**Polytech'Lille — GIS 3 — Structures de Données — Feuille de TP numéro 4**

---

Ce TP porte sur l'algorithme de Graham, qui calcule l'enveloppe convexe d'un nuage de points. Le but pédagogique est le suivant : comprendre suffisamment l'algorithme pour faire la correspondance avec le code C donné en fin d'énoncé ; savoir implanter une pile de points par une liste en adaptant un code existant ; connaître quelques aspects supplémentaires du langage C : fonction `qsort` de la bibliothèque standard et écriture dans des fichiers.

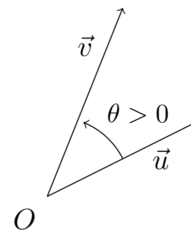
# 1 Préliminaires

## 1.1 Orientation dans le plan

Soient $\vec{u}$ et $\vec{v}$ deux vecteurs dans le plan, de coordonnées

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \end{pmatrix}, \quad \vec{v} = \begin{pmatrix} x_v \\ y_v \end{pmatrix}.$$

Supposons qu'on veuille superposer $\vec{u}$ sur $\vec{v}$ en effectuant une rotation autour de l'origine (voir figure de droite). Faut-il tourner dans le sens trigonométrique ou dans le sens inverse ? Réponse : il suffit de tester le signe de l'angle orienté $\theta$ défini par les deux vecteurs ; il faut tourner dans le sens trigonométrique si le signe est positif, dans le sens inverse s'il est négatif. Ce signe s'obtient par un simple calcul de déterminant :




$$\text{signe}(\theta) = \text{signe} \begin{vmatrix} x_u & x_v \\ y_u & y_v \end{vmatrix}.$$

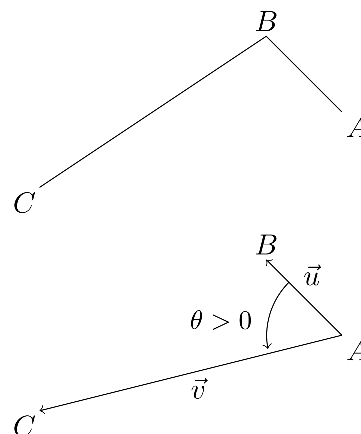
**Question 1.** Pour superposer $\vec{u}$ sur $\vec{v}$, faut-il tourner dans le sens trigonométrique ou dans le sens inverse ?

$$\vec{u} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad \vec{v} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}.$$

Considérons maintenant trois sommets consécutifs

$$A = \begin{pmatrix} x_A \\ y_A \end{pmatrix}, \quad B = \begin{pmatrix} x_B \\ y_B \end{pmatrix}, \quad C = \begin{pmatrix} x_C \\ y_C \end{pmatrix}$$

d'un polygone (figure de droite, en haut) et imaginons qu'on se déplace de $A$ vers $C$. Arrivé en $B$, tourne-t-on vers la gauche ou vers la droite ? Réponse : on tourne à gauche si l'angle $\theta$ entre $\vec{u}$ et $\vec{v}$ est positif (figure de droite, en bas), où $\vec{u} = B - A$ désigne le vecteur de $A$ vers $B$ et $\vec{v} = C - A$ désigne le vecteur de $A$ vers $C$. Cela revient à appliquer la méthode décrite en début de section en prenant $A$ comme origine.




$$\text{signe}(\theta) = \text{signe} \begin{vmatrix} x_B - x_A & x_C - x_A \\ y_B - y_A & y_C - y_A \end{vmatrix}.$$

**Question 2.** Lorsqu'on se déplace de $A$ vers $C$ (en passant par $B$), tourne-t-on vers la gauche ou vers la droite ? Même question si on se déplace de $C$ vers $A$.

$$A = \begin{pmatrix} 4 \\ 1 \end{pmatrix}, \quad B = \begin{pmatrix} 3 \\ 2 \end{pmatrix}, \quad C = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

## 1.2 Trier un tableau en C

La fonction `qsort`, de la bibliothèque standard du langage C, permet de trier des tableaux d'éléments d'un type quelconque, suivant une relation d'ordre quelconque. Prenons l'exemple d'un tableau $T$ de $n$ doubles, qu'on souhaite trier par ordre croissant. Il suffit d'écrire une fonction de comparaison, paramétrée par les adresses $p$ et $q$ de deux éléments de $T$, qui retourne $-1$, 0 ou 1, suivant que le double désigné par $p$ est inférieur, inférieur ou égal ou supérieur au double désigné par $q$. Pour des raisons de généricité, les adresses passées en paramètre sont de type `const void*`. Il suffit d'effectuer une conversion de pointeurs en début de fonction pour se ramener au type `double*`.

```
int compare_doubles (const void* p0, const void* q0)
{   double* p = (double*)p0;
    double* q = (double*)q0;
    if (*p < *q)
        return -1;
    else if (*p == *q)
        return 0;
    else
        return 1;
}
```

Dans la fonction qui doit trier $T$, l'appel à `qsort` peut alors s'écrire :

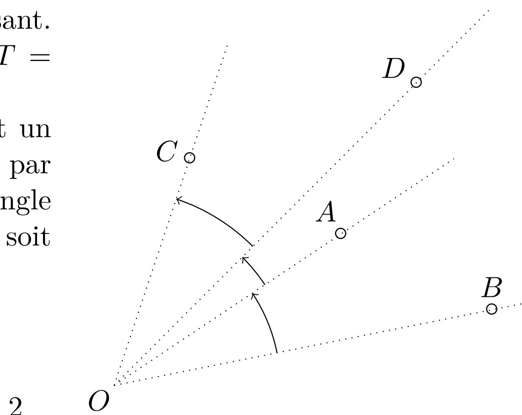
```
    double T [n];
    ...
    qsort (T, n, sizeof (double), &compare_doubles);
```

**Question 3.** Modifier la fonction `compare_doubles` pour que $T$ soit trié par ordre décroissant.

## 1.3 Trier un tableau de points par angle croissant

On se donne un tableau $T$ de $n$ points dans le plan, qu'on souhaite trier par « angle » croissant. Sur la figure de droite, cela revient à obtenir $T = [B, A, D, C]$.

Plus précisément, chaque point de $T$ définit un vecteur (sur la figure, les vecteurs sont portés par les droites en pointillé). On souhaite que l'angle orienté $\theta$ entre deux vecteurs consécutifs de $T$ soit de signe positif.



**Question 4.** En C, supposons que $T$ soit un tableau de $n$ éléments de type `struct point` (déclaration ci-dessous). Écrire une fonction de comparaison `compare_points` qui puisse être passée à `qsort` pour trier $T$ par angle croissant.

```
struct point {
    double x;     /* abscisse */
    double y;     /* ordonnée */
    char ident;   /* identificateur */
};
```

**Question 5.** Écrire une fonction `tourne_a_gauche`, paramétrée par trois points $A$, $B$ et $C$, qui retourne `true` si le chemin de $A$ vers $C$ tourne à gauche en $B$, et `false` sinon.

## 2 L'algorithme de Graham

L'algorithme de Graham calcule l'enveloppe convexe d'un ensemble de $n$ points du plan (voir Figure 1) avec une complexité en $\Theta(n \log(n))$. Un pseudo-code est donné Figure 2. Voir aussi [1, section 33.3].

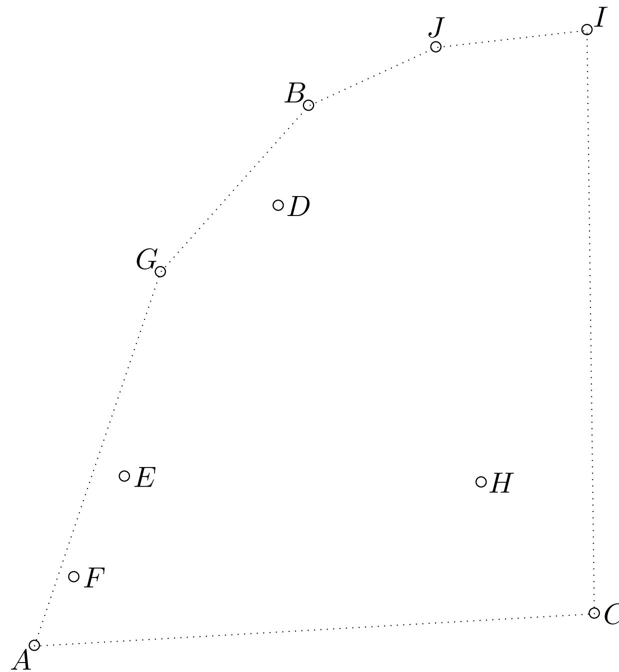



Figure 1 – Enveloppe convexe d'un ensemble de points. Supposons que l'origine soit en $A$ et énumérons les autres les sommets de l'enveloppe convexe par angle croissant : $C$, $I$, $J$, $B$, $G$. On remarque une propriété, exploitée par l'algorithme de Graham : à chaque point rencontré, on tourne à gauche.

```
procedure Graham (T)
  T[0, 1, ..., n − 1] est un tableau de n points (n ≥ 3). L'origine est en T_0.
    Les autres points ont des coordonnées strictement positives.
  begin
    Trier T[1, ..., n − 1] par angle croissant
  On suppose pour simplifier que tous les angles sont non nuls
    Initialiser une pile de points à vide
    Empiler T_0, puis T_1
  La pile contient toujours au moins deux points
    i := 2
    while i < n do
      cour := le point en sommet de pile
      prec := le point en-dessous du sommet de pile
      On considère le chemin défini par les trois points prec, cour et T_i
      if ce chemin tourne à gauche en cour then
        Empiler T_i
        i := i + 1
      else
        Dépiler un point
      end if
    end do
    Le contenu de la pile constitue l'enveloppe convexe de T. Afficher ce contenu
  end
```

FIGURE 2 – L'algorithme de Graham. Cet algorithme utilise une pile de points. On peut montrer qu'à chaque itération, cette pile contient au moins deux éléments.

Appliquons le pseudo-code de la Figure 2 sur l'exemple de la Figure 1. L'origine est en $A$. Après le tri initial, le tableau $T$ contient : $A$, $C$, $H$, $I$, $J$, $F$, $D$, $E$, $B$, $G$. Les valeurs successives de la pile sont :

| | |
|---|---|
| $A, C$ | $A, C, I, J, D$ |
| $A, C, H$ | $A, C, I, J, D, E$ |
| $A, C, I$ | $A, C, I, J, B$ |
| $A, C, I, J$ | $A, C, I, J, B, G$ |
| $A, C, I, J, F$ | |

# 3 Réalisation

Une archive est disponible sur `cristal.univ-lille.fr/~boulier/polycopies/SD/Graham.tgz`.

**Question 6.** Coder le fichier `point.c` dont le fichier d'entête est ci-dessous (voir questions 4 et 5).

```
/* point.h */

#include <stdbool.h>
struct point {
    double x;       /* abscisse */
    double y;       /* ordonnée */
    char ident;     /* identificateur */
};

extern void init_point (struct point*, double, double, char);
extern int compare_points (const void*, const void*);
extern bool tourne_a_gauche (struct point*, struct point*, struct point*);
```

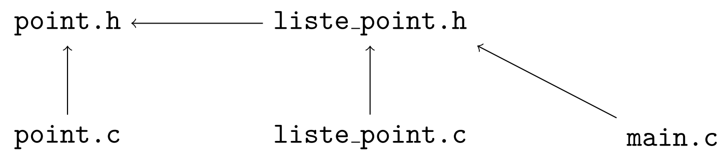



Figure 3 – Représentation graphique du logiciel à réaliser.

**Question 7.** On souhaite implanter la pile de points de l'algorithme de Graham au moyen d'une liste chaînée. Adapter le module `liste_double` en un module `liste_point` qui permette de coder naturellement l'algorithme de Graham. Faut-il ajouter des fonctions à ce module ? Compléter le code donné en fin de document. Voir Figure 3.

```
/* liste_double.h */

struct maillon_double
{   double value;
    struct maillon_double* next;
};

struct liste_double
{   struct maillon_double* tete;
    int nbelem;
};

extern void init_liste_double (struct liste_double*);
extern void clear_liste_double (struct liste_double*);
extern void ajouter_en_tete_liste_double (struct liste_double*, double);
extern void extraire_tete_liste_double (double*, struct liste_double*);
```

## Références

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique.* Dunod, Paris, 2ème edition, 2002.

```c
/* Graham.c */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "liste_point.h"

#define N 10
#define MAX_COORDINATES 100
#define SCENARIO 78738

int main ()
{   struct point T[N];
    struct liste_point L;
    FILE* f;
    int i;

    srand48 (SCENARIO);
/*
 * On crée N points. Le point A est en (0,0).
 * On les enregistre dans "points.dat"
 */
    init_point (&T[0], 0, 0, 'A');
    for (i = 1; i < N; i++)
        init_point (&T[i], drand48 () * MAX_COORDINATES,
                           drand48 () * MAX_COORDINATES, 'A' + i);
    f = fopen ("points.dat", "w");
    assert (f != NULL);
    for (i = 0; i < N; i++)
        fprintf (f, "%f %f %c\n", T[i].x, T[i].y, T[i].ident);
    fclose (f);
/*
 * On trie T [1 .. N-1] par angle croissant.
 * On s'assure qu'il n'y a pas deux points alignés.
 */
    qsort (T+1, N-1, sizeof (struct point), &compare_points);
    for (i = 1; i < N-1; i++)
        assert (compare_points (&T[i], &T[i+1]) != 0);
/*
 * À FAIRE : BOUCLE PRINCIPALE DE L'ALGORITHME DE GRAHAM.
 *          UTILISER L POUR LA PILE DE POINTS.
 */
    f = fopen ("enveloppe.dat", "w");
    assert (f != NULL);
    fprintf (f, "%f %f\n", T[0].x, T[0].y);
/*
 * À FAIRE : IMPRIMER TOUS LES POINTS DE L DANS "enveloppe.dat"
 */
    fclose (f);
/*
 * Commande Gnuplot:
 * plot [-10:100][-10:100] "points.dat" with labels, "enveloppe.dat" with lines
 */
    return 0;
}
```