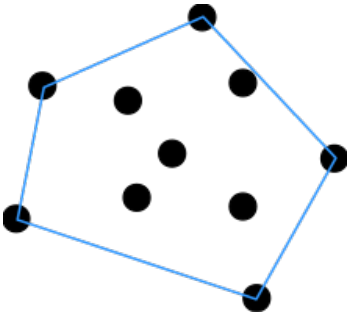
L'affichage de l'enveloppe convexe dans le plan euclidien usuel

Projet de Mathématiques



ESIREM
École Supérieure d'Ingénieurs de Recherche en Matériaux et en Infotronique

Auteurs :
Armen Mahmedov
Thibaut Malatier

Professeur :
Lionel Garnier

Introduction

Le projet est réalisé dans la cadre du module mathématique (ITC311) proposé par Lionel GARNIER.

Le but du projet est l'affichage de l'enveloppe convexe d'un nombre variable de points dans le plan euclidien. Pour réaliser le projet, nous devons tout d'abord définir les différents termes qui permettront de mieux appréhender le sujet. Ensuite dans une seconde partie nous verrons divers algorithmes avec des exemples qui permettent de calculer l'enveloppe convexe. Puis dans la dernière partie nous mettrons en pratique les éléments vu précédemment pour réaliser un programme en C++ avec la librairie OpenGL permettant tracer l'enveloppe convexe d'un nombre variable de points.

Table des matières

Table des figures

1 Théorie

Dans cette partie nous allons aborder les différents points théoriques sur les diverses notions pour mieux répondre a la problématique posée. Afin d'approcher notre problème, nous nous devons tout d'abord de rappeler certaines notions sur les espaces, puis nous verrons quelques définitions permettant de mieux comprendre la notion d'enveloppe convexe. Une partie de ces définitions à été developpé à partir de l'ouvrage de Frédéric Holweck et Jean-Noël Martin [1]

1.1 Espaces

Soit $\mathbb{K}$ un corps commutatif et E un ensemble non vide, munie d'une loi de composition interne, noté $+$ et d'une loi de composition externe, noté $\bullet$.

Définition 1. *Espace vectoriel*

$(E,+,\bullet)$ est un $\mathbb{K}$-espace vectoriel ou un espace vectoriel sur $\mathbb{K}$ si :

1. la loi $+$ vérifie les propriétés suivantes :
   - associativité :
   $$\forall(\vec{u},\vec{v},\vec{w}) \in E^3, (\vec{u}+\vec{v}+\vec{w} = \vec{u}+(\vec{v}+\vec{w}) = \vec{u}+\vec{v}+\vec{w}$$
   - commutativité :
   $$\forall(\vec{u},\vec{v}) \in E^2, \vec{u}+\vec{v} = \vec{v}+\vec{u}$$
   - symétrie :
   $$\forall\vec{u} \in E, \exists\vec{u'} \in E \mid \vec{u}+\vec{u'} = \vec{u'}+\vec{u} = \vec{0}$$
   - element neutre $\vec{0}$:
   $$\forall\vec{u} \in E, \vec{u}+\vec{0} = \vec{0}+\vec{u} = \vec{u}$$
2. la loi de composition externe vérifie les propositions suivantes :
   - $\forall\vec{u} \in E, 1 \bullet \vec{u} = \vec{u}$
   - $\forall\vec{u} \in E, \forall(k1,k2) \in \mathbb{K}^2$ on a :
     - $k1 \bullet (k2 \bullet \vec{u} = (k1 * k2) \bullet \vec{u}$
     - $(k1+k2) \bullet \vec{u} = k1 \bullet \vec{u} + k2\vec{u}$
   - $\forall(\vec{u},\vec{v}) \in E^2, \forall(k1 \in \mathbb{K}, k \bullet (\vec{u}+\vec{v}) = k \bullet \vec{u} + k \bullet \vec{v}$

Voir référence [2]

Définition 2. *Espace affine*

Soit V un espace vectoriel sur un corps $\mathbb{K}$ ($\mathbb{R}$ ou $\mathbb{C}$), un espace affine de direction V est un ensemble non vide E muni d'une application ϕ qui à chaque couple de point (A,B) de E^2, associe un élément de V, noté $\overrightarrow{AB}$ vérifiant les deux propriété suivantes :

$$\begin{aligned} \forall(A,B,C) \in E^3, \overrightarrow{AB} + \overrightarrow{BC} = \overrightarrow{AC} \\ \forall A \in E, \forall\overrightarrow{v} \in V, \exists! B \in E, \overrightarrow{AB} = \overrightarrow{v} \end{aligned} \tag{1}$$

1.2 Notion de convexe

Définition 3. *Convexe*

Une partie C de E est dite convexe lorsque deux points quelconques étant pris dans C, le segment qui les joint est entièrement contenu dans C.

$$\forall (x, y) \in C^2, \forall t \in [0, 1],\ tx + (1 - t)y \in C$$

Autrement dit un ensemble est convexe s'il contient tout droite passant par deux points.

