Autour des enveloppes convexes

1 Introduction

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `enveloppes` contenant une source en C et un makefile :

```
curl cdn.sci-phy.org/mp2i/tp12-enveloppes.tgz | tar xvz
```

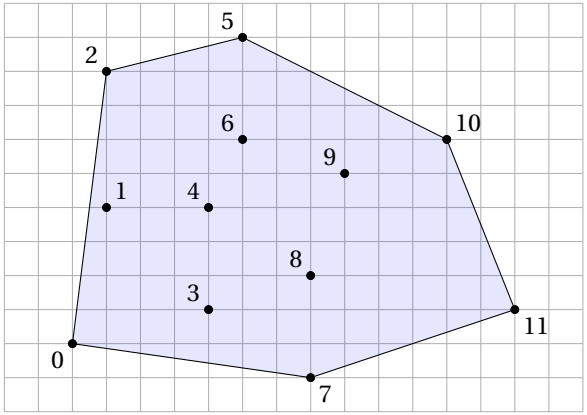
Le fichier à compléter est le fichier `enveloppes.c`. On y trouvera toutes les fonctions que l'on souhaite créer, à compléter, ainsi que quelques fonctions déjà écrites destinées aux tests.

Il s'agit du seul fichier dans ce projet, mais le répertoire contient également un `makefile` pour compiler le compiler comme d'habitude (On rappelle les commandes « `make all` » pour compiler le programme, « `make run` » pour le compiler et l'exécuter si la compilation a réussi).

1.2 Enveloppes convexes

Rappelons tout d'abord qu'une partie $\mathcal{C}$ du plan $\mathbb{R}^2$ est convexe si et seulement si pour toute paire de point $p, q \in \mathcal{C}^2$ le segment $[p, q]$ est inclus dans $\mathcal{C}$. L'enveloppe convexe d'un ensemble $\mathcal{P}$ de points de $\mathbb{R}^2$ est le plus petit convexe contenant $\mathcal{P}$.

Lorsque $\mathcal{P}$ est un ensemble fini (on parle de nuage de points), le bord de l'enveloppe convexe est un polygône convexe dont les sommets appartiennent à $\mathcal{P}$, comme dans l'exemple ci-dessous :



On représentera en langage C un nuage de n points $\mathcal{P}$ par un tableau à n lignes et 2 colonnes, tel que celui défini ci-dessous :

```c
int pts_1[][2] = {{0,0}, {1,4}, {1,8}, {4,1}, {4,4}, {5,9}, {5,6},
                                               {7,-1}, {7,2}, {8,5}, {11,6}, {13,1}};
```

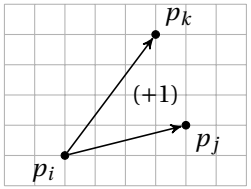
On fournit une fonction `void disp_points(int pts[][2], int n)` qui affiche les points d'un nuage dans l sortie standard. L'unité « élémentaire » utilisé est la ligne verticalement, une taille de deux caractères horizontalement. Précisons que si le nuage contient deux points de même ordonnée et d'abscisse très proches, il est possible que celui de droite ne soit pas affiché pour des raisons de place.

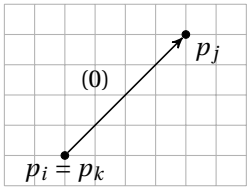
Dans la suite, on notera p_i le point d'index i dans le tableau représentant $\mathcal{P}$. **Pour simplifier, on supposera dans la suite qu'il n'y a pas de points confondus, ni de triplets de points distincts $p_i, p_j, p_k \in \mathcal{P}^3$ qui soient alignés.**

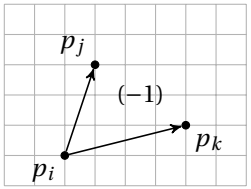
Dans un premier temps, on se pose la question de l'orientation d'un triplet de trois points $p_i, p_j, p_k \in \mathcal{P}^3$.

1. Écrire en C une fonction `int orientation(int pts[][2], int i, int j, int k)` prenant en argument un tableau représentant un nuage de points et trois index i, j et k (pas nécessairement distincts, mais tous compris entre 0 et $n-1$ où n est le nombre de points de $\mathcal{P}$) et renvoyant :

- 0 si les points sont alignés (ce qui, compte tenu des hypothèses, signifie qu'un même point est utilisé au moins deux fois) ;
- 1 si l'angle $\overrightarrow{p_i p_j}, \overrightarrow{p_i, p_k}$ est dans $]0, \pi[$;
- -1 si l'angle $\overrightarrow{p_i p_j}, \overrightarrow{p_i, p_k}$ est dans $]-\pi, 0[$.







2. Dans le fichier fourni, le nuage d'exemple est mémorisé dans le tableau `pts_1`. Vérifier que `orientation(pts_1, 3, 9, 6)` renvoie 1 et `orientation(pts_1, 4, 8, 1)` renvoie -1.

Cette fonction d'orientation va nous permettre de construire, de différentes manières, l'enveloppe convexe du nuage de points.

2 Algorithme de Jarvis

2.1 Principe

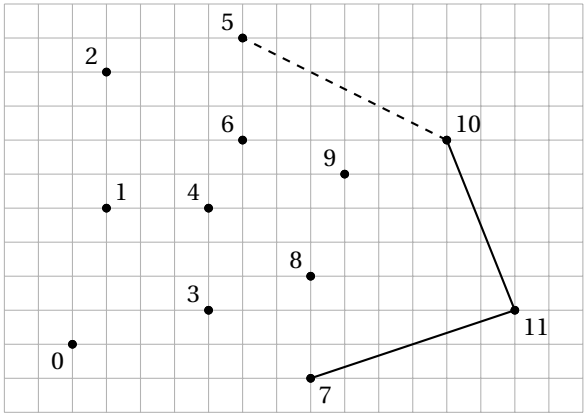
Ce premier algorithme, proposé par R. Jarvis en 1973, consiste à envelopper peu à peu le nuage de points $\mathcal{P}$ dans une sorte de « paquet cadeau », qui à la fin du processus est exactement le bord de l'enveloppe convexe. On commence par choisir le point de plus petite ordonnée (celui d'index 7 sur l'exemple) dans le nuge de points (le plus à gauche s'il y en a deux), puis à chaque étape de la procédure on sélectionne le prochain point du nuage.

La procédure de sélection fonctionne comme suit. Soit p_i le dernier point inséré dans le paquet cadeau à cet instant. Par exemple $i = 10$ dans l'exemple de la figure ci-dessous. Considérons la relation binaire $\preceq_i$ définie sur $\mathcal{P} \setminus \{p_i\}$ par

$$p_j \preceq_i p_k \quad \equiv \quad \texttt{orientation}(\texttt{pts}, i, j, k) \leqslant 0.$$

3. Vérifier que $\preceq_i$ est une relation d'ordre total sur $\mathcal{P} \setminus \{p_i\}$.

À chaque étape de l'algorithme, le point suivant le point p_i, dans la séquence qui constituera le bord de l'enveloppe convexe, est le plus grand de $\mathcal{P} \setminus \{p_i\}$ pour l'ordre défini par $\preceq_i$, ce qui peut être calculé en temps linéaire en la taille du nuage. Ainsi, après p_{10}, on choisira ainsi le point p_5 :



2.2 Implémentation

4. Proposer une fonction `int lowest(int pts[][2], int n)` prenant en entrée un tableau contenant un nuage de points et sa taille et retournant l'index du point avec la plus petite ordonnée (en cas d'égalité des ordonnées deux deux points, on choisira le point de plus petite abscisse, même si cela n'a en fait guère d'importance avec les hypothèses précédentes).

5. Justifier que le point identifié par la fonction `lowest` est nécessairement sur le bord de l'enveloppe convexe du nuage de points.

6. Proposer une fonction `int next(int pts[][2], int n, int i)` prenant en argument un index i désignant un point P_i et retournant l'index du point P_j maximal, dans $\mathcal{P} \setminus \{\mathrm{P}_i\}$, pour $\preceq_i$.

7. En déduire une fonction `int jarvis(int pts[][2], int n, int* buffer)` prenant en argument un nuage de point, sa taille, et un buffer supposé suffisamment grand (on utilisera par exemple l'adresse d'un tableau statique de taille n, comme proposé dans le `main` fourni), remplissant le (début du) buffer avec les index des points se situant sur le bord de l'enveloppe convexe (en partant du point retourné par `lowest`, en les égrennant dans le sens direct, et sans répéter le point initial à la fin) et renvoyant le nombre de points sur ladite enveloppe convexe.

8. Vérifier que, pour le tableau d'exemple `pts_1`, le buffer contient bien les points constituant le bord de l'enveloppe convexe après un appel à la fonction `jarvis`. On pourra utiliser la fonction `print_array(int buffer[], int n)` qui prend en argument un tableau et un nombre d'éléments n et affiche les n premiers éléments du tableau, et utiliser la fonction d'affichage pour vérifier la position des différents points (attention, le tri a pu changer la numérotation des points).

9. Vérifier que l'on obtient bien la même enveloppe convexe avec `pts_2` (il s'agit des mêmes points mais dans un ordre différent).

