I - src/geometry.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 14-09-2024 13:55:46
 *  Last modified : 25-04-2025 21:40:59
 *  File : geometry.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
 #include <math.h>

 #include "geometry.h"
 #include "misc.h"

 const int DIM = 2;

 // Renvoie la distance euclidienne entre deux points
 float dist_euclidean(int p1, int p2, PointCloud *X){
     return sqrt(pow(X->pts[p2].x - X->pts[p1].x, 2)
+
             pow(X->pts[p2].y - X->pts[p1].y, 2));
 }

 // Renvoie la distance entre deux points
 // Cette fonction sert d'interface pour que l'on
puisse facilement
 // la modifier lors de l'application de notre
programme à un sujet
 float dist(int p1, int p2, PointCloud *X){
     // return dist_euclidean(p1, p2, X);

     return X->dist[p1][p2];
 }

 //////////////////////
 //      POINTS      //
 //////////////////////

 // Affiche un point
 void pointPrint(Point p){
     printf("(%f, %f)", p.x, p.y);
 }

 // Initialise un nuage de points
 PointCloud *pointCloudInit(int size){
     PointCloud *pointCloud =
malloc(sizeof(PointCloud));
     pointCloud->pts = malloc(size * sizeof(Point));
     pointCloud->weights = malloc(size *
sizeof(float));
     pointCloud->dist = malloc(size * sizeof(float*));
     for(int i=0; i<size; i++) pointCloud->dist[i] =
calloc(size, sizeof(float));
     pointCloud->size = size;
     return pointCloud;
 };

 // Teste si deux points sont égaux
 bool pointAreEqual(Point p1, Point p2){
     return p1.x == p2.x && p1.y == p2.y;
 };

 // Libère un nuage de points
 void pointCloudFree(PointCloud *X){
     free(X->pts);
     free(X->weights);
     for(int i=0; i<X->size; i++) free(X->dist[i]);
     free(X->dist);
     free(X);
 };

 PointCloud *pointCloudLoad(char *filename, char
```

```
*dist_filename){
67      FILE *file = fopen(filename, "r");
68      if(file == NULL){
69          print_err("Erreur: fichier points
introuvable\n");
70          exit(1);
71      }
72
73      int size;
74      fscanf(file, "%d", &size);
75      PointCloud *pointCloud = pointCloudInit(size);
76
77      for(int i = 0; i < size; i++){
78          float weigth;
79          fscanf(file, "%f %f %f %*[^\n]",
&(pointCloud->pts[i].x),
80              &(pointCloud->pts[i].y), &weigth);
81          pointCloud->weights[i] = weigth;
82
83      }
84      fclose(file);
85
86      // Si un fichier distance a été fourni
87      if (dist_filename != NULL){
88          FILE *dist_file = fopen(dist_filename, "r");
89          if(file == NULL){
90              print_err("Erreur: fichier distances
introuvable\n");
91          }
92
93          fscanf(dist_file, "%d", &size);
94          int poubelle;
95          for(int i = 0; i<size; i++){
96              for(int j=i+1; j<size; j++){
97                  float weigth;
98                  fscanf(dist_file, "%d %d %f",
&poubelle, &poubelle, &weigth);
99                  pointCloud->dist[i][j] = weigth;
100             }
101         }
102
103         fclose(dist_file);
104
105         for(int i=0; i<size;i++)
106             for(int j=0; j<i+1; j++)
107                 pointCloud->dist[i][j] = pointCloud-
>dist[j][i];
108
109     } else { // Sinon on utilise les distances
euclidiennes
110         printf("Remplissage par des distances
euclidiennes\n");
111         for(int i = 0; i<size; i++){
112             for(int j=0; j<size; j++){
113                 pointCloud->dist[i][j] =
dist_euclidean(i, j, pointCloud);
114             }
115         }
116     }
117
118     return pointCloud;
119 };
120
121 void pointCloudPrint(PointCloud *X){
122     for(int i=0; i<X->size; i++){
123         printf("Point %d : ", i);
124         pointPrint(X->pts[i]);
125         printf(" de poids %.2f\n", X->weights[i]);
126     }
127     printf("Distances :\n");
128     for(int i=0; i<X->size; i++){
129         for(int j=i+1; j<X->size; j++)
130             printf("%d -- %.3f --> %d\n", i, dist(i,
j, X), j);
```

```
    }
}

//////////////////////
//     Simplex      //
//////////////////////

int compare(const void *a, const void *b){
    return *(int *)a - *(int *)b;
}

// Initialise un simplexe
Simplex *simplexInit(int i, int j, int k){
    Simplex *s = malloc(sizeof(Simplex));
    // Respect de l'ordre lexicographique
    int tab[3] = {i, j, k};
    qsort(tab, 3, sizeof(int), compare);

    s->i = tab[0];
    s->j = tab[1];
    s->k = tab[2];
    return s;
}

// Renvoie l'identifiant associé à un simplexe
int simplexId(Simplex *s, int n){
    return (s->i+1) + (n+1) * (s->j+1) + (n+1) *
(n+1) * (s->k+1);
}

// Renvoie le simplexe associé à un identifiant
Simplex simplexFromId(int id, int n){
    Simplex s;
    s.i = (id % (n+1)) - 1;
    s.j = ((id / (n+1)) % (n+1)) - 1;
    s.k = (id / ((n+1)*(n+1))) - 1;
    return s;
}

// Libère un simplexe
void simplexFree(Simplex *s){
    free(s);
}

// Renvoie le maximum de simplex possible pour un
nuage de points de taille n
unsigned long long simplexMax(int n){
    return (n+1)*(n+1)*(n+1);
}

// Affiche un simplexe
void simplexPrint(Simplex *s){
    printf("{%d, %d, %d}", s->i, s->j, s->k);
}

// Renvoie la dimension d'un simplexe
int dimSimplex(Simplex *s){
    if (s->i != -1){
        return 2;
    } else if (s->j != -1){
        return 1;
    } else {
        return 0;
    }
}

// Renvoie vrai si s1 est une face de s2, false
sinon
bool isFaceOf(Simplex *s1, Simplex *s2){
    // Si les dimensions ne sont pas cohérentes
    if (dimSimplex(s1) != dimSimplex(s2) - 1){
        return false;
    }

```

```
    // Si s1 est un point
    if (s1->i == -1 && s1->j == -1){
        if (s1->k == s2->i || s1->k == s2->j || s1->k == s2->k){
            return true;
        }
    }

    // Si s1 est une arête
    if (s1->j != -1){
        if (s1->j == s2->i && s1->k == s2->j){
            return true;
        } else if (s1->j == s2->i && s1->k == s2->k){
            return true;
        } else if (s1->j == s2->j && s1->k == s2->k){
            return true;
        } else {
            return false;
        }
    }

    // Sinon on est dans un cas impossible
    return false;
}

//////////////////////
// Simplical Complex //
//////////////////////

// Initialise un complexe simplicial
SimComplex *simComplexInit(unsigned long long n){
    SimComplex *cmpx = malloc(sizeof(SimComplex));
    cmpx->simplices = malloc(n * sizeof(bool));
    for(unsigned long long i = 0; i<n; i++){
        cmpx->simplices[i] = false;
    }
    cmpx->size = 0;
    return cmpx;
}

// Libère un complexe simplicial
void simComplexFree(SimComplex *cmpx){
    free(cmpx->simplices);
    free(cmpx);
}

// Teste si un simplex est dans un complexe simplicial
bool simComplexContains(SimComplex *cmpx, Simplex *s, int n){
    return cmpx->simplices[simplexId(s, n)];
}

// Insère un simplex dans un complexe simplicial
void simComplexInsert(SimComplex *cmpx, Simplex *s, int n){
    cmpx->simplices[simplexId(s, n)] = true;
    cmpx->size++;
}

//////////////////////
//     Filtration    //
//////////////////////

// Initialise une filtration
Filtration *filtrationInit(unsigned long long size){
    Filtration *filt = malloc(sizeof(Filtration));
    filt->filt = malloc(size * sizeof(int));
    filt->nums = malloc(size * sizeof(unsigned long long));
    for(unsigned long long i = 0; i<size; i++){
        filt->filt[i] = -1;
```

```
269         filt->nums[i] = -1;
270     }
271     filt->max_name = 0;
272     filt->size = size;
273     return filt;
274 }
275
276 // Libère une filtration
277 void filtrationFree(Filtration *filtration){
278     free(filtration->nums);
279     free(filtration->filt);
280     free(filtration);
281 }
282
283 // Ajoute un simplexe à une filtration
284 // n est le nombre de points de l'espace
285 // k est le l'identifiant du complexe dans lequel s
apparait en premier
286 // num est le numéro utilisé pour l'ordre total
287 void filtrationInsert(Filtration *filtration,
Simplex *s, int n, int k,
288   unsigned long long num){
289     int id = simplexId(s, n);
290     filtration->filt[id] = k;
291     filtration->nums[id] = num;
292
293     // Mise à jour du nom maximal
294     if (num+1 > filtration->max_name) filtration-
>max_name = num+1;
295 }
296
297 // Teste si un simplexe est dans une filtration
298 bool filtrationContains(Filtration *filtration,
Simplex *s, int n){
299     return filtration->filt[simplexId(s, n)]!=-1;
300 }
301
```

```
302 // Affiche une filtration, l'option sorted trie
selon les noms de simplexes,
303 // n'a de sens et ne fonctionne que lorsque la
fonction de nommage est injective
304 void filtrationPrint(Filtration *filt, int n, bool
sorted){
305     int *sortByName = calloc(simplexMax(n),
sizeof(int));
306     if (sorted){
307         for(int i=0; i<filt->size; i++){
308             if (filt->filt[i] != -1){
309                 sortByName[filt->nums[i]] = i;
310             }
311         }
312     } else {
313         for(int i=0; i<filt->size; i++)
314             sortByName[i] = i;
315     }
316
317     for(int r = 0; r<filt->size; r++){
318         if (filt->filt[sortByName[r]] != -1){
319             Simplex s = simplexFromId(sortByName[r],
n);
320             printf("Simplex %lld (dans K_%d): ",
filt->nums[sortByName[r]],
321                 filt->filt[sortByName[r]]);
322             simplexPrint(&s);
323             printf("\n");
324         }
325     }
326
327     printf("Nom max : %lld\n", filt->max_name);
328 }
329
330 // Renvoie le tableau des identifiants des simplexes
dans la filtration
331 int *reverseIdAndSimplex(Filtration *filt){
```

```
332     int *reversed = malloc(filt->max_name*sizeof(int));
333     for(int i=0; i<filt->size; i++){
334         if(filt->filt[i] != -1)
335             reversed[filt->nums[i]] = i;
336 
337     }
338 
339     return reversed;
340 }
341 
342 // Renvoie le plus grand nom de simplexe + 1 dans une filtration
343 // unsigned long long filtrationMaxName(Filtration *filt){
344 //     unsigned long long max = 0;
345 //     for(int i=0; i<filt->size; i++){
346 //         if (filt->nums[i] > max){
347 //             max = filt->nums[i];
348 //         }
349 //     }
350 //     return max+1;
351 // }
352 
353 // Ecrit une filtration dans un fichier
354 void filtrationToFile(Filtration *filtration, Point* pts, int n, char *filename){
355     FILE *f = fopen(filename, "w");
356     if (f == NULL) {
357         printf("Impossible d'ouvrir le fichier !\n");
358         exit(1);
359     }
360 
361     // Ecriture des points dans le fichier
362     for(int i=0;i<n;i++){
363         fprintf(f, "v %d %f %f\n", i, pts[i].x, pts[i].y);
364     }
365 
366     // Ecriture des faces dans le fichier
367     for(int i=0;i<filtration->size;i++){
368         if(filtration->filt[i] != -1){
369             Simplex s = simplexFromId(i, n);
370             fprintf(f, "t %d %d %d %lld\n", s.i, s.j, s.k, filtration->nums[i]);
371         }
372     }
373 
374     fclose(f);
375 
376 }
```

II - src/geometry.h

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 14-09-2024 22:16:49
 *  Last modified : 25-04-2025 21:44:41
 *  File : geometry.h
 */
#ifndef GEOMETRY_H
#define GEOMETRY_H

#include <stdbool.h>

extern const int DIM;

typedef struct {
    float x;
    float y;
} Point;

typedef struct {
    Point *pts;
    float *weights;
    int size;
    float **dist;
} PointCloud;

// Un simplexe est défini par une figure à au plus 3 sommets
typedef struct {
    int i;
    int j;
    int k;
} Simplex;

typedef struct {
    bool *simplices;
    int size;
} SimComplex;

typedef struct {
    int x;
    int y;
} Tuples;

typedef struct {
    int *filt; // F[i] = k si le simplexe i est dans le complexe k
    unsigned long long *nums; // N[i] = k si le simplexe i est le k-ième simplexe ajouté
    int size;
    unsigned long long max_name;
} Filtration;

// Math
extern float dist_euclidean(int p1, int p2, PointCloud *X);
extern float dist(int p1, int p2, PointCloud *X);

// Points
extern void pointPrint(Point p);
extern bool pointAreEqual(Point p1, Point p2);
extern PointCloud *pointCloudInit(int size);
extern void pointCloudFree(PointCloud *pointCloud);
extern PointCloud *pointCloudLoad(char *filename, char *dist_filename);
void pointCloudPrint(PointCloud *X);

// Simplexes
extern Simplex *simplexInit(int i, int j, int k);
extern int simplexId(Simplex *s, int n);
extern Simplex simplexFromId(int id, int n);
extern void simplexFree(Simplex *s);
extern unsigned long long simplexMax(int n);
```

```
68  extern void simplexPrint(Simplex *s);
69  extern int dimSimplex(Simplex *s);
70  extern bool isFaceOf(Simplex *s1, Simplex *s2);
71
72  // Simplicial complexes
73  extern SimComplex *simComplexInit(unsigned long long
n);
74  extern void simComplexInsert(SimComplex *cmpx,
Simplex *s, int n);
75  extern void simComplexFree(SimComplex *cmpx);
76  extern bool simComplexContains(SimComplex *cmpx,
Simplex *s, int n);
77
78  // Filtrations
79  extern Filtration *filtrationInit(unsigned long long
size);
80  extern void filtrationFree(Filtration *filtration);
81  extern void filtrationInsert(Filtration *filtration,
Simplex *s, int n, int k,
82      unsigned long long num);
83  extern bool filtrationContains(Filtration
*filtration, Simplex *s, int n);
84  extern void filtrationPrint(Filtration *filt, int n,
bool sorted);
85  extern void filtrationToFile(Filtration *filtration,
Point* pts, int n, char *filename);
86  extern unsigned long long
filtrationMaxName(Filtration *filtration);
87  int *reverseIdAndSimplex(Filtration *filt);
88
89  #endif
```

III - src/list.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 21-04-2025 18:44:32
 *  Last modified : 23-04-2025 22:54:15
 *  File : list.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "list.h"

db_int_list *create_list(){
    db_int_list *l = malloc(sizeof(db_int_list));
    l->nb = 0;
    l->start = NULL;
    l->end = NULL;
    return l;
}

bool is_empty_list(db_int_list *l){
    return l->nb == 0;
}

node *create_node(int v){
    node *n = malloc(sizeof(node));
    n->value = v;
    n->next = NULL;
    n->prec = NULL;
    return n;
}

// Insert n avant v (n avec next et prec NULL)
void insert_before(db_int_list *l, node *n, node *v){
    if (v->prec == NULL){
        v->prec = n;
        n->next = v;
        l->start = n;
    } else {
        v->prec->next = n;
        n->prec = v->prec;
        n->next = v;
        v->prec = n;
    }
    l->nb++;

}

// Insert n après v (n avec next et prec NULL)
void insert_after(db_int_list *l, node *n, node *v){
    if (v->next == NULL){
        v->next = n;
        n->prec = v;
        l->end = n;
    } else {
        v->next->prec = n;
        n->next = v->next;
        n->prec = v;
        v->next = n;
    }
    l->nb++;

}

void append_list(db_int_list *l, int v){
    node *n = create_node(v);
    if (is_empty_list(l)){
        l->start = n;
        l->end = n;
        l->nb++;
    } else {
        node *c = l->start;
```

```
73          bool is_c_greater = false;
74          bool is_elmt_found_in_list = false;
75
76          // après execution, c le plus grand noeud
plus petit que v ou
77          // le dernier noeud de la liste
78          while(c->next != NULL && !is_c_greater && !
is_elmt_found_in_list){
79              if (c->value < v) c = c->next;
80              else if (c->value == v)
is_elmt_found_in_list = true;
81              else is_c_greater = true;
82          }
83
84          // Ajoute que s'il n'est pas déjà présent
85          if (!is_elmt_found_in_list){
86              if (is_c_greater) insert_before(l, n, c);
87              // Si c est le dernier de la liste
88              else if (c->value < v) insert_after(l,
n ,c);
89              else insert_before(l, n, c);
90          }
91      }
92  }
93
94  // Supprime le noeud c de la liste l en libérant la
mémoire
95  void remove_node(db_int_list *l, node *c){
96      // Si milieu de liste
97      if (c->prec != NULL && c->next != NULL){
98          c->prec->next = c->next;
99          c->next->prec = c->prec;
100
101      // Sinon si fin de liste
102      } else if (c->prec != NULL){
103          c->prec->next = NULL;
104          l->end = c->prec;
105
106      // Sinon si debut de liste
107      } else if (c->next != NULL){
108          c->next->prec = NULL;
109          l->start = c->next;
110
111      // Sinon si les deux (liste = [c])
112      } else {
113          l->start = NULL;
114          l->end = NULL;
115
116      }
117
118      l->nb--;
119      free(c);
120  }
121
122  // Supprime l'élément v de la liste l
123  void remove_list(db_int_list *l, int v){
124      if (is_empty_list(l)) return;
125
126      node *c = l->start;
127      bool is_found = false;
128      while(c != NULL && !is_found){
129          if (c->value == v){
130              is_found = true;
131              remove_node(l, c);
132          }
133          c = c->next;
134      }
135  }
136
137  // Crée une nouvelle liste contenant les elmts de l1
et l2
138  db_int_list *join_list(db_int_list *l1, db_int_list
*l2){
139      db_int_list *l = create_list();
```

```
    node *c1 = l1->end;
    node *c2 = l2->end;

    // Ajoute les éléments par ordre décroissant (donc append en O(1)
    // puisque pas besoin de parcourir toute la liste l)
    while(c1 != NULL && c2 != NULL){
        if(c1->value < c2->value){
            append_list(l, c2->value);
            c2 = c2->prec;
        } else {
            append_list(l, c1->value);
            c1 = c1->prec;
        }
    }

    // Si on arrive ici c'est que c1 = NULL ou c2 = NULL donc qu'une seule des
    // deux boucles n'est exécutée : on garde l'ordre
    while(c1 != NULL){
        append_list(l, c1->value);
        c1 = c1->prec;
    }

    while(c2 != NULL){
        append_list(l, c2->value);
        c2 = c2->prec;
    }

    return l;
}

// Crée une nouvelle liste contenant suelement les éléments soit
// dans l1 soit dans l2 mais pas des deux
db_int_list *xor_list(db_int_list *l1, db_int_list *l2){
    db_int_list *l = create_list();
    node *c1 = l1->end;
    node *c2 = l2->end;

    // Ajoute les éléments par ordre décroissant (donc append en O(1)
    // puisque pas besoin de parcourir toute la liste l)
    while(c1 != NULL && c2 != NULL){
        if(c1->value != c2->value){ // Evite le cas de deux mêmes indices
            if(c1->value < c2->value){
                append_list(l, c2->value);
                c2 = c2->prec;
            } else {
                append_list(l, c1->value);
                c1 = c1->prec;
            }
        } else {
            c1 = c1->prec;
            c2 = c2->prec;
        }
    }

    // Si on arrive ici c'est que c1 = NULL ou c2 = NULL donc qu'une seule des
    // deux boucles n'est exécutée : on garde l'ordre
    while(c1 != NULL){
        append_list(l, c1->value);
        c1 = c1->prec;
    }

    while(c2 != NULL){
        append_list(l, c2->value);
```

```
203         c2 = c2->prec;
204     }
205 
206     return l;
207 }
208 
209 
210 void print_list(db_int_list *l){
211     node *c = l->start;
212 
213     printf("[");
214     while(c != NULL){
215         printf("%d", c->value);
216         if (c->next != NULL) printf("; ");
217         c = c->next;
218     }
219     printf("]\n");
220 }
221 
222 int len_list(db_int_list *l){
223     return l->nb;
224 }
225 
226 void free_list(db_int_list *l){
227     node *c = l->start;
228     while(c != NULL){
229         node *v = c;
230         c = c->next;
231         free(v);
232     }
233     free(l);
234 }
```

IV - src/list.h

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 21-04-2025 18:44:39
 *  Last modified : 23-04-2025 23:02:46
 *  File : list.h
 */
#ifndef LIST_H
#define LIST_H

 #include <stdbool.h>

 // Listes ordonnées
 typedef struct node_t {
     int value;
     struct node_t* prec;
     struct node_t* next;
 } node;

 typedef struct {
     int nb;
     node *start;
     node *end;
 } db_int_list;

 db_int_list *create_list();
 void append_list(db_int_list *l, int v);
 void remove_list(db_int_list *l, int v);
 bool is_empty_list(db_int_list *l);
 db_int_list *join_list(db_int_list *l1, db_int_list
*l2);
 db_int_list *xor_list(db_int_list *l1, db_int_list
*l2);
 void print_list(db_int_list *l);
 int len_list(db_int_list *l);
 void free_list(db_int_list *l);


#endif
```

V - src/misc.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 15-09-2024 16:30:38
 *  Last modified : 15-04-2025 20:31:54
 *  File : misc.c
 */

#include <stdio.h>
#include <stdlib.h>

 #include "geometry.h"

 void print_err(char* str){
     fprintf(stderr, "[ERROR] %s", str);
 }

 // Copie une matrice O(n^2)
 int **copy_matrix(int **t, int n){
     int **copy = malloc(n * sizeof(int*));
     for(int i = 0; i<n; i++){
         copy[i] = malloc(n * sizeof(int));
         for(int j = 0; j<n; j++){
             copy[i][j] = t[i][j];
         }
     }
     return copy;
 }

 // Affiche une matrice
 void printMatrix(int **matrix, int n, int m){
     for(int i = 0; i<n; i++){
         for(int j = 0; j<m; j++){
             printf("%d ", matrix[i][j]);
         }
         printf("\n");
     }
 }
```

VI - src/misc.h

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 15-09-2024 15:52:06
 *  Last modified : 12-10-2024 22:37:39
 *  File : misc.h
 */
#ifndef MISC_H
#define MISC_H

 extern void print_err(char* str);
 extern int **copy_matrix(int **t, int n);
 extern void printMatrix(int **matrix, int n, int m);

 #endif
```

VII - src/persDiag.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 10-09-2024 16:33:19
 *  Last modified : 25-04-2025 22:06:35
 *  File : persDiag.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "persDiag.h"
#include "misc.h"
#include "reduc.h"

////////////////////////////
//    Filtrations & VR    //
////////////////////////////

// Ajoute un simplexe à un complexe simplicial en incrémentant le compteur
// si le simplexe n'est pas déjà présent
void addSimplex(SimComplex *K, Simplex *s, int n, int *count){
    if (!simComplexContains(K, s, n)){
        simComplexInsert(K, s, n);
        (*count)++;
    }
}

// Renvoie le VR complexe simplical associé à un nuage de points et un temps t
SimComplex *VRSimplex(PointCloud *X, float t){
    int n = X->size;
    SimComplex *K = simComplexInit(simplexMax(n));

    int count = 0;
    for(int i = 0; i<n; i++){
        // Condition pour considérer un point d'un VR weighted
        if (X->weights[i] < t){
            // Ajoute le simplexe {i}
            Simplex *s_i = simplexInit(-1, -1, i);
            addSimplex(K, s_i, n, &count);

            for(int j = i+1; j<n; j++){
                // Même condition
                if (X->weights[j]<t){
                    // Ajoute le simplexe {j} s'il est pas déjà présent
                    Simplex *s_j = simplexInit(-1, -1, j);
                    addSimplex(K, s_j, n, &count);

                    // Ajout des simplexes de dimensions > 0
                    if (dist(i, j, X) + X->weights[i] + X->weights[j] < 2*t){
                        // Ajoute le simplexe {i, j} à cmpx s'il n'est pas déjà présent
                        Simplex *s_ij = simplexInit(-1, i, j);
                        addSimplex(K, s_ij, n, &count);

                        // S'il existe déjà deux autres simplexes {i, k} et {j, k}
                        // alors on ajoute le simplexe {i, j, k} à cmpx
                        for(int k = 0; k<n; k++){
                            // Evite le cas où on considère les mêmes arêtes
                            if (k == i || k == j)
```

```
continue;
                            Simplex *s_ik =
simplexInit(-1, i, k);
                            Simplex *s_jk =
simplexInit(-1, j, k);
                            if (simComplexContains(K,
s_ik, n) && simComplexContains(K, s_jk, n)){
                                Simplex *s_ijk =
simplexInit(i, j, k);
                                addSimplex(K, s_ijk,
n, &count);
                                simplexFree(s_ijk);
                            }
                            simplexFree(s_ik);
                            simplexFree(s_jk);
                        }
                        simplexFree(s_ij);
                    }
                    simplexFree(s_j);
                }
            }
            simplexFree(s_i);
        }
    }

    K->size = count;

    return K;
}

// Renvoie la distance maximale à considérer lors de
la création
// d'une filtration
float maxDistOfPointCloud(PointCloud *X){
    // Calcule la distance maximale
    float max_d = 0;
    for(int i=0; i<X->size; i++){
        for(int j=i+1; j<X->size; j++){
            if (max_d < dist(i, j, X)) max_d =
dist(i, j, X);
        }
    }

    // Calcule le poids maximal
    float max_w = 0;
    for(int i=0; i<X->size; i++){
        if(X->weights[i]>max_w) max_w = X-
>weights[i];
    }

     // Retourne la distance maximale à considérer
     return max_d + 2*max_w;
}

// Renvoie une filtration associée à un nuage de
points via la VR complexe
// simplicial
Filtration *buildFiltration(PointCloud *X){
    int n = X->size;
    unsigned long long max_simplex =
simplexMax(n); // Nombre maximal de simplexes possible
    float eps = 1; // Epsilon pour la filtration ie
1seconde
    float dist_max = maxDistOfPointCloud(X); //
Distance maximale à considérer
    // Initialisation de la filtration
    Filtration *filt = filtrationInit(max_simplex);

    int nb_simplex = 0;
    int nb_complex = 1; // 0 est le complexe vide
    float t = 0.0;      // Rayon des boules
    int last_size = 0;  // Taille du dernier
complexe simplical
```

```
    while(t < dist_max+eps){
        SimComplex *K = VRSimplex(X, t);

        // Si le complexe n'est pas vide et n'est pas égal au précédent
        if (K->size != 0 && K->size != last_size){
            last_size = K->size;

            for(unsigned long long s = 0; s<max_simplex; s++){
                // Si le simplexe est présent dans le complexe
                if (K->simplices[s]){
                    // Récupère le simplexe associé à l'identifiant
                    Simplex simp = simplexFromId(s, n);

                    // Si le simplexe n'a jamais été rencontré
                    if (!filtrationContains(filt, &simp, n)){
                        filtrationInsert(filt, &simp, n, (int) t, nb_simplex);
                        nb_simplex++;
                    }
                }
            }

            nb_complex++;
        }

        simComplexFree(K);
        t = t + eps;
    }

    return filt;
}

// Renvoie une liste de paires correspondant aux idéntifiants des simplexes
// récupérés depuis la matrice réduite O(filt->max_name)
Tuple *extractPairsFilt(int *low, Filtration *filt, unsigned long long *size_pairs,
 int *reversed){
    Tuple *pairs = malloc(filt->max_name * sizeof(Tuple));
    *size_pairs = 0;

    bool seen[filt->max_name];
    for(unsigned long long i=0; i<filt->max_name; i++) seen[i] = false;

    unsigned long long count = 0;
    for(unsigned long long j = filt->max_name-1; count < filt->max_name; j--){
        if (!seen[j]){ // Pas déjà appairé
            if (low[j] != -1){
                // On a trouvé une paire
                int x = filt->filt[reversed[low[j]]];
                int y = filt->filt[reversed[j]];
                if (x != y){
                    pairs[*size_pairs].x = reversed[low[j]];
                    pairs[*size_pairs].y = reversed[j];
                    (*size_pairs)++;
                }

                seen[low[j]] = true;
                seen[j] = true;
            } else {
```

```
                // On a trouvé un cycle encore en
vie
                pairs[*size_pairs].x = reversed[j];
                pairs[*size_pairs].y = -1;
                (*size_pairs)++;
            }
        }
        count++;
    }

    Tuple *pairs_resized = realloc(pairs,
*size_pairs * sizeof(Tuple));
    assert(pairs_resized != NULL);
    return pairs_resized;
}

////////////////////////////
//  Persistance Diagram   //
////////////////////////////

PersistenceDiagram *persDiag_create(){
    PersistenceDiagram *pd =
malloc(sizeof(PersistenceDiagram));
    pd->size_death1D = 0;
    pd->size_death1D = 0;
    pd->size_pairs = 0;
    return pd;
}

void fillPairs(PersistenceDiagram *pd, Filtration
*filtration, Tuple *pairs){
    // Assignation du rang d'apparition des
simplexes dans la filtration
    // depuis la liste de paires
    pd->pairs = malloc(pd->size_pairs *
sizeof(Tuple));
    for(unsigned long long i=0; i<pd->size_pairs; i+
+){
    pd->pairs[i].x = filtration->filt[pairs[i].x];

    if (pairs[i].y != -1)
        pd->pairs[i].y = filtration-
>filt[pairs[i].y];
    else
        pd->pairs[i].y = -1;
    }
}

// Crée un diagramme de persistance à partir d'une
filtration injective
PersistenceDiagram *PDCreateV1(Filtration
*filtration, PointCloud *X){
    PersistenceDiagram *pd = persDiag_create();

    // Matrice tq reversed[i] = j si le simplexe j
est le i-ème simplexe de la filtration
    // Cohérent dans l'hypothèse de filtration
injective
    printf("reversed\n");
    int *reversed =
reverseIdAndSimplex(filtration); // O(filt->max_name)

    // Matrice de bordure associée à la filtration
    printf("boundary\n");
    int **boundary = buildBoundaryMatrix(reversed,
filtration->max_name, X->size); //O(filt->max_name)

    // Matrice low associée à la matrice de bordure
    printf("low\n");
    int *low = buildLowMatrix(boundary, filtration-
>max_name); //O(filt->max_name^2)

    // Réduction de la matrice de bordure
    printf("reduced\n");
```

```
    reduceMatrix(boundary, filtration->max_name, low); //O(filt->max_name^3)

    printf("pairs\n");
    Tuple *pairs = extractPairsFilt(low, filtration, &(pd->size_pairs), //O(filt->max_name)
        reversed);
    fillPairs(pd, filtration, pairs);

    // Récupération des dimensions des simplexes et donc catégorises les
    // classes d'homologie
    printf("dims\n");
    pd->dims = malloc(pd->size_pairs * sizeof(int));
    for(unsigned long long i=0; i<pd->size_pairs; i++){
        Simplex s = simplexFromId(pairs[i].x, X->size);

        pd->dims[i] = dimSimplex(&s);
        if(dimSimplex(&s) == 1) pd->size_death1D++;
    }

    // Rajoute les temps de naissance des tueurs de classes 1D
    printf("deathD1\n");
    pd->death1D = malloc(pd->size_death1D * sizeof(Simplex));
    int c = 0;
    for(unsigned long long i=0; i<pd->size_pairs; i++){
        Simplex s = simplexFromId(pairs[i].x, X->size);
        Simplex death_s = simplexFromId(pairs[i].y, X->size);
        if(dimSimplex(&s) == 1)
        {
            pd->death1D[c] = death_s;
            c++;
        }
    }

    // Libération de la mémoire
    free(reversed);
    for(unsigned long long i=0; i<filtration->max_name; i++) free(boundary[i]);
    free(boundary);
    free(low);
    free(pairs);
    return pd;
}

PersistenceDiagram *PDCreateV2(Filtration *filtration, PointCloud *X){
    PersistenceDiagram *pd = persDiag_create();

    // Tableau tq reversed[i] = k avec i l'indice du simplexe j dans
    // la filtration
    int *reversed = reverseIdAndSimplex(filtration); // O(filt->max_name)

    // Matrice de bordure
    boundary_mat B = buildBoundaryMatrix2(reversed, filtration->max_name, X->size);

    // Tableau ou chaque case contient une liste des simplexes de dimension
    // egale a l'indice
    db_int_list **dims = simpleByDims(B, reversed, X->size);

    // Reduction
    reduceMatrixOptimized(B, dims);
```

```
292
293     // Remplissage du tableau low
294     int *low = malloc(sizeof(int)*(filtration->max_name));
295     for(unsigned long long int i=0; i<filtration->max_name; i++)
296         low[i] = get_low(B, i);
297
298     // Tableau des paires (s_i, s_j) de la matrice réduite
299     Tuple *pairs = extractPairsFilt(low, filtration, &(pd->size_pairs), //O(filt->max_name)
300         reversed);
301
302     // Rempli les paires de (K_i, K_j) associés aux paires précédentes
303     fillPairs(pd, filtration, pairs);
304
305     // Récupération des dimensions des simplexes et donc catégorises les
306     // classes d'homologie
307     pd->dims = malloc(pd->size_pairs * sizeof(int));
308     for(unsigned long long i=0; i<pd->size_pairs; i++){
309         Simplex s = simplexFromId(pairs[i].x, X->size);
310
311         pd->dims[i] = dimSimplex(&s);
312         if(dimSimplex(&s) == 1) pd->size_death1D++;
313     }
314
315     // Rajoute les temps de naissance des tueurs de classes 1D
316     pd->death1D = malloc(pd->size_death1D * sizeof(Simplex));
317     int c = 0;
318     for(unsigned long long i=0; i<pd->size_pairs; i++){
319         Simplex s = simplexFromId(pairs[i].x, X->size);
320         Simplex death_s = simplexFromId(pairs[i].y, X->size);
321         if(dimSimplex(&s) == 1)
322         {
323             pd->death1D[c] = death_s;
324             c++;
325         }
326     }
327
328
329     free(reversed);
330     free_boundary(B);
331     for(int i=0; i<=DIM; i++){
332         free_list(dims[i]);
333     }
334     free(dims);
335     free(low);
336     free(pairs);
337
338     return pd;
339 }
340
341 // Exporte un diagramme de persistance dans un fichier
342 void PDExport(PersistenceDiagram *pd, char *filename, char *death_filename,
343   bool bigger_dims){
344     FILE *f = fopen(filename, "w");
345     if (f == NULL){
346         print_err("Erreur lors de l'ouverture du fichier");
347     }
348
349     // Ecriture des paires
```