Dans la figure 1 on peut apercevoir un ensemble convexe car on si on prend 2 point quelconque dans C le segment reliant les 2 point sera toujours dans C. Alors que dans la figure 2 si on prend 2 point comme décrit par la figure, le segment les reliant n'est pas dans C.

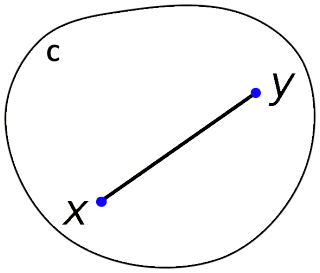


FIGURE 1 – Exemple d'un ensemble convexe

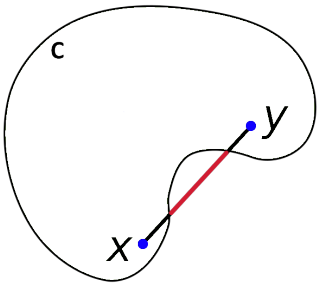


FIGURE 2 – Exemple d'un ensemble non convexe

Voir référence [4] et [5]

Définition 4. *Polygone convexe*

Un polygone convexe est un polygone simple dont l'intérieur est un ensemble convexe.

Pour un polygone simple, les propriétés suivantes sont équivalentes :
— le polygone est convexe
— les angles du polygone sont tous inférieurs à 180 degrés
— tout segment joignant deux sommets du polygone est inclus dans la composante fermée bornée délimitée par le polygone
— Le polygone est toujours entièrement inclus dans un demi-plan dont la frontière porte un côté quelconque du polygone.

Définition 5. *Barycentre*

On considère un espace affine A, d'espace vectoriel associé V sur le corps $\mathbb{R}$. Soit $A_1, A_2, \ ... \ , A_n$ des point de A et $\alpha_1, \alpha_2, \ ... \ \alpha_n$ des réels tels que la somme des réels est différentes de 0. Alors il existe un point G de A tel que :

$$\sum_{i=1}^{n} \alpha_i \overrightarrow{GA_i} = \vec{0}$$

Voir référence [1]

Définition 6. *Enveloppe convexe*

Soit A une partie de E. L'enveloppe convexe de A est l'intersection de toutes les parties convexes de E qui contiennent A. En d'autres termes, l'enveloppe convexe de E est le plus petit polygone convexe contenant E.

2 Application et présentation des algorithmes

Le calcul de l'enveloppe convexe d'un ensemble de points peut se faire en utilisant plusieurs algorithmes. En fonction de l'algorithme utilisé et du nombre de point la complexité de l'algorithme peut varier, impactant ainsi le temps d'exécution. Pour cela nous ferons une étude de quelques algorithmes peuvent nous convenir.

2.1 Marche de Jarvis

L'algorithme de 'Package-Wrapping', connu aussi sous le nom de la Marche de Jarvis, est un algorithme pour chercher les sommets de convexe. L'avantage de cette algorithme est qu'il est simple a comprendre et facile à réaliser car il simule une façon naturelle employé par un humain pour chercher un convexe. La possibilité de généraliser pour résoudre le problème dans un référentiel de plusieurs dimensions est un autre avantage de cet algorithme. Ici nous l'utiliseront dans le cas d'un plan 2D. L'inconvénient est qui est moins performant que d'autres algorithmes qui existes dans le pire scénario.

L'idée de l'algorithme est d'envelopper l'ensemble de points dans un papier cadeau on accroche ce papier à l'un des points, on le tend, puis on tourne autour du nuage de point. Prenons l'exemple de la figure 3 avec les points *P1, P2 ... P8*. On va détailler entièrement le fonctionnement de l'algorithme de Jarvis sur l'exemple de la figure 3.

