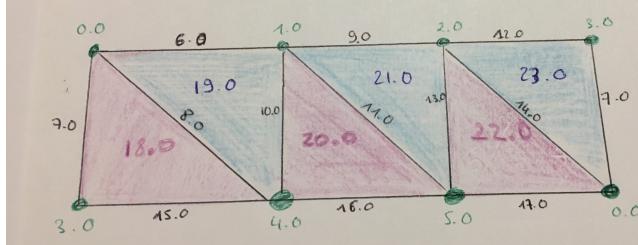
Torus (Figure 4)



Klein Bottle (figure 5)



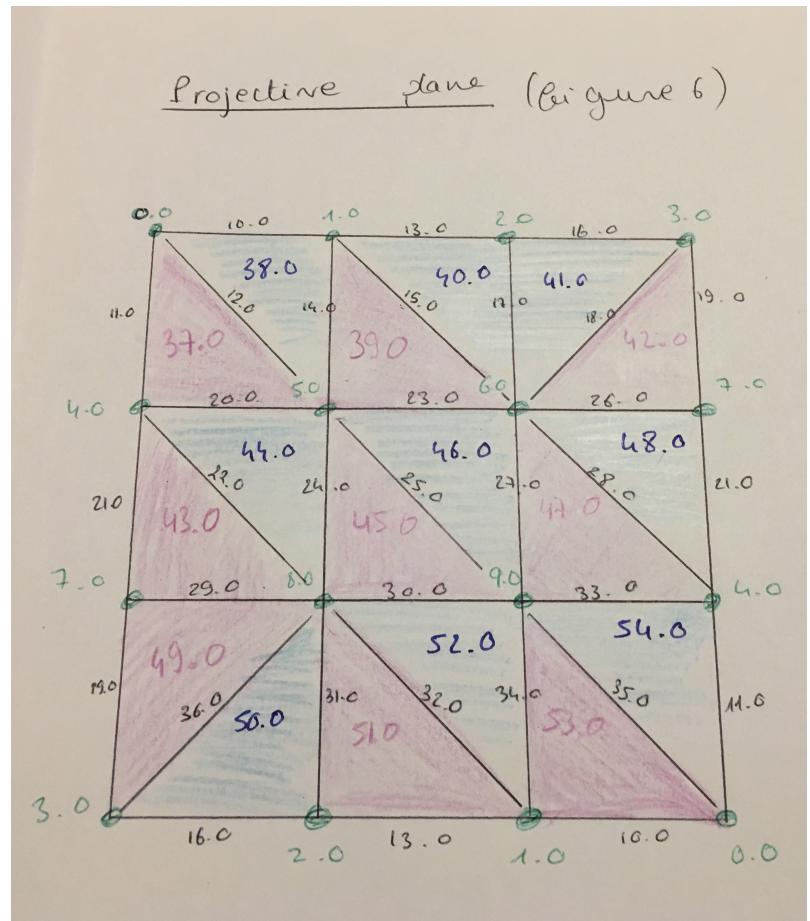
Nobius Band (figure 3)



- *MatrixToFiltration(int[][] matrix)* qui à partir de cette matrice crée le *Vector<Simplex>* représentatif de la triangulation. Elle insère tous d'abord tous les sommets de dimension 0 en allant récupérer $matrix[i][j]$ pour indiquer verts. La fonction valeur est elle initialisée à 0 puis incrémentée à chaque ajout de simplex. Elle va ensuite chercher toutes les arrêtes de dimension 1 qui sont les voisins vers le bas, vers la droite, et vers le bas-droites. Enfin, elle récupère les surfaces qui peuvent être de 2 types (en violet et bleu sur les figures 3,4,5).
- N.B. : cette fonction n'ajoute jamais plusieurs fois un même simplexe puisqu'elle retient dans le *HashSet<TreeSet<Integer>> Triangulated* tous les simplexes déjà insérés dans la triangulation. Enfin, cette fonction permet bien de respecter la règle de la triangulation puisqu'on ne créer jamais une surface avant les deux simplexes qu'elle contient. En effet, on fait d'abord tous les sommets, puis toutes les arrêtes, puis toutes les surfaces.

d. PROJECTIVE PLANE

La forme projective plane ne peut pas être générée selon le même modèle que précédemment puisque toutes les diagonales de la triangulation ne sont pas orientées selon le même sens (voire coin en haut à droite et en bas à gauche sur la figure 6). Ceci implique la prise en compte de plusieurs cas frontières.



