Topological Data Analysis
TD4

Joel Tagne, Mathias Ollu

1 Code explanation

Our code is organized in the following fashion:

1.1 Two main classes

First, two main classes, stored in python files of the same name.

One class **Simplex** which stores vertices, dimension and filtration value for each simplex of a set. Vertices are stored as a frozenset to allow to store indexes of sub-simplexes in a dictionnary.

One **SparseBoundaryMatrix** class. This is the central class which stores a matrix as a dictionary of column-row sets which is adapted to the reduction algorithm.

After initialisation, an instance of SparseBoundaryMatrix can be attributed to a filtration via the method **from_simplices**, which creates the corresponding boundary matrix inside the SparseBoundaryMatrix structure. Critically, this method sorts simplices according to the total order specified by **simplex_key**: first the filtration value, then the dimension, and finally the lexical order of the sorted vertices. This is indeed a total order, as the lexical order ensures that two simplices of the same dimension can always be sorted as they necessarily have at least one different vertex.

The boundary matrix can then be reduced in place through the **gaussian elimination** method (explained in more detail in section 2, complexity analysis).

Finally, barcodes can be inferred through the **get_barcode** method which stores and prints barcodes based on the reduced matrix. If the matrix is not reduced yet, is computes the reduction, and then computes barcodes.

SparseBoundaryMatrix also provides auxiliary methods, especially **low**, which computes the lowest non-zero coefficient of a given column in the matrix.

1.2 Test and plot functions

test_plots.py allows to test our code on a few topological structures (question 5-6). They make use of data files in the data repository, especially the filtration files which are generated by the functions in **filtration_generation.py**.

filtration_generation.py computes the following filtrations:

Sphere S^d. Use the boundary complex of the $(d+1)$-simplex: emit $d+2$ vertices at $t = 1$, then all faces of size $2 \dots (d+1)$ with strictly increasing timestamps, omitting the full $(d+1)$-cell. Face-first order holds by construction; orientation is irrelevant over Z_2.

Ball B^d. Take the full d-simplex: vertices at $t = 1$, then every face up to dimension d, including the d-cell. The complex is contractible, so only H_0 persists (useful sanity check for the ∂-reduction pipeline).

Möbius band. Start from a 2×3 strip (6 vertices), add horizontal and vertical edges plus a single "twist" bridge and minimal diagonals, then fill with 2-simplices so that the edge combinatorics encodes the half-twist. Over Z_2 this yields a single H_1 generator.

Torus T^2. Build a 3×3 periodic grid: add wrap-around horizontal and vertical edges and triangulate each square (diagonals), then insert 2-simplices. Periodic identifications along both axes produce two independent H_1 generators and one H_2 class.

Klein bottle. Use a 3×3 grid with periodicity in one direction and a twisted identification in the other: across the twisted boundary, reindex vertices by a reflection before adding edges and triangles. With Z_2 coefficients this realizes two H_1 generators and a top class.

Projective plane RP^2. "Star + pentagon" scheme: one center joined to a 5-cycle, plus selected chords to implement antipodal pairing, then 10 triangles to close the complex. The filtration respects face-first order and reproduces $\beta_0 = 1$, $\beta_1 = 1$, $\beta_2 = 1$ over Z_2.

1.3 Performance evaluation

evaluate_perf.py allows to calculate computation times of the algorithms of the SparseBoundaryMatrix class on several datasets. Its plot option provides a graphical representation of the computation time as a function of the number of simplices (cf figure 2).

1.4 Examples

Finally, a simple example of usage of the functions is provided in the **example.py** file, which applies the algorithms to the simple filtration example in TD4.

2 Complexity analysis (question 3)

The reduction algorithm relies on three main mechanisms which allow it to be in $O(m^3)$, for m the number of simplices, i.e. the number of columns in the boundary matrix B.

First, B is implemented as a sparse matrix. Columns are stored as a dictionary, whith column indices as keys and corresponding row-coefficient sets as values. This allows to compute $low()$ values in $O(K)$ for K the maximum number of non zero coefficient over all columns of B. B is sub-triangular, hence $K < \frac{m}{2} - 1$. For a sparse matrix, this can be approximated by $O(1)$.

Second, the reduction algorithm stores low values in a dictionary as it iterates through the columns of B. This allows to check if a reduction of column j is needed in $O(1)$ time.

Third, the operation on columns on line 11 of algo 1 leverages the $Z/2Z$ space. As coefficients are only ones or zeros, to nullify the common pivot of columns i and j, we simply have to add one column to the other, which is equivalent for per column row sets to taking the symmetric difference of the row sets. The symmetric difference operation on sets X, Y has worse case complexity $O(length(X) + length(Y))$ when X, Y are completely different. In our case of a sparse matrix, each column's row set can be approximate to have a constant length (non dependent on m). Hence, the symmetric difference operation has best case complexity $O(O(1) + O(1)) = O(1)$.