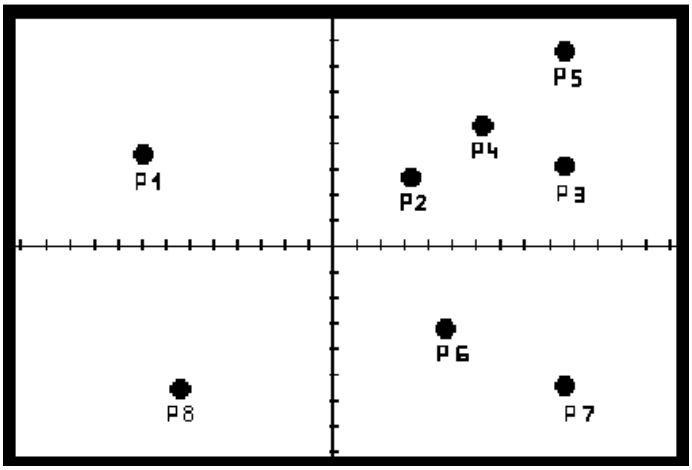


FIGURE 3 – Plan euclidien contenant des points

Algorithm 1 Marche de Jarvis

```
1: function MarcheJarvis(S)                                   ▷ S liste of point
2:     pointOnHull = leftmost point in S
3:     i = 0
4:     do
5:         P[i] = pointOnHull
6:         endpoint = S[0]          ▷ initial endpoint for a candidate edge on the hull
7:         for j = 1 to |S| do
8:             if (endpoint == pointOnHull) or (S[j] is on left of line from P[i] to endpoint)
   then
9:                 endpoint = S[j]                ▷ found greater left turn, update endpoint
10:            end if
11:        end for
12:        i = i + 1
13:        pointOnHull = endpoint
14:    while (endpoint == P[0])                          ▷ wrapped around to first hull point
15: end function
```

On applique l'algorithme de Jarvis à notre plan de la figure 3 en passant en paramètre $S = [P1, P2, P3, P4, P5, P6, P7, P8]$. Le point le plus a gauche est $P1$. On va détailler dans la figure 4 le déroulement pour le point $P1$ puis on détaillera que dans un cas particulier.

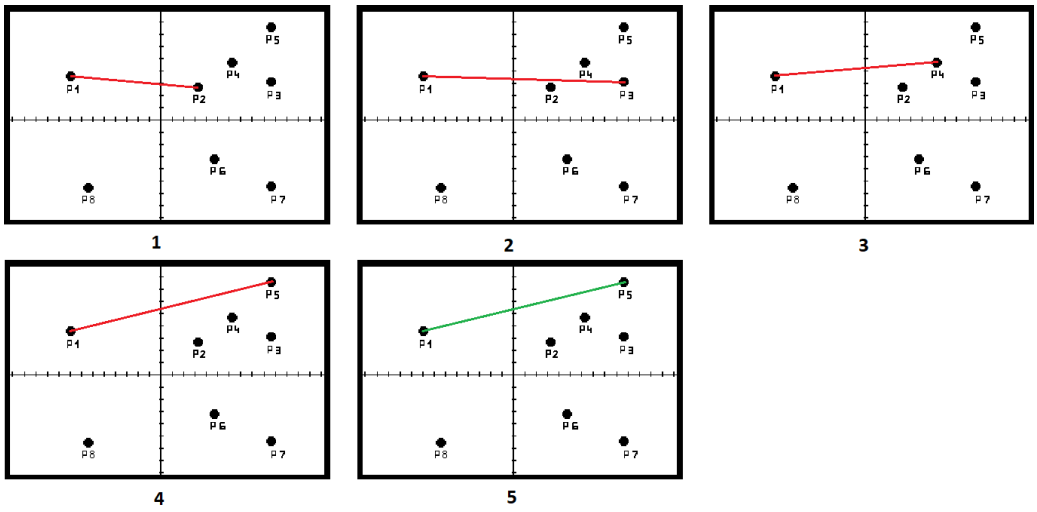


FIGURE 4 – Premier étape de l'algorithme de Jarvis

Dans la figure 4 partie 1, on voit que P2 est le prochain point, on va donc dire que cela

peut potentiellement être le prochain point. Ensuite on regarde le prochain point P3 et on se pose la question si P3 est a gauche de la ligne de P1 à P2, ici c'est le cas donc notre prochain point potentielle devient P3. (Pour trouver si un point est a gauche ou a droite d'un segment on va utiliser un produit vectoriel. Un exemple sera fait plus tard dans le rapport). Et ainsi de suite, on voit que P4 est a gauche de la ligne P1 à P3 donc P4 devient notre nouveau point potentielle. On regarde pour P5 qui est aussi a gauche, donc P5 deivne le nouveau point. Le prochain point est P6 qui n'est pas a gauche donc le point P5 fait bien partie de l'enveloppe convexe, et on trace la segment $[P1, P5]$ et P5 devient notre point de référence.