Il y a encore une fois 2 étapes :

- Une fonction *MatrixProjectivePlane()* qui retourne la matrice représentative de la triangulation désirée. Cette matrice est en accord avec la triangulation suivante :
- Les cas frontières évoqués plutôt nous incite à utiliser une fonction spécifique *MatrixToFiltrationProjectivePlane(int[][] matrix)*. Elle est très similaire à la fonction *MatrixToFiltration(int[][] matrix)* avec l'utilisation d'un *HashSet<TreeSet<Integer>> Triangulated* et l'ajout progressif des sommets, puis

des arrêtes puis des surfaces. Cependant, elle prend garde aux 2 cas limite : le carré haut-droite et le carré bas-gauche, afin de prendre en compte la présence du 9^{ème} sommet (par rapport au 3 cas précédents qui n'avaient que 8 sommets) et les arrêtes / faces engendrées.

e. EXPORTFILTRATION ET MAIN

- Ensuite, nous utilisons une fonction commune utile ExportFiltration(String FileName, Vector<Simplex> Filtration) qui écrit dans un fichier FileName sous la forme désirée par ReadFiltration la Filtration placée en argument.
- **Fonction main :**
Appelle l'ensemble des fonctions de filtration des différentes formes: *Ball(d)* et *Sphere (d)* pour $d < 11$, *MatrixToFiltration(MatrixMobiusBand())*, *MatrixToFiltration(MatrixTorus())*, *MatrixToFiltration(MatrixKleinBottle ())*, *MatrixToFiltrationProjectivePlane(MatrixProjectivePlane())*. Puis cette filtration est utilisée en argument de la fonction *ExportFiltration* qui se charge d'écrire dans un fichier au nom explicite les représentations lisibles pour Read Filtration.

8. Barcodes of classical filtrations

En exécutant notre *SparseImplementation*, on obtient des filtrations correctes au sens où l'on retrouve les nombres de Betti des homologies connues. On a néanmoins un doute pour les formes KleinBottle et ProjectivePlane. Nous émettons l'hypothèse que les valeurs trouvées sont vérifiées dans le corps $\mathbb{Z}/2\mathbb{Z}$.

Ball_i pour $i = 0 \dots 10$: $\beta_0 = 1$ et $\beta_1 = \beta_2 = \dots = \beta_{10} = 0$
Sphere_i pour $i = 0 \dots 10$: $\beta_0 = 1$ et $\beta_j = 0$ (si $j \neq i-1$) et $\beta_{i-1} = 1$

	MobiusBand	Torus	KleinBottle	ProjectivePlane
β_0	1	1	1	1
β_1	1	2	2	1
β_2	0	1	1	1

9. Timings of filtrations

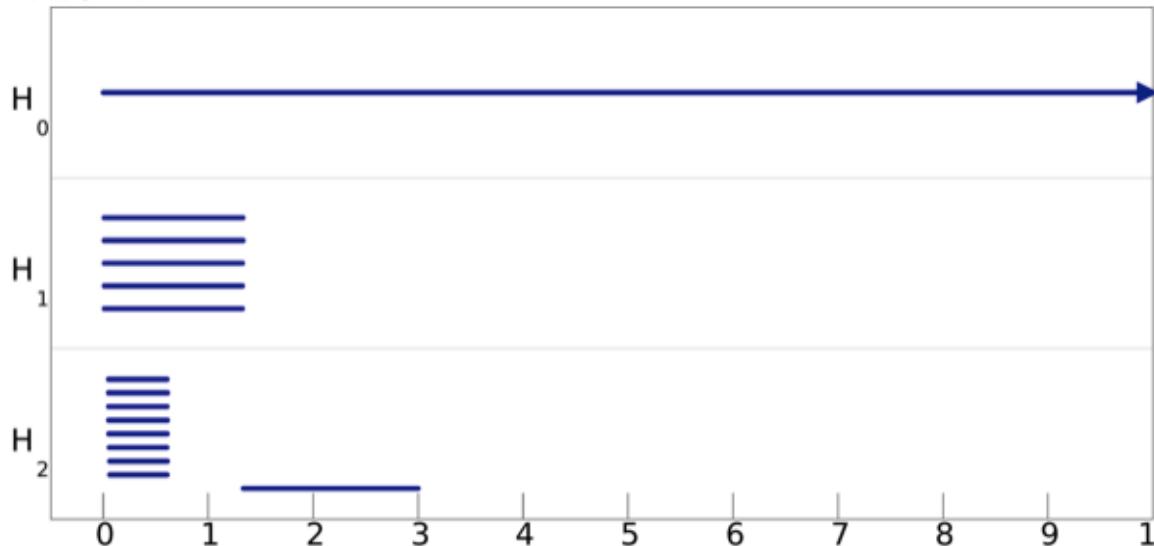
Dans ce tableau on ne prend pas en compte le temps d'écriture dans les fichiers outputs mais simples le temps de création et de réduction.

Filtration	Number of Simplices	ReadFiltration	Sort	Reduction	Total
A	428 643	2.3075	0.02851	1.0324	3.3700
B	108 161	0.8228	0.01001	1.7168	2.5496
C	180 347	1.1521	0.00410	4.9945	6.1507
D	2 716 431	1.9071	0.30417		

Pour la filtration 6 nous obtenons malheureusement une *Garbage Collector Error*. Nous avons essayé plusieurs techniques afin d'y pallier comme l'option XMX2G. Cependant, rien n'a fonctionné. Afin d'améliorer notre algorithme, il s'agirait donc maintenant de ne plus simplement chercher à optimiser le temps mais également ne pas être trop gourmand en espace.

10 . Inference of topological Structure from barcodes

filtration B:



Il ne nous est pas apparu de technique déterministe pour inférer la structure topologique d'un jeu de donnée depuis son diagramme de filtration telle que celui ci-dessus. Parmi les quatre filtrations proposées, seule le nombre de Betty associé à la dimension 0 est non nul et vaut 1. Comme tous les exemples sont donnés en dimension 3, il nous est apparu qu'il devait y avoir une ressemblance de structure avec la boule unité. Devenir plus précis est néanmoins complexe.

Pour essayer de visualiser ces structures, nous avons systématiquement imaginé qu'un dataset de points était généré dans une structure particulière puis que ces points étaient reliés entre eux peu à peu. Chaque point détient une certaine influence, représentée par la taille d'une boule qui l'entoure, et qui sera relié en priorité avec un point dont la boule qui l'entoure intersecte la première boule. Lorsque l'on trouve un amas de points, la boule globale qui les entoure est d'autant plus grande que le nombre de points contenus est grand.

Nous voyons la filtration A comme un sample aléatoire de nombreux points sur une sphère en 3D. Générant initialement de nombreuses composantes connexes (augmentant ainsi H0), les points proches finissent par détruire des composantes connexes et simultanément créer de nouveaux segments (augmentant le nombre de barres dans H1). Lorsque les segments sont rassemblés, demeure un trou 2D (au milieu de la sphère en quelque sorte) qui finira par être lui même absorbé pour devenir la boule 3D.

Pour ce qui est de la filtration B, il est possible d'imaginer huit boules tangentes chacune avec trois autres (les placer au sommet d'un cube par exemple). On observe 8 trous de H2 initialement au sein de chaque sphère, et sur une face du cube un trou de H1, soit un total de 5 trous de H1 indépendant. Lorsque les sphères se remplissent selon la filtration, il ne demeure qu'un trou de H2 au milieu du cube, finalement annihilé par le dernier simplex de la filtration.

Nous n'avons pas trouvé la différence fondamentale entre les filtrations C et D. Nous pensons donc qu'il s'agit de structures similaires, proche de celles d'un tore. Des points générés aléatoirement sur le tore 2D engendrent de nombreuses composantes connexes. Peu à peu reliées entre elle, l'intérieur du conduit du tore finit par se remplir, un seul cycle H1 demeurant plus longtemps jusqu'à ce que les boules des amas de points se touchent et finissent par combler l'intérieur du tore, éliminant au passage un trou H2. Demeure alors le trou H1 principal que le tore encercle, et qui finira par être absorbée quand les points seront de plus en plus nombreux et que les boules « diamétralement opposés » finiront par se toucher et combler le trou H1.

Dans chacun des cas, la triangulation définitive est une boule 3D.