We can now calculate the worst case complexity of our algorithm:

- We iterate through the m columns
- We calculate the low value of column j as many times as necessary, until it shares no low value with a previous column. As B is sub-triangular, there are in the worst case $\frac{m}{2} - 1$ low values, and j columns that could share the same low value. Hence, we iterate a maximum of $min(\frac{m}{2} - 1, j)$ times over column j, which approximates to $O(m)$ time.
- For each *while*-iteration over column j, we compute the lows value of the column in $O(m)$ time, and take the symmetric difference of two columns of the matrix in $O(m + m) = O(m)$ in the worst case.

Hence the worst case complexity is $O(m^3)$.

In the best case, the matrix is sparse. Hence, all operations take $O(1)$ time except for the iteration through the columns of B, making the best case complexity $O(m)$.

Algorithm 1: Reduction of the boundary matrix

Input: sparse boundary matrix B, number of columns in B $\mathrm{N}_c olumns$
Output: reduced boundary matrix B

```
lows ← emptydictionnary
for j from 1 to N_Columns do
    equal_Lows ← True
    while equal_Lows do
        low_j ← B.low(j)
        if low_j = −1 then
            break
        end
        if low_j ∈ lows.keys then
            B.column[j]
              ← symm_diff(B.column[j], B.column[low[j]])end
            if low_j ∉ lows.key then
                add (low_j, j) to dictionnary lows equalLows False
            end
        end
        end
        return B
```

Algorithm 2: Low function

Input: sparse matrix B, column j
Output: lowest non null coefficient's index in column j

```
if (j not in columns of B) OR (column j of B empty) then
    return −1
end
return argmax(column j of B)
```

3 Tests on standard topological structures

Tests on classical topological structure confirm that our algorithm is correct. Figure 1 plots the computed barcode and indicates the theoretical persistent modules for each topological structure.

The persistence barcodes correctly reproduce the expected topological invariants of each complex.

For the sphere S^2, only one long H_0 and one H_2 class remain, while H_1 loops vanish — consistent with a simply connected closed surface.
The torus T^2 shows two persistent H_1 classes and one H_2 component, matching

its Betti numbers $(1, 2, 1)$.
The 2-ball B^2 is contractible, hence only one infinite H_0 bar.
The Klein bottle and Möbius strip both display non-orientable features: one or two H_1 generators but no orientable H_2 surface for Möbius ($\beta_2 = 0$), while the Klein bottle keeps one top-dimensional class over Z_2.
Finally, the projective plane RP^2 combines one H_0, one H_1, and one H_2 class, confirming the expected homology pattern $(1, 1, 1)$.

Overall, the filtrations and reduction algorithm produce consistent persistent homology signatures across all tested manifolds.

The filtrations for these structures are displayed in the appendix (figure 3). To display the computed betties, execute the python script in **test_and_plot_barcodes.py**.

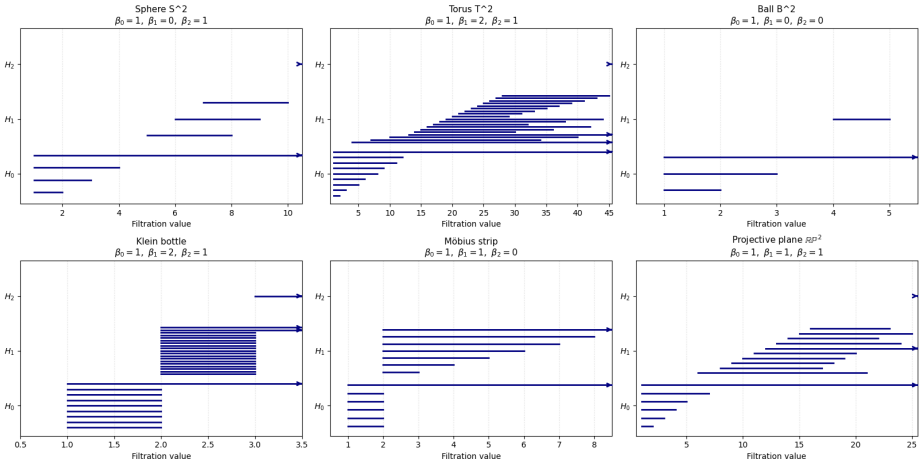

Figure 1: Barcodes of standard topological structures

4 Performance measure

We have tested our algorithms on the filtration datasets A to D, and have recorded for each the full computation time.
Figure 2 displays these results, confirming that computation time is approximately linear in the number of simplices. This indicates that the boundary matrix is indeed sparse.

5 Topological structure analysis

When restricting to H_1, H_2 and H_2, we can respectively attribute their cardinal to the number of related components, the holes and the cavities of the underlying