```
350     for(unsigned long long i=0; i<pd->size_pairs; i+
+){
351         if (pd->dims[i] >= 2 && !bigger_dims)
continue; // On veut que dims 0 et 1
352 
353         if (pd->pairs[i].y == -1){
354             fprintf(f, "%d %d inf\n", pd->dims[i],
pd->pairs[i].x);
355         } else {
356             fprintf(f, "%d %d %d\n",pd->dims[i], pd-
>pairs[i].x, pd->pairs[i].y);
357         }
358     }
359 
360     fclose(f);
361 
362     // Ecriture des simplexes tuant les classes 1D
363     FILE *f_death = fopen(death_filename, "w");
364     for(int i = 0; i<pd->size_death1D; i++)
365         fprintf(f_death, "%d %d %d\n", pd-
>death1D[i].i,
366             pd->death1D[i].j, pd->death1D[i].k);
367 
368     fclose(f_death);
369     printf("Diagram exported at %s\n", filename);
370 }
371 
372 // Libère la mémoire allouée pour un diagramme de
persistance
373 void PDFree(PersistenceDiagram *pd){
374     free(pd->dims);
375     free(pd->pairs);
376     free(pd->death1D);
377     free(pd);
378 }
```

VIII - src/persDiag.h

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 10-09-2024 16:23:55
 *  Last modified : 25-04-2025 22:09:29
 *  File : persDiag.h
 */
#ifndef PERSDIAG_H
#define PERSDIAG_H

#include "geometry.h"

typedef struct {
    int x;
    int y;
} Tuple;

typedef struct {
    int *dims;
    Tuple *pairs;
    unsigned long long size_pairs;
    Simplex *death1D; // Simplexes tuant des 1D homologies
    int size_death1D;
    int size_dims;
} PersistenceDiagram;

extern Filtration *buildFiltration(PointCloud *X);
extern int *reverseIdAndSimplex(Filtration *filt);
extern int **buildBoundaryMatrix(int *reversed, unsigned long long n, int nb_pts);
extern int *buildLowMatrix(int **boundary, unsigned long long n);
extern Tuple *extractPairsFilt(int *low, Filtration *filt,
    unsigned long long *size_pairs, int *reversed);

PersistenceDiagram *PDCreateV1(Filtration *filtration, PointCloud *X);
PersistenceDiagram *PDCreateV2(Filtration *filtration, PointCloud *X);
void PDExport(PersistenceDiagram *pd, char *filename, char *death_filename, bool bigger_dims);
void PDFree(PersistenceDiagram *pd);

#endif
```

IX - src/reduc.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 22-04-2025 20:23:47
 *  Last modified : 25-04-2025 22:23:03
 *  File : reduc.c
 */
#include <stdio.h>
#include <stdlib.h>

#include "geometry.h"
#include "misc.h"
#include "list.h"
#include "reduc.h"

// Renvoie la matrice low associée à une matrice de bordure O(n^2)
int *buildLowMatrix(int **boundary, unsigned long long n){
    int *low = malloc(n*sizeof(int));
    for(unsigned long long j=0; j<n; j++){
        low[j] = -1;
        for(unsigned long long i=0; i<n; i++)
            if(boundary[i][j] != 0) low[j] = i;
    }
    return low;
}


// Renvoie l'indice de la première colonne de la matrice low ayant la même valeur O(i)
int sameLow(int *low, int i){
    for(int j=i-1; j>=0; j--){
        if(low[i] == low[j] && low[i] != -1){
            return j;
        }
    }
    return -1;
}

// Met à jour la colonne j de la matrice low O(n)
void updateLow(int **boundary, int* low, int j, int n){
    low[j] = -1;
    for(int i=0; i<n; i++)
        if(boundary[i][j] != 0) low[j] = i;
}

// Construit la matrice de bordure associée à une filtration en O(max_name)
// reversed est le tableau des identifiants des simplexes dans la filtration
// nb_pts est le nb de points dans l'ensemble
// max_name est le nom maximal attribué dans une filtration
int **buildBoundaryMatrix(int *reversed, unsigned long long max_name, int nb_pts){
    int **boundary = malloc(max_name * sizeof(int*));

    for(unsigned long long i = 0; i<max_name; i++){
        boundary[i] = malloc(max_name * sizeof(int));

        for(unsigned long long j = 0; j<max_name; j++){
            Simplex s1 = simplexFromId(reversed[i], nb_pts);
            Simplex s2 = simplexFromId(reversed[j], nb_pts);
            if (isFaceOf(&s1, &s2)) boundary[i][j] = 1;
            else boundary[i][j] = 0;
        }
    }
```

```

    return boundary;
}

// Standart Algorithm O(n^3)
void reduceMatrix(int **boundary, unsigned long long n, int *low){
    for(unsigned long long i=0; i<n; i++){
        int j = sameLow(low, i); // O(n)
        while(j != -1){ // O(n) * O(n)
            // Soustraction de la colonne j à la colonne i
            for(unsigned long long k=0; k<n; k++){
                boundary[k][i] = (boundary[k][i] + boundary[k][j]) % 2;
            }

            updateLow(boundary, low, i, n);
            j = sameLow(low, i);
        }
    }
}

// Récupère le plus grand indice de ligne tq la case est non nulle
// O(1)
int get_low(boundary_mat B, int j){
    if(is_empty_list(B.s[j])) return -1;
    else return B.s[j]->end->value;
}

// Renvoie un tableau dim de sorte que dim[i] est une liste de noms
// de simplexes de dimension i ; il faut que D soit la dimension maximale
// de tous les simplexes O(nb total de simplexes^2)
db_int_list **simpleByDims(boundary_mat B, int *reversed, int n){
    db_int_list **dims = malloc(sizeof(db_int_list*)*(DIM+1));
    for(int j=0; j<=DIM; j++)
        dims[j] = create_list();

    for(int i=0; i<B.n; i++){
        Simplex s = simplexFromId(reversed[i], n);
        append_list(dims[dimSimplex(&s)], i);
    }
     return dims;
}

// Construit la matrice de bordureV2 associée à une filtration en O(max_name)
// reversed est le tableau des identifiants des simplexes dans la filtration
// nb_pts est le nb de points dans l'ensemble
// max_name est le nom maximal attribué dans une filtration
boundary_mat buildBoundaryMatrix2(int *reversed, unsigned long long max_name, int nb_pts){
    boundary_mat B = boundary_init(max_name);

    for(unsigned long long i = 0; i<max_name; i++){
        for(unsigned long long j = 0; j<max_name; j++){
            Simplex s1 = simplexFromId(reversed[i], nb_pts);
            Simplex s2 = simplexFromId(reversed[j], nb_pts);
            if (isFaceOf(&s1, &s2)) append_list(B.s[j], i);
        }
    }

```

```
    return B;
}

// Réduit la matrice de bordure de façon intelligente
// simplexes_by_dims est un tableau de listes de simplexes pour lequel l'indice
//      i correspond à la liste de simplexes de dimension i
void reduceMatrixOptimized(boundary_mat B, db_int_list **simplexes_by_dims){
    // Réduction de chacune des matrices par les dimensions (de 0 à D-1)
    for(int d=0; d<DIM; d++){
        // Simplexes de dimensions d+1 (cad consitituant les colonnes)
        db_int_list *simplexes = simplexes_by_dims[d+1];
        node *c = simplexes->start;

        // Boucle sur les colonnes en croissant, dim[d+1] itérations au pire
        while(c != NULL){
            int j = c->value;

            node *c_i = c->prec;
            while(c_i != NULL) { // dim[d] itération au pire
                int i = c_i->value;
                int k = get_low(B, i);
                int k2 = get_low(B, j);

                // Si la colonne a le même low
                if (k == k2 && k != -1){
                    db_int_list *col = xor_list(B.s[i], B.s[j]); // O(l1 * l2)
                    free(B.s[j]);
                    B.s[j] = col;
                    c_i = c->prec; // Recommence la lecture pour trouver les low
                } else {
                    c_i = c_i->prec;

                }

            }

            c = c->next;
        }
    }
}

// Initialise une matrice de bordure
boundary_mat boundary_init(int n){
    boundary_mat B;
    B.n = n;
    B.s = malloc(n*sizeof(db_int_list*));
    for(int i=0; i<n; i++)
        B.s[i] = create_list();

    return B;
}

// Création d'une matrice intxint a partir de la representation
// boundary_mat
int **boundary_to_mat(boundary_mat B){
    // Allocations mémoires
    int **m = calloc(B.n, sizeof(int*));
    for(int i=0; i<B.n; i++)
        m[i] = calloc(B.n, sizeof(int));

    // Remplissage de la matrice
    // On rappelle que B.s[j] contient tous les
```

```
indices i
180      // de ligne ou m[i][j] != 0
181      for(int j=0; j<B.n; j++){
182          node *c = B.s[j]->start;
183          while(c != NULL){
184              m[c->value][j] = 1;
185              c = c->next;
186          }
187      }
188
189      return m;
190  }
191
192  // Affiche la matrice de bordure B
193  void print_boundary(boundary_mat B){
194      int **m = boundary_to_mat(B);
195
196      for(int i=0; i<B.n; i++){
197          for(int j=0; j<B.n; j++)
198              printf("%d ", m[i][j]);
199
200          printf("\n");
201      }
202
203      for(int i=0; i<B.n; i++)
204          free(m[i]);
205
206      free(m);
207
208  }
209
210  void free_boundary(boundary_mat B){
211      for(int i=0; i<B.n; i++)
212          free_list(B.s[i]);
213
214      free(B.s);
215  }
```

X - src/reduc.h

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 22-04-2025 20:23:49
 *  Last modified : 25-04-2025 22:07:19
 *  File : reduc.h
 */
#ifndef REDUC_H
#define REDUC_H

#include "list.h"

typedef struct {
    int n; // Longueur de la liste de sommmets s
    db_int_list **s;

} boundary_mat;

// Version 1 de la reduction
int *buildLowMatrix(int **boundary, unsigned long long n);
int **buildBoundaryMatrix(int *reversed, unsigned long long max_name, int nb_pts);
void reduceMatrix(int **boundary, unsigned long long n, int *low);

// Version 2 de la reduction
boundary_mat buildBoundaryMatrix2(int *reversed, unsigned long long max_name, int nb_pts);
int get_low(boundary_mat B, int j);
void reduceMatrixOptimized(boundary_mat B, db_int_list **simplexes_by_dims);
db_int_list **simpleByDims(boundary_mat B, int *reversed, int n);

// Misc
boundary_mat boundary_init(int n);
void free_boundary(boundary_mat B);
void print_boundary(boundary_mat B);
int **boundary_to_mat(boundary_mat B);

#endif
```

XI - prgms/analyse_cplx.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 24-04-2025 22:30:19
 *  Last modified : 25-04-2025 21:05:27
 *  File : analyse_cplx.c
 *  Ce fichier a pour but de produire un fichier sur différents temps
 *  d'exécution en fonction de la version de réduction
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "../src/geometry.h"
#include "../src/persDiag.h"

// Crée un nuage de n points aléatoire
PointCloud *create_random(int n){
    PointCloud *X = pointCloudInit(n);
    for(int i=0; i<n; i++){
        X->pts[i].x = rand()%100;
        X->pts[i].y = rand()%100;
        X->weights[i] = rand()%50;
    }

    return X;
}


int main(){
    int seed = 0;
    srand(seed);

    int nb_sizes = 11;
    int sizes[11] = {5, 10, 20, 30, 40, 50, 60, 65, 70, 75, 80};


    FILE *file = fopen("times_cmpx.dat", "w");
    for(int i=7; i<nb_sizes; i++){
        printf("Traitement taille=%d\n", sizes[i]);
        PointCloud *X = create_random(sizes[i]);
        Filtration *f = buildFiltration(X);

        printf("Debut V1\n");
        clock_t start_V1 = clock();
        PDCreateV1(f, X);
        clock_t end_V1 = clock();

        printf("Debut V2\n");
        clock_t start_V2 = clock();
        PDCreateV2(f, X);
        clock_t end_V2 = clock();

        double elapV1 = (double)(end_V1 - start_V1)/CLOCKS_PER_SEC;
        double elapV2 = (double)(end_V2 - start_V2)/CLOCKS_PER_SEC;
        filtrationFree(f);
        pointCloudFree(X);
        fprintf(file, "%d %f %f\n", sizes[i], elapV1, elapV2);
        fflush(file);
        printf("Fin traitement\n");
    }

    fclose(file);
}
```

XII - tests/tests_geometry.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 24-09-2024 16:57:12
 *  Last modified : 23-04-2025 22:41:38
 *  File : tests_geometry.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "../src/geometry.h"

void test_geometry(){
    // Tests points
    Point p1 = {1, 2};
    Point p2 = {3, 4};

    assert(!pointAreEqual(p1, p2));
    assert(pointAreEqual(p1, p1));

    // Tests simplex
    Simplex *s = simplexInit(2, 3, 1);
    assert(s->i == 1);
    assert(s->j == 2);
    assert(s->k == 3);

    assert(simplexId(s, 3) == 78);

    Simplex s0 = simplexFromId(52, 3);
    assert(s0.i == -1);
    assert(s0.j == 0);
    assert(s0.k == 2);

    simplexFree(s);

    // Tests des dimensions de simplexes
    Simplex *s1 = simplexInit(0, -1, -1);   // Point
    Simplex *s2 = simplexInit(0, 1, -1);    // Arête
    Simplex *s3 = simplexInit(0, 1, 2);     // Triangle

    assert(dimSimplex(s1) == 0);
    assert(dimSimplex(s2) == 1);
    assert(dimSimplex(s3) == 2);

    // Tests de faces
    Simplex *s4 = simplexInit(0, 1, 2);
    Simplex *s5 = simplexInit(0, 1, 3);
    Simplex *s6 = simplexInit(2, 1, -1);

    assert(isFaceOf(s4, s5) == 0); // Pb de dimension
    assert(isFaceOf(s6, s4) == 1); // Ok
    assert(isFaceOf(s6, s5) == 0); // Pas une face

    simplexFree(s1);
    simplexFree(s2);
    simplexFree(s3);
    simplexFree(s4);
    simplexFree(s5);
    simplexFree(s6);

}
```

XIII - tests/tests_list.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 22-04-2025 18:45:23
 *  Last modified : 23-04-2025 21:57:41
 *  File : tests_list.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "../src/list.h"
#include "tests_list.h"

void tests_list(){
    printf("--- Tests list ---\n");
    db_int_list *l = create_list();
    printf("Liste vide : ");
    print_list(l);
    assert(len_list(l) == 0);
    assert(is_empty_list(l));

    append_list(l, 5);
    append_list(l, 0);
    append_list(l, 0);
    append_list(l, 4);
    append_list(l, 3);
    assert(len_list(l) == 4);
    printf("Liste ordonnée sans doublon de 5 0 0 4 3 : ");
    print_list(l);

    assert(is_empty_list(l) == false);

    printf("Meme liste apres suppression de 8 et 4 : ");
    remove_list(l, 8);
    remove_list(l, 4);
    assert(len_list(l) == 3);
    print_list(l);

    db_int_list *l2 = create_list();
    append_list(l2, 9);
    append_list(l2, 5);
    append_list(l2, 0);
    append_list(l2, -7);
    append_list(l2, -7);
    append_list(l2, 11);

    printf("Nouvelle liste de 9 5 0 -7 -7 et 11 : ");
    print_list(l2);
    db_int_list *l_join = join_list(l, l2);
    assert(len_list(l_join) == 6);
    printf("Union des deux listes précédentes : ");
    print_list(l_join);

    printf("Xor de cette union et de la liste 7 11 5 4 : ");
    db_int_list *l3 = create_list();
    append_list(l3, 7);
    append_list(l3, 11);
    append_list(l3, 5);
    append_list(l3, 4);
    db_int_list *l_xor = xor_list(l_join, l3);
    print_list(l_xor);
}
```

XIV - tests/tests_persDiag.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 08-10-2024 17:01:34
 *  Last modified : 24-04-2025 22:41:06
 *  File : tests_persDiag.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>

#include "tests_persDiag.h"
#include "../src/persDiag.h"
#include "../src/geometry.h"

void tests_persDiag(){
    // Tests boundary matrix
    // Création d'une filtration à la main dont on connait la matrice
    // basé sur https://iuricichf.github.io/ICT/algorithm.html

    // Définition de la filtration
    int N = 11; // Nombre de points dans la filtration
    Simplex **simPts = malloc(N*sizeof(Simplex *));
    for(int i=0; i<N; i++) simPts[i] = simplexInit(-1, -1, i);

    Simplex **simEdges = malloc(N*sizeof(Simplex *));
    simEdges[0] = simplexInit(-1, 0, 3);
    simEdges[1] = simplexInit(-1, 1, 5);
    simEdges[2] = simplexInit(-1, 2, 6);
    simEdges[3] = simplexInit(-1, 3, 4);
    simEdges[4] = simplexInit(-1, 5, 6);
    simEdges[5] = simplexInit(-1, 3, 7);
    simEdges[6] = simplexInit(-1, 3, 8);
    simEdges[7] = simplexInit(-1, 8, 4);
    simEdges[8] = simplexInit(-1, 9, 5);
    simEdges[9] = simplexInit(-1, 10, 6);
    simEdges[10] = simplexInit(-1, 7, 8);

    Simplex *f = simplexInit(3, 7, 8); // Face

    int max_simplex = (N+1)*(N+1)*(N+1);
    Filtration *base_filt = filtrationInit(max_simplex);
    for(int i=0; i<3; i++) filtrationInsert(base_filt, simPts[i], N, 1, i);
    for(int i=3; i<7; i++) filtrationInsert(base_filt, simPts[i], N, 2, i);
    for(int i=0; i<5; i++) filtrationInsert(base_filt, simEdges[i], N, 2, i+7);
    for(int i=7; i<N; i++) filtrationInsert(base_filt, simPts[i], N, 3, i+5);
    for(int i=5; i<N; i++) filtrationInsert(base_filt, simEdges[i], N, 3, i+11);
    filtrationInsert(base_filt, f, N, 3, 22);

    int *reversed = reverseIdAndSimplex(base_filt);

    filtrationPrint(base_filt, N, true);

    int *low = malloc(base_filt->max_name*sizeof(int));
    for(int i=0; i<base_filt->max_name; i++){
        low[i] = -1;
    }
    low[7] = 3;
    low[8] = 5;
    low[9] = 6;
    low[10] = 4;
```

```
    low[11] = 6;
    low[16] = 12;
    low[17] = 13;
    low[18] = 13;
    low[19] = 14;
    low[20] = 15;
    low[21] = 13;
    low[22] = 21;

    // Tests de l'extraction des paires par rapport à la filtration initiale
    unsigned long long size_pairs_filt;
    Tuple *pairs_filt = extractPairsFilt(low, base_filt, &size_pairs_filt,
        reversed);
    Tuple pair = {1, 2};
    unsigned long long c = 0;

    for(unsigned long long i=0; i<size_pairs_filt; i++){
        if (pairs_filt[i].y != -1){
            assert(c==0);
            assert(base_filt->filt[pairs_filt[i].x] == pair.x &&
                base_filt->filt[pairs_filt[i].y] == pair.y);
            c++;
        }
    }

    // Tests de création de diagramme de persistance
    PointCloud *X = pointCloudInit(N);

    // On se fiche de la valeur des points pour l'instant
    for(int i=0; i<N; i++)
        X->pts[i] = (Point) {0, 0};


    PersistenceDiagram *pd = PDCreateV2(base_filt, X);

    c = 0;
    for(unsigned long long i=0; i<pd->size_pairs; i++){
        if (pd->pairs[i].y != -1){
            assert(pd->pairs[i].x == pair.x &&
                pd->pairs[i].y == pair.y);
            c++;
        }
    }

    // Tests de l'exportation
    PDExport(pd, "exportedPD/test.dat", "exportedPD/test_death.txt", true);

    // Libération de la mémoire
    for(int i=0; i<11; i++) simplexFree(simPts[i]);
    for(int i=0; i<11; i++) simplexFree(simEdges[i]);
    simplexFree(f);
    free(simPts);
    free(simEdges);
    free(reversed);
    free(low);
    free(pairs_filt);
    PDFree(pd);
    free(X->pts);
    free(X);
    filtrationFree(base_filt);
}
```

XV - tests/tests_reduc.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 23-04-2025 20:33:47
 *  Last modified : 25-04-2025 22:08:36
 *  File : tests_reduc.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "tests_reduc.h"
#include "../src/geometry.h"
#include "../src/reduc.h"
#include "../src/misc.h"

void tests_reduc(){
    printf("--- Tests reduc.h ---\n");

    int N = 11; // Nombre de points
    int n = 23; // Nombre de simplexes

    // Filtration sur laquelle les tests se reposent
    // basé sur https://iuricichf.github.io/ICT/algorithm.html
    Simplex **simPts = malloc(N*sizeof(Simplex *));
    for(int i=0; i<N; i++) simPts[i] = simplexInit(-1, -1, i);

    Simplex **simEdges = malloc(N*sizeof(Simplex *));
    simEdges[0] = simplexInit(-1, 0, 3);
    simEdges[1] = simplexInit(-1, 1, 5);
    simEdges[2] = simplexInit(-1, 2, 6);
    simEdges[3] = simplexInit(-1, 3, 4);
    simEdges[4] = simplexInit(-1, 5, 6);
    simEdges[5] = simplexInit(-1, 3, 7);
    simEdges[6] = simplexInit(-1, 3, 8);
    simEdges[7] = simplexInit(-1, 8, 4);
    simEdges[8] = simplexInit(-1, 9, 5);
    simEdges[9] = simplexInit(-1, 10, 6);
    simEdges[10] = simplexInit(-1, 7, 8);

    Simplex *f = simplexInit(3, 7, 8); // Face

    Filtration *base_filt = filtrationInit(simplexMax(N));
    for(int i=0; i<3; i++) filtrationInsert(base_filt, simPts[i], N, 1, i);
    for(int i=3; i<7; i++) filtrationInsert(base_filt, simPts[i], N, 2, i);
    for(int i=0; i<5; i++) filtrationInsert(base_filt, simEdges[i], N, 2, i+7);
    for(int i=7; i<N; i++) filtrationInsert(base_filt, simPts[i], N, 3, i+5);
    for(int i=5; i<N; i++) filtrationInsert(base_filt, simEdges[i], N, 3, i+11);
    filtrationInsert(base_filt, f, N, 3, 22);

    int *reversed = reverseIdAndSimplex(base_filt);

    // Matrice de bordure associée à une filtration
    int **trueBoundary = calloc(n, sizeof(int*));
    boundary_mat B = boundary_init(n); // Temporaire, à remplacer par testB1 plus bas

    // Calcule la véritable matrice de bordure
    for(int i=0; i<n; i++){
        trueBoundary[i] = calloc(n, sizeof(int));
        for(int j=0; j<n; j++){
            bool condition =
                (i==0 && j==7) ||
                (i==1 && j==8) ||
                (i==2 && j==9) ||
```

```
                (i==3 && j==7) || (i==3 && j==10) ||
(i==3 && j==16) || (i==3 && j==17) ||
                (i==4 && j==10) || (i==4 && j==18) ||
                (i==5 && j==8) || (i==5 && j==11) ||
(i==5 && j==19) ||
                (i==6 && j==9) || (i==6 && j==11) ||
(i==6 && j==20) ||
                (i==12 && j==16) || (i==12 && j==21)
||
                (i==13 && j==17) || (i==13 && j==18)
|| (i==13 && j==21) ||
                (i==14 && j==19) ||
                (i==15 && j==20) ||
                (i==16 && j==22) ||
                (i==17 && j==22) ||
                (i==21 && j==22);

            if (condition){
                trueBoundary[i][j] = 1;
                append_list(B.s[j], i);
            }
            else trueBoundary[i][j] = 0;
        }
    }

    // Teste la construction de la matrice de bordure
    // Version 1
    int **testB1 = buildBoundaryMatrix(reversed,
base_filt->max_name, N);
    for(int i=0; i<base_filt->max_name; i++){
        for(int j=0; j<base_filt->max_name; j++){
            assert(testB1[i][j] == trueBoundary[i]
[j]);
        }
    }

    // Test la construction de la matrice de bordure
    // Version 2
    boundary_mat testB2_bound =
buildBoundaryMatrix2(reversed, base_filt->max_name, N);
    int **testB2 = boundary_to_mat(testB2_bound);
    for(int i=0; i<base_filt->max_name; i++){
        for(int j=0; j<base_filt->max_name; j++){
            assert(testB2[i][j] == trueBoundary[i]
[j]);
        }
    }

    // Teste si low est correcte
    // Vraie matrice low d'exemple
    int *true_low = malloc(n*sizeof(int));
    for(int i=0; i<n; i++){
        true_low[i] = -1;
    }
    true_low[7] = 3;
    true_low[8] = 5;
    true_low[9] = 6;
    true_low[10] = 4;
    true_low[11] = 6;
    true_low[16] = 12;
    true_low[17] = 13;
    true_low[18] = 13;
    true_low[19] = 14;
    true_low[20] = 15;
    true_low[21] = 13;
    true_low[22] = 21;

    int *test_lowV1 = buildLowMatrix(testB1, n);
    int *test_lowV2 = buildLowMatrix(testB2, n);
    for(int j=0; j<23; j++){
        assert(test_lowV1[j] == true_low[j]);
        assert(test_lowV2[j] == true_low[j]);

    }
```

```
129
130     // Teste si la liste de listes par dimension est
bonne :
131     int D = 2; // Nombre de dimensions des
simplexes
132     db_int_list **dims = simpleByDims(B, reversed,
N);
133     for(int i=0; i<=DIM; i++){
134         printf("Les simplexes de dimension %d :\n",
i);
135         print_list(dims[i]);
136     }
137
138     // Teste la réduction
139     // Véritable matrice réduite
140     int **true_reduced = malloc(n*sizeof(int *));
141     for(int i=0; i<n; i++){
142         true_reduced[i] = malloc(n*sizeof(int));
143         for(int j=0; j<n; j++){
144             bool condition =
145                 (i==0 && j==7) ||
146                 (i==1 && j==8) || (i==1 && j==11) ||
147                 (i==2 && j==9) || (i==2 && j==11) ||
148                 (i==3 && j==7) || (i==3 && j==10) ||
(i==3 && j==16) || (i==3 && j==17) ||
149                 (i==4 && j==10) ||
150                 (i==5 && j==8) || (i==5 && j==19) ||
151                 (i==6 && j==9) || (i==6 && j==20) ||
152                 (i==12 && j==16) ||
153                 (i==13 && j==17) ||
154                 (i==14 && j==19) ||
155                 (i==15 && j==20) ||
156                 (i==16 && j==22) ||
157                 (i==17 && j==22) ||
158                 (i==21 && j==22);
159             if (condition) true_reduced[i][j] = 1;
160             else true_reduced[i][j] = 0;
161         }
162     }
163
164     printf("Matrice de bordure : \n");
165     printMatrix(trueBoundary, n, n);
166
167     // Teste si la matrice de la V1 est bien réduite
168     int *low = buildLowMatrix(trueBoundary, n);
169     reduceMatrix(trueBoundary, n, low);
170     for(int i=0; i<n; i++){
171         for(int j=0; j<n; j++)
172             assert(trueBoundary[i][j] ==
true_reduced[i][j]);
173     }
174
175     // Teste si la matrice de la V2 est bien réduite
176     reduceMatrixOptimized(testB2_bound, dims);
177     printf("Affichage de la matrice réduite : \n");
178     print_boundary(testB2_bound);
179
180     int **reducedV2 = boundary_to_mat(testB2_bound);
181     for(int i=0; i<n; i++){
182         for(int j=0; j<n; j++)
183             assert(reducedV2[i][j] ==
true_reduced[i][j]);
184     }
185
186     // Tests de low après réduction
187     int *true_low_reduced = malloc(23*sizeof(int));
188     for(int i=0; i<23; i++){
189         true_low_reduced[i] = -1;
190     }
191     true_low_reduced[7] = 3;
192     true_low_reduced[8] = 5;
193     true_low_reduced[9] = 6;
194     true_low_reduced[10] = 4;
195     true_low_reduced[11] = 2;
```

```
    true_low_reduced[16] = 12;
    true_low_reduced[17] = 13;
    true_low_reduced[19] = 14;
    true_low_reduced[20] = 15;
    true_low_reduced[22] = 21;

    int *test_low_reducedV1 =
buildLowMatrix(trueBoundary, n);
    int *test_low_reducedV2 =
buildLowMatrix(reducedV2, n);

    for(int j=0; j<23; j++){
        assert(true_low_reduced[j] ==
test_low_reducedV1[j]);
        assert(true_low_reduced[j] ==
test_low_reducedV2[j]);

    }


    // Libérations
    for(int i=0; i<N; i++){
        free(simPts[i]);
        free(simEdges[i]);
    }
    free(f);
    filtrationFree(base_filt);
    free(reversed);
    for(int i=0; i<n; i++){
        free(trueBoundary[i]);
        free(testB1[i]);
        free(testB2[i]);
    }
    free(trueBoundary);
    free_boundary(B);
    free(testB1);
    free(testB2);
    free_boundary(testB2_bound);
    free(true_low);
    free(test_lowV1);
    free(test_lowV2);
    for(int i=0; i<n; i++){
        free(true_reduced[i]);
        free(reducedV2[i]);
    }
    free(reducedV2);
    for(int i=0; i<=D; i++)
        free_list(dims[i]);
    free(dims);
    free(low);
    free(true_low_reduced);
    free(test_low_reducedV1);
    free(test_low_reducedV2);
}
```

XVI - /main.c

```c
/*
 *  Contact : Elowan - elowarp@gmail.com
 *  Creation : 08-10-2024 17:08:19
 *  Last modified : 25-04-2025 21:22:21
 *  File : main.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "src/geometry.h"
#include "src/persDiag.h"
#include "src/misc.h"

int main(int argc, char *argv[]){
    if (argc<2){
        print_err("Il faut un nom de ville !\n");
        return 1;
    }

    bool euclidean = false;
    char *name =
malloc((strlen(argv[1])+1)*sizeof(char));
    strcpy(name, argv[1]);

    if (argc==3) {
        if (strcmp(argv[1], "-e")!=0) {
            print_err("Trop d'options ! -e pour
choisir des distances euclidiennes\n");
            return 1;
        } else {
            name = realloc(name,
(strlen(argv[2])+1)*sizeof(char));
            strcpy(name, argv[2]);
            euclidean = true;
        }
    }

    char *pts_filename = malloc((strlen(name) +
14)*sizeof(char));
    char *pd_filename = malloc((strlen(name) +
16)*sizeof(char));
    char *death_filename = malloc((strlen(name) +
22)*sizeof(char));

    strcpy(pts_filename, "data/");
    strcpy(pd_filename, "exportedPD/");
    strcpy(death_filename, "exportedPD/");
    strcat(pts_filename, name);
    strcat(pd_filename, name);
    strcat(death_filename, name);
    strcat(pts_filename, "_pts.txt");
    strcat(pd_filename, ".dat");
    strcat(death_filename, "_death.txt");

    char *dist_filename = NULL;
    if (!euclidean){
        dist_filename = malloc((strlen(name) +
15)*sizeof(char));
        strcpy(dist_filename, "data/");
        strcat(dist_filename, name);
        strcat(dist_filename, "_dist.txt");
    }

    // Routine principale
    printf("Chargement de l'ensemble des points...
\n");
    PointCloud *X = pointCloudLoad(pts_filename,
dist_filename);
    printf("Construction d'une filtration...\n");
    Filtration *filt = buildFiltration(X);
    printf("Construction du diagramme de
```

```
persistance...\n");
64      PersistenceDiagram *pd = PDCreateV1(filt, X);
65
66      printf("Exportation du diagramme de
persistance...\n");
67      PDExport(pd, pd_filename, death_filename, false);
68
69      PDFree(pd);
70      filtrationFree(filt);
71      pointCloudFree(X);
72      free(pts_filename);
73      free(dist_filename);
74      free(pd_filename);
75      free(death_filename);
76      free(name);
77      printf("Fin des calculs et de l'exportation\n");
78      return 0;
79  }
```

XVII - /retrieve_data.py

```python
'''
  Contact : Elowan - elowarp@gmail.com
  Creation : 06-11-2024 10:52:34
  Last modified : 08-12-2024 20:48:49
  File : retrieve_data.py

  Ce fichier permet depuis un fichier d'une liste de points selon la norme
  donnée par le projet de récupérer toutes les distances nécessaires au
  calcul des classes d'homologies. Elle utilise l'api de Geoapify, dont
   on supposera la clé apparaitre dans un fichier .env à la racine du
   dossier Code.
'''
import datetime
import requests
import numpy as np
from dotenv import load_dotenv
import os
import subprocess
from csvkit.utilities.csvsql import CSVSQL

load_dotenv()
API_GEOAPIFY = os.getenv("API_GEOAPIFY")

def build_dist(filename):
    url = "https://api.geoapify.com/v1/routing?waypoints={}%7C{}&mode={}&apiKey={}"
    lat, lon = np.loadtxt("data/"+filename+"_pts.txt", skiprows=1, unpack=True, usecols=[0, 1], dtype="str")

    nb_lines = len(lon) # Nb lignes dans le fichier

    with open('{}_dist.txt'.format(filename), 'w') as file:
        file.write(str(nb_lines) + "\n")

        for i in range(nb_lines):
            for j in range(nb_lines):
                if i < j:
                    coord1 = ",".join([lat[i], lon[i]])
                    coord2 = ",".join([lat[j], lon[j]])
                    # Calcule le temps min entre les deux moyennes
                    resp1 = requests.request("GET", url.format(coord1, coord2, "drive", API_GEOAPIFY))
                    resp2 = requests.request("GET", url.format(coord2, coord1, "drive", API_GEOAPIFY))
                    time1 = resp1.json()["features"][0]["properties"]["time"]
                    time2 = resp2.json()["features"][0]["properties"]["time"]
                    moy1 = (time1+time2)/2

                    resp1 = requests.request("GET", url.format(coord1, coord2, "walk", API_GEOAPIFY))
                    resp2 = requests.request("GET", url.format(coord2, coord1, "walk", API_GEOAPIFY))
                    time1 = resp1.json()["features"][0]["properties"]["time"]
                    time2 = resp2.json()["features"][0]["properties"]["time"]
                    moy2 = (time1+time2)/2

                    file.write("{} {} {}\n".format(i, j, min(moy1, moy2)))
                    print("i={}, j={} done".format(i,
```

```
j))
53
54 if __name__ == "__main__":
55     build_dist("marseille")
```