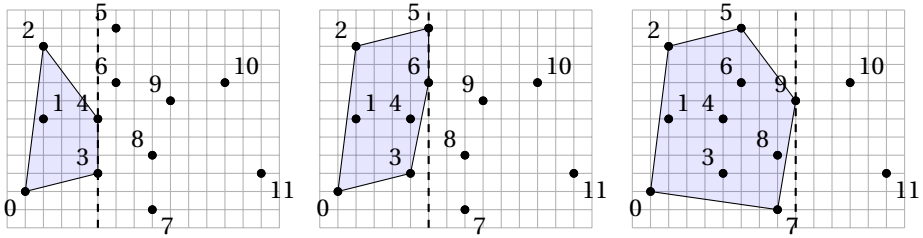
10. Tester de même le bon fonctionnement de `jarvis` sur le nuage de points `pts_3`.

11. Quelle est la complexité de la fonction `jarvis` ?

3 Algorithme de Graham

3.1 Principe

L'idée de l'algorithme de Graham est de balayer le nuage de points horizontalement de gauche à droite par une droite verticale, tout en mettant à jour l'enveloppe convexe des points de $\mathcal{P}$ situés à gauche de cette droite, comme illustré ci-dessous :



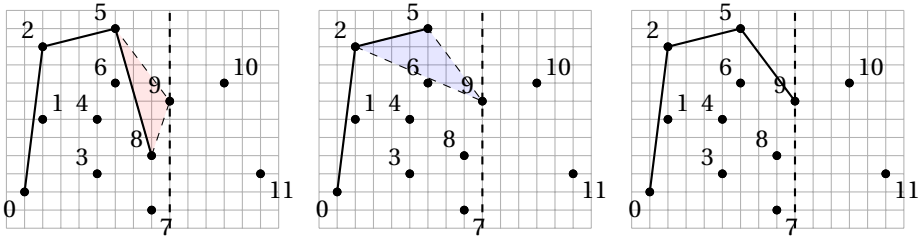
Plus précisément, l'algorithme visite chaque point de $\mathcal{P}$ une fois, par ordre croissant

d'abscisse. Afin de pouvoir les traiter plus simplement par ordre croissant d'index, on commencera donc par les trier (par simplicité, les points sont déjà triés par abscisse croissante dans le présent exemple). Puis, à chaque nouveau point p_i visité, il met à jour le bord de l'enveloppe convexe du sous-nuage $\{p_0, p_1, \ldots, p_i\}$ situé à gauche de p_i.

On remarque que les points p_0 et p_i sont sur ce bord, et on appelle enveloppe supérieure la partie du bord située au-dessus de la droite passant par p_0 et p_i (p_0 et p_i compris), et enveloppe inférieure la partie du bord située au-dessous (p_0 et p_i compris). Le bord de l'enveloppe convexe est donc constitué de l'union de ces deux enveloppes, après suppression des doublons de p_0 et p_i.

Par exemple, dans le cas du nuage $\mathcal{P}$ de la figure du dessus à gauche, le sous-nuage $\{p_0, p_1, p_2, p_3, p_4\}$ a pour enveloppe supérieure la séquence (p_0, p_2, p_4) et pour enveloppe inférieure la séquence (p_0, p_3, p_4), le bord de son enveloppe convexe étant donné par la séquence (p_0, p_3, p_4, p_2). Informatiquement, les indices des sommets des enveloppes inférieure et supérieure seront stockés dans deux piles d'entiers séparées, `lower_env` (pour enveloppe inférieure) et `upper_env` (pour enveloppe supérieure).

La mise à jour de l'enveloppe supérieure est illustrée dans la figure ci-dessous : tant que le point visité (p_9 dans ce cas) et les deux points dont les indices sont situés au sommet de la pile `upper_env` (dans l'ordre : p_8 et p_5) forme une séquence (p_9, p_8, p_5) d'orientation négative, on dépile l'indice situé au sommet de `upper_env` (8 dans ce cas). On poursuit ce processus d'élimination jusqu'à ce que l'orientation devienne positive ou qu'il ne reste plus qu'un seul indice dans la pile. L'indice du point visité (p_9 dans ce cas) est alors inséré au sommet de `upper_env`. La mise à jour de l'enveloppe inférieure s'opère de manière symétrique.



Ainsi, dans l'exemple ci-dessus, à l'étape considérant le point p_9, on rejette p_8 car l'orientation du triangle est négative, mais on conserve p_5 car l'orientation est positive, et on ajoute 9 sur la pile `upper_env`.

12. Afin de préparer la suite de l'algorithme, proposer une fonction `void sort(int pts[][2], int n)` prenant en argument un pointeur vers un tableau de coordonnées et sa taille et ordonnant les points en place dans le tableau par ordonnée croissante. Bien que le coût de ce tri conditionnera celui de l'algorithme, on pourra simplement ici utiliser un tri simple quadratique.

3.2 Mise en place de la pile

Nous allons avoir besoin d'une pile. Cependant, dans le cas présent, on dispose d'une information intéressante : le nombre d'éléments dans la pile est, à tout instant, majoré par le nombre N de points dans le nuage. Il suffit donc, pour notre pile, d'allouer un tableau de taille N, il ne débordera pas (sous réserve que l'on évite les erreurs dans le programme).

On décrira la pile par le type suivant :

```c
struct stack {
    int* data;
    int size;
}

typede struct stack Stack;
```

On remarquera que l'on ne mémorise pas la taille du tableau, juste le nombre d'éléments stockés, à un instant t, dans la pile. En effet, si tout se passe bien, la pile ne doit pas déborder!

13. Proposer une fonction `Stack create(int n)` qui crée et renvoie une pile vide (la fonction devra incidemment allouer un tableau de taille n, on pourra supposer que cette allocation se passe bien).

14. Proposer une fonction `bool is_empty(Stack s)` qui renvoie un booléen indiquant si la pile passée en argument est vide ou non.

15. Proposer une fonction `void push(int elem, Stack* s)` qui ajoute l'entier `elem` sur la pile `s`. On supposera que l'ajout ne cause pas de débordement de la pile. Attention, on passe ici un pointeur vers la structure de pile puisque la pile sera modifiée!

16. Proposer une fonction `int top(Stack s)` qui renvoie l'entier au sommet de la pile `s`, sans toucher au contenu de la pile.

17. Proposer une fonction `int pop(Stack* s)` qui retire et renvoie l'entier au sommet de la pile `s`.

18. Proposer enfin une fonction `void delete(Stack* s)` qui libère le tableau associé à la pile. Cette fonction devra être appelée avant la fin de la portée de `s` afin d'éviter toute fuite de mémoire. Bien évidemment, aucune manipulation de la pile ne doit être effectuée après un appel à `delete`!

3.3 Implémentation de l'algorithme

19. Proposer une fonction `void update_upper(int pts[][2], Stack* upper_env, int i)` prenant en argument le tableau `pts`, la pile `upper_env` et l'index `i` du point à visiter et mettant à jour l'enveloppe supérieure selon la méthode décrite.

20. Faire de même avec une fonction `update_lower` de même signature pour l'enveloppe inférieure.

21. En déduire une fonction `int graham(int pts[][2], int n, int* buffer)` prenant un tableau décrivant un nuage de points et sa taille, un pointeur vers une zone mémoire supposée suffisamment grande, remplit la zone mémoire avec une séquence d'entiers décrivant une séquence de points p_i décrivant le bord d'une enveloppe convexe dans le sens direct en partant du point 0, et renvoie le nombre de points que comprend cette séquence.

22. Pour les tableaux `pts_1` et `pts_2`, vérifier que la fonction renvoye 6 et que `buffer` les entiers 0, 7, 11, 10, 5 et 2. On pourra utiliser la fonction `void print_array(int* buffer, int n)` pour l'affichage.

23. Tester également le bon fonctionnement de `graham` sur `pts_3`.

24. Déterminer la complexité en temps de la fonction `graham`, en supposant que l'on a utilisé un tri quasilinéaire ($O(n \log n)$).

4 Aller plus loin

On suppose disposer d'un polygone convexe construit à partir d'un ensemble de points p_i du nuage $\mathcal{P}$.

25. En se servant de `orientation`, proposer une fonction `bool inside(int pts[][2], int* buffer, int n)` prenant en argument un nuage de points, une enveloppe convexe (décrite par un tableau d'index et sa taille) et un index de point, et retournant un booléen indiquant si le point se trouve à l'interieur strict de l'enveloppe (on rappelle qu'il n'y a pas de triplet de points alignés).

26. On suppose le nuage de points divisés en deux sous-nuages distincts, et que l'on a calculé une enveloppe convexe pour chacun des deux sous-nuages. Proposer une méthode permettant de construire une enveloppe convexe pour le nuage entier à partir des deux enveloppes. On pourra imposer les conditions que l'on souhaite quant à la partition des points entre les deux sous-nuages.

27. En déduire une méthode dichotomique permettant de construire l'enveloppe convexe d'un nuage, et essayer de l'implémenter. Quelle est sa complexité ?