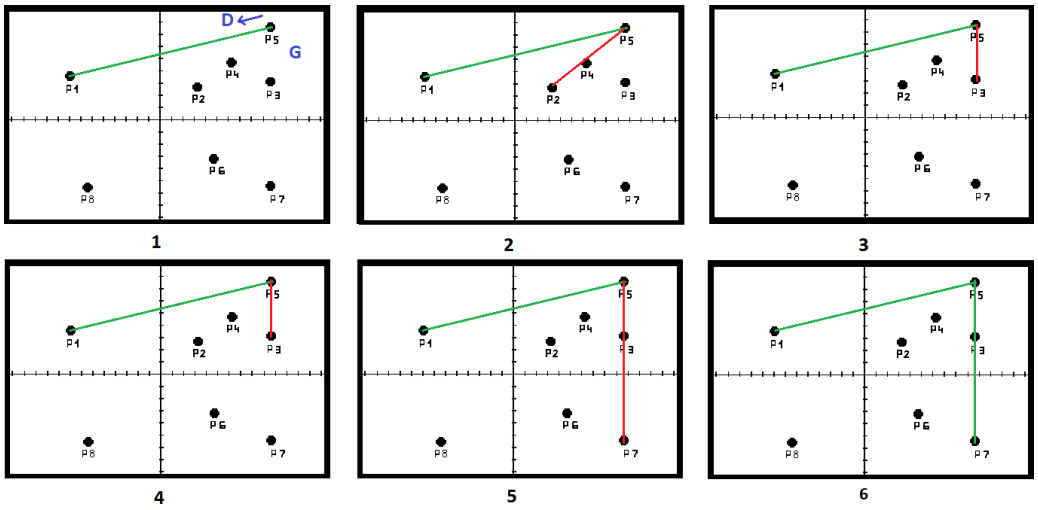


FIGURE 5 – Seconde étape de l'algorithme de Jarvis

Dans la figure 5 le D désigne la Droite et G la Gauche (partie 1). Maintenant on va regarder du point de vue de P5, le premier point est P1, ensuite le second point est P2 et se trouve a gauche du segment $[P5, P1]$, donc P2 peut faire partie de l'enveloppe convexe (partie 2). Le prochain point dans la liste est P3 qui se trouve a gauche donc on trace un trait rouge (partie 3), puis on regarde pour P4 qui se trouve a droite du segment $[P5, P3]$ donc on ne prend pas ce point (partie 4). Idem pour le Point P6 qui se trouve a droite. Si on regarde de P5, P7 et P3 sont aligné (colinéaires) (partie 5), ce cas la n'est pas prévu par l'algorithme, si on implémente cette algorithme il faudra prévoir ce as, nous allons donc prendre le point qui est le plus éloigné du point a partir duquel on est entrain de regarder (ici P5), ensuite on aura le choisir soit de mettre ou ne pas mettre le point P3 dans le tableau final des point qui feront partie de l'enveloppe convexe. On poursuit le déroulement de l'algorithme et on regarde si le dernier point (P8) est a gauche ou pas, ici ce n'est pas le cas, donc le point P7 fait partie de l'enveloppe convexe et on peut tracer le segment

$[P5, P7]$ (partie 6).

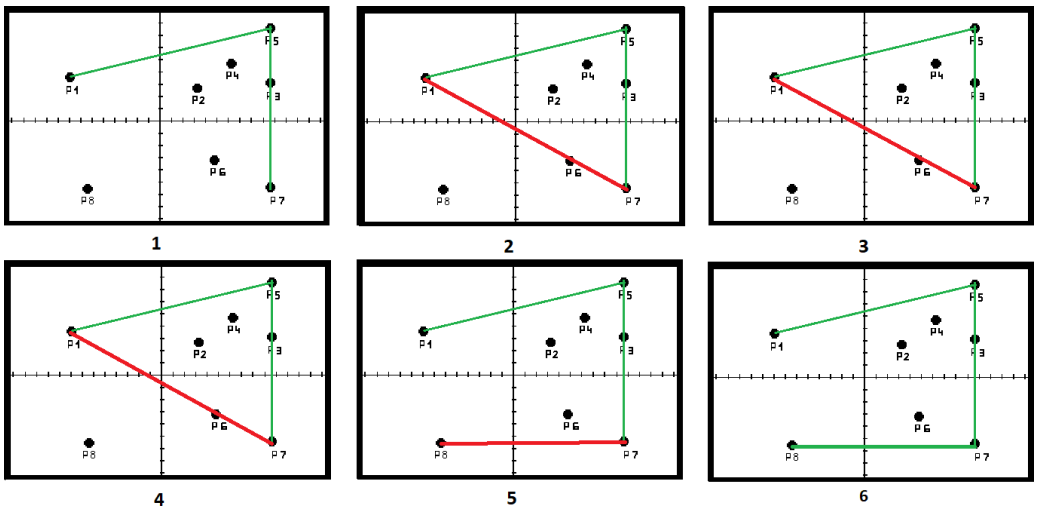


FIGURE 6 – Troisième étape de l'algorithme de Jarvis

Dans la figure 6 (partie 1) on prend le point de vue de P7. Le premier point la la liste est P1, et on regarde s'il y a des point a gauche du segment $[P7, P1]$ (partie 2). Les points P2, P3, P4 et P5 sont a droit donc on ne pas les prend en compte (partie 3). Ce pendