Projet: Enveloppe convexe
Algorithmique et structures de données
Licence 2 Informatique

Julien Bernard

Table des matières

Le but de ce projet est d'implémenter différents algorithmes pour calculer l'enveloppe convexe d'un nuage de points. L'enveloppe convexe d'un nuage de points est le plus petit polygone qui entoure tous les points. Il existe de nombreux algorithmes pour calculer une enveloppe convexe, nous nous intéresserons à trois de ces algorithmes : la marche de Jarvis *(Jarvis march)*, le parcours de Graham *(Graham scan)* et l'enveloppe rapide *(Quickhull)*.

Dans un premier temps, on implémentera des fonctions relatives à la manipulation de vecteurs et de points, ainsi que les principales structures qui seront utilisées par la suite. Puis dans un deuxième temps, on implémentera les trois algorithmes. Enfin, on implémentera un pilote pour pouvoir visualiser les résultats.

Partie 1 : Structures pour la géométrie

En informatique géométrique, on définit généralement la même structure pour désigner les vecteurs et les points. Ce n'est pas mathématiquement rigoureux mais cette simplification permet d'avoir le même ensemble d'opérations pour les deux objets. Généralement ce type est appelé `vector` ou `vec` (et donc, un point est alors vu comme un vecteur depuis l'origine).

```
struct vec {
  double x;
  double y;
};
```

Dans la suite, on notera x l'abscisse du point P et y l'ordonnée du point P. Si on a deux points P_1 et P_2, alors les coordonnées du vecteur $\overrightarrow{P_1P_2}$ sont $(x_2 - x_1, y_2 - y_1)$.

Les fonctions qui suivent sont simples à implémenter. La difficulté réside dans le fait de bien comprendre leur signification géométrique.

Question 1.1 Écrire une fonction qui calcule le produit scalaire [1] de deux vecteurs $\overrightarrow{V_1}$ et $\overrightarrow{V_2}$. Mathématiquement, le produit scalaire est défini par :

$$x_1x_2 + y_1y_2$$

```
double dot(const struct vec *v1, const struct vec *v2);
```

Le produit scalaire représente la coordonnée du projeté de $\overrightarrow{V_1}$ sur un axe de vecteur unité $\overrightarrow{V_2}$.

Question 1.2 Écrire une fonction qui calcule le produit vectoriel 2D [2] de deux vecteurs $\overrightarrow{P_1P_2}$ et $\overrightarrow{P_1P_3}$ définis par les trois points P_1, P_2, P_3. Mathématiquement, ce produit vectoriel 2D est défini par :

$$(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$$

1. `https://en.wikipedia.org/wiki/Dot_product`
2. `https://en.wikipedia.org/wiki/Cross_product#Computational_geometry`

```
double cross(const struct vec *p1,
    const struct vec *p2, const struct vec *p3);
```

Le produit vectoriel 2D détermine si P_3 est à gauche ou à doite de la droite (P_1P_2). En effet, le produit vectoriel sera positif si l'angle formé par $[P_1P_2)$ et $[P_1P_3)$ est positif, négatif si cet angle est négatif, nul si les trois points sont alignés.

Question 1.3 À partir du produit vectoriel, écrire une fonction qui dit si la suite de point P_1, P_2, P_3 constitue un tournant à gauche.

```
bool is_left_turn(const struct vec *p1,
    const struct vec *p2, const struct vec *p3);
```

Partie 2 : Ensemble de points

Nous allons maintenant considérer des ensembles de points et les opérations qui vont avec. Un ensemble de points sera représenté par un tableau dynamique de points :

```
struct vecset {
  struct vec *data;
  size_t size;
  size_t capacity;
};
```

Question 2.1 Donner le code d'une fonction qui crée un ensemble de points vide.

```
void vecset_create(struct vecset *self);
```

Question 2.2 Donner le code d'une fonction qui détruit un ensemble de points.

```
void vecset_destroy(struct vecset *self);
```

Question 2.3 Donner le code d'une fonction qui ajoute un point à un ensemble de points.

```
void vecset_add(struct vecset *self, struct vec p);
```

—

On considère une fonction de comparaison de points avec un contexte qui renvoie un entier strictement négatif si `p1` est «plus petit» que `p2`, un entier strictement positif si `p1` est «plus grand» que `p2` et 0 si `p1` est «égal» à `p2`.

```
typedef int (*comp_func_t)(const struct vec *p1,
    const struct vec *p2, const void *ctx);
```

Question 2.4 Donner le code d'une fonction qui renvoie le maximum d'un ensemble de points suivant une fonction de comparaison donnée.

```
const struct vec *vecset_max(const struct vecset *self,
    comp_func_t func, const void *ctx);
```

Question 2.5 Donner le code d'une fonction qui renvoie le minimum d'un ensemble de points suivant une fonction de comparaison donnée.

```
const struct vec *vecset_min(const struct vecset *self,
    comp_func_t func, const void *ctx);
```

Question 2.6 Donner le code d'une fonction qui trie l'ensemble de points suivant la fonction de comparaison donnée.

```
void vecset_sort(struct vecset *self, comp_func_t func,
    const void *ctx);
```

—

On va maintenant voir le tableau dynamique comme une pile, avec le haut de la pile à la fin du tableau.

Question 2.7 Donner le code d'une fonction qui empile un élément.

```
void vecset_push(struct vecset *self, struct vec p);
```

Question 2.8 Donner le code d'une fonction qui dépile un élément.

```
void vecset_pop(struct vecset *self);
```

Question 2.9 Donner le code d'une fonction qui renvoie le premier élément de la pile.

```
const struct vec *vecset_top(const struct vecset *self);
```

Question 2.10 Donner le code d'une fonction qui renvoie le second élément de la pile.

```
const struct vec *vecset_second(const struct vecset *self);
```

Partie 3 : Marche de Jarvis

Le principe de la marche de Jarvis est d'envelopper l'ensemble de points dans un papier cadeau. On part d'un point, généralement celui qui a l'ordonnée ou l'abscisse minimale, puis on cherche successivement les points qui permettent d'entourer tous les autres.

L'algorithme 1 décrit la marche de Jarvis de manière formelle. Dans cet algorithme, F est le premier point de l'enveloppe *(First)*, C est le point courant de l'enveloppe *(Current)*, N est le point suivant de l'enveloppe *(Next)*.

Algorithm 1 Marche de Jarvis

```
function JarvisMarch(S)
    R ← ∅
    F ← leftmost point in S
    C ← F
    repeat
        R ← R ∪ {C}
        N ← a point in S
        for all I ∈ S do
            if (C, I, N) is a left turn then
                N ← I
            end if
        end for
        C ← N
    until F = C
    return R
end function
```

La complexité de cet algorithme est particulière puisqu'elle dépend du résultat *(output sensitive)*. En effet, on va faire la boucle principale autant de fois qu'il y a de points dans l'enveloppe convexe. Si on a n points et qu'il y a h points dans l'enveloppe convexe, alors la complexité est en $O(nh)$. Dans le pire cas, tous les points sont dans l'enveloppe convexe et la complexité est en $O(n^2)$.

Question 3.1 Donner le code d'une fonction qui implémente la marche de Jarvis. Attention, cet algorithme doit être implémenté avec soin. En particulier, certaines situations particulières ne sont pas traitées dans l'algorithme 1 mais elles doivent l'être dans votre code.

```
void jarvis_march(const struct vecset *in, struct vecset *out);
```

Partie 4 : Parcours de Graham

Le principe du parcours de Graham est de choisir le point de plus petite ordonnée B *(Bottom)* (en cas d'égalité, choisir le point de plus petite abscisse) puis de trier l'ensemble des points en fonction de l'angle que chacun des points fait avec l'axe des abscisses relativement à B (indice : `atan2(3)`). Ensuite, on examine successivement tous les points par groupe de trois. Si ces trois points forment un tournant à gauche, alors c'est que le deuxième point ne fait pas partie de l'enveloppe convexe donc on l'élimine. Et on recommence.

L'algorithme 2 décrit le parcours de Graham de manière formelle. Le résultat attendu $\mathcal{R}$ est vu comme une pile.

La complexité de cet algorithme se décompose ainsi : la détermination du point de plus petite ordonnée est en $O(n)$, le tri est en $O(n \log n)$ et le parcours final est en $O(n)$. Pour le parcours final, il faut noter que la boucle `while` interne ne peut pas être appelée plus de n fois sur tout le parcours puisqu'un point ne peut être enlevé qu'une seule fois. Au final, la complexité du parcours de Graham est de $O(n \log n)$.

Algorithm 2 Parcours de Graham

function ($\mathcal{S}$)
  $B \leftarrow$ lowest point in $\mathcal{S}$
  SORT($\mathcal{S}$)
  $F \leftarrow$ first point in $\mathcal{S}$
  PUSH($\mathcal{R}, B$)
  PUSH($\mathcal{R}, F$)
  for all $I \in \mathcal{S} \setminus \{B, F\}$ **do**
    $T \leftarrow$ top point of $\mathcal{R}$
    $S \leftarrow$ second point of $\mathcal{R}$
    while $|\mathcal{R}| \geq 2$ and (S, T, I) is a left turn **do**
      POP($\mathcal{R}$)
    end while
    PUSH($\mathcal{R}, I$)
  end for
  return $\mathcal{R}$
end function

Question 4.1 Donner le code d'une fonction qui implémente le parcours de Graham. Attention, de la même manière que précédemment, l'algorithme 2 passe sous silence un certain nombre d'aspects auxquels il faudra pendre garde dans votre code.

```
void graham_scan(const struct vecset *in, struct vecset *out);
```

Partie 5 : Enveloppe rapide

L'algorithme d'enveloppe rapide se déroule en deux étapes. Dans un premier temps, on va déterminer le point de plus petite abscisse A et le point de plus grande abscisse B. Puis on va calculer l'ensemble $\mathcal{S}_1$ des points qui sont à gauche de $\overrightarrow{AB}$ et l'ensemble $\mathcal{S}_2$ des points qui sont à gauche de $\overrightarrow{BA}$. Dans un deuxième temps, on va calculer pour un ensemble $\mathcal{S}$ qui se situe à gauche de deux points X et Y le point M le plus éloigné de la droite (XY) (ce point fait partie de l'enveloppe convexe). Puis, on calcule l'ensemble $\mathcal{S}_1$ des points qui sont à gauche de $\overrightarrow{XM}$ et l'ensemble $\mathcal{S}_2$ des points qui sont à gauche de $\overrightarrow{MY}$ et on recommence récursivement.

L'algorithme 3 décrit l'enveloppe rapide. QUICKHULL est le premier temps de l'algorithme tandis que FINDHULL est le second temps de l'algorithme.

La complexité de l'algorithme d'enveloppe rapide s'analyse de manière très similaire à l'algorithme du tri rapide. La partition des points se fait en $O(n)$ et, en moyenne, coupe l'ensemble en deux parts à peu près égales. D'où une équation de complexité : $C(n) = O(n) + 2C(n/2)$ et donc une complexité de $O(n \log n)$.

Question 5.1 Donner le code d'une fonction qui implémente l'enveloppe rapide.

```
void quickhull(const struct vecset *in, struct vecset *out);
```

Algorithm 3 Enveloppe rapide

function QUICKHULL($\mathcal{S}$)
  $A \leftarrow$ leftmost point in $\mathcal{S}$
  $B \leftarrow$ rightmost point in $\mathcal{S}$
  $\mathcal{S}_1 \leftarrow \varnothing$
  $\mathcal{S}_2 \leftarrow \varnothing$
  for all $I \in \mathcal{S} \setminus \{A, B\}$ **do**
    if I is on the left of $\overrightarrow{AB}$ **then**
      $\mathcal{S}_1 \leftarrow \mathcal{S}_1 \cup \{I\}$
    else
      $\mathcal{S}_2 \leftarrow \mathcal{S}_2 \cup \{I\}$
    end if
  end for
  $\mathcal{R}_1 \leftarrow$ FINDHULL($\mathcal{S}_1, A, B$)
  $\mathcal{R}_2 \leftarrow$ FINDHULL($\mathcal{S}_2, B, A$)
  return $\{A\} \cup \mathcal{R}_1 \cup \{B\} \cup \mathcal{R}_2$
end function
function FINDHULL($\mathcal{S}, X, Y$)
  if $\mathcal{S} = \varnothing$ **then**
    return $\varnothing$
  end if
  $M \leftarrow$ farthest point from line (XY)
  $\mathcal{S}_1 \leftarrow \varnothing$
  $\mathcal{S}_2 \leftarrow \varnothing$
  for all $I \in \mathcal{S} \setminus \{M\}$ **do**
    if I in on the left of $\overrightarrow{XM}$ **then**
      $\mathcal{S}_1 \leftarrow \mathcal{S}_1 \cup \{I\}$
    end if
    if I in on the left of $\overrightarrow{MY}$ **then**
      $\mathcal{S}_2 \leftarrow \mathcal{S}_2 \cup \{I\}$
    end if
  end for
  $\mathcal{R}_1 \leftarrow$ FINDHULL($\mathcal{S}_1, X, M$)
  $\mathcal{R}_2 \leftarrow$ FINDHULL($\mathcal{S}_2, M, Y$)
  return $\mathcal{R}_1 \cup \{M\} \cup \mathcal{R}_2$
end function

Partie 6 : Pilote

Il est conseillé de faire cette partie en premier, après avoir lu tout le sujet. Le pilote est un programme qui va permettre de faire appel aux algorithmes précédents. Le pilote récupère les points sur son entrée standard et doit envoyer le résultats sur sa sortie standard. Les ensembles de points (en entrée ou en sortie) seront envoyés de la manière suivante : en premier le nombre de points, puis les coordonnées de chaque point sur une ligne, séparées par un espace.

Voici un bout de code pour vous aider à implémenter le pilote et en particulier à lire l'entrée.

```
int main() {
  setbuf(stdout, NULL); // avoid buffering in the output

  char buffer[BUFSIZE];
  fgets(buffer, BUFSIZE, stdin);

  size_t count = strtol(buffer, NULL, 10);

  for (size_t i = 0; i < count; ++i) {
    struct vec p;

    fgets(buffer, BUFSIZE, stdin);

    char *endptr = buffer;
    p.x = strtod(endptr, &endptr);
    p.y = strtod(endptr, &endptr);

    // then do something with p
  }

  return 0;
}
```

Un générateur de points `hull-generator` vous est fourni. Ce programme vous permet de générer des points au hasard dans un carré de côté 1000 centré en $(0, 0)$. Il prend en paramètre le nombre de points à générer. Il donne ces points sur la sortie standard et les stocke également dans un fichier `points.log` de manière à pouvoir rejouer ces mêmes points.

Un visualiseur de points `hull-viewer` vous est fourni. Ce programme vous permet de visualiser l'ensemble de points ainsi que l'enveloppe convexe que vous avez calculée. Il prend en paramètre le pilote que vous aurez implémenté. Il récupère les points sur son entrée standard puis lance votre pilote et récupère son résultat et affiche le tout. Tout ce que le pilote écrit sur l'erreur standard se retrouve dans le fichier `hull.log`.

En admettant que votre pilote s'appelle `hull`, voici une ligne de commande qui vous permettra de tester votre pilote avec 100 points :

```
$ ./hull-generator 100 | ./hull-viewer ./hull
```

Considérations générales et évaluation

Le projet est à faire en binôme. Toutes les structures et fonctions sont à placer dans un seul fichier `hull.c`. Vous indiquerez le nom des binômes dans un commentaire en haut du fichier. Vous rendrez ce fichier sur MOODLE avant la date indiquée (aucun retard autorisé).

La note du projet tiendra compte des points suivants (liste non-exhaustive) :

— la complexité optimale des algorithmes proposés ;
— les commentaires dans le code source ;
— l'absence de corruption mémoire ;
— l'absence de fuites mémoire ;
— le respect des consignes.