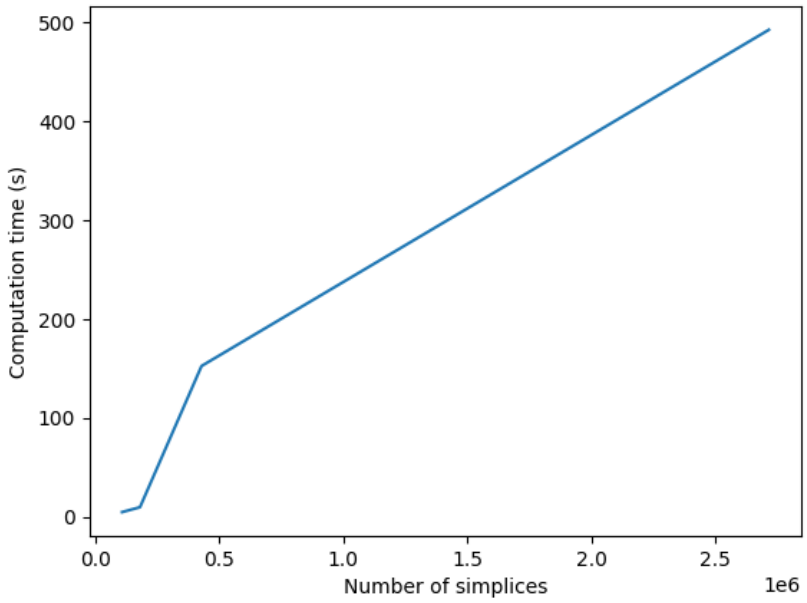

Figure 2: Computation time in function of simplices number

topological structure. This allows to interpretate the barcodes for filtrations A to D in the following way:

- **filtration A:** The structure clearly has a unique related component, which agglomerates early (i.e. for low filtration values). Multiple hole form as soon as the related component comes together, eventually killed by a cavity. This might resemble to a sphere S^2: the multiple holes correspond to the forming spheric cavity, until the filtration value becomes so big that the cavity is filled (e.g. a ball), hence the death of the cavity for filtration value -5.

- **filtration B:** It is unclear whether the multiple bars for H_1 and H_2 are noise or not. However, their low persistence and their high number might indicate that they are indeed noise: we do not know of a topological structure with betties higher than 2 or 3. However, the last bar for H_2 (between 1.3 and 3) is alone, and persists longer than the other bars in H_1 and $H-2$. This could hence be a sphere, although the presence of multiple low-persistence bars for H_2 between 0 and 0.6 are difficult to interpretate if it were a circle.

- **filtration C:** This barcode resembles a circle, with betty-1 and 2 equal to one and the only hole being filled for a high filtration value, i.e. when

filtration radius' fill the circle.

- **filtration D:** 4 large bars appear quite clearly, allowing to infer a 1-2-1 homology, i.e. that of a torus. Given that D has a very high number of points, it makes sense that the single bar for H_1 forms rapidly: all points sit close to each other. The fact that a hole appears juste after the unique related component forms indicates that the structure has a circular shape. The second bar can be attributed to the hole alongside the torus's tube: once it dies, a cavity appears, which is exactly the interior of the tube.

6 Appendix

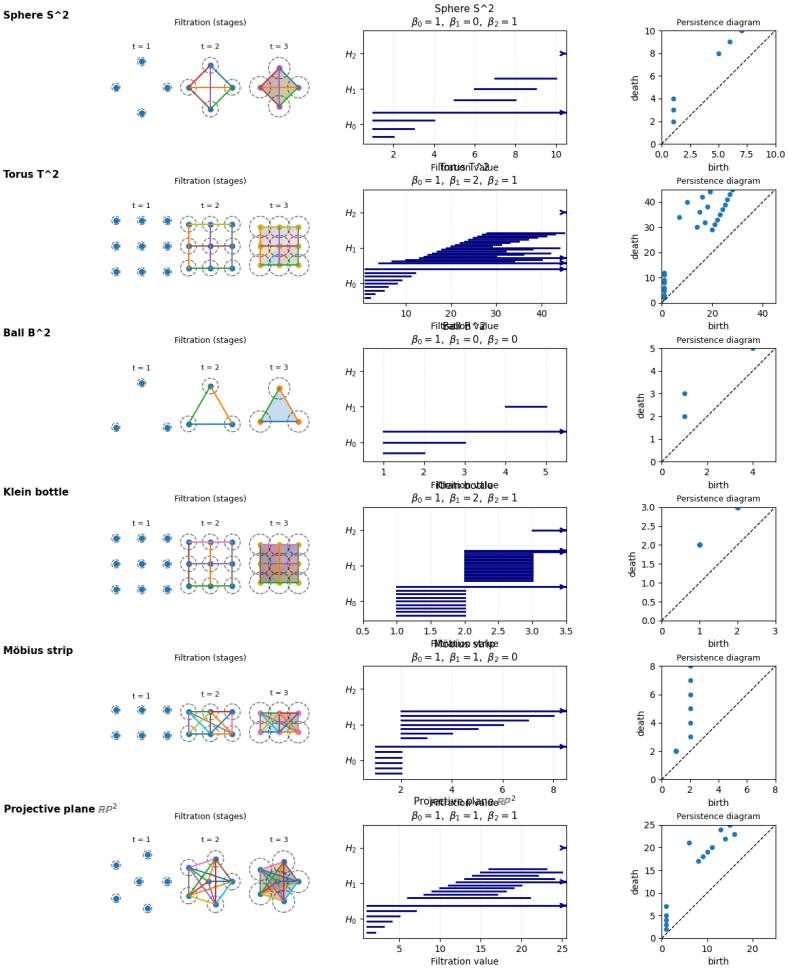

Figure 3: Filtrations for standard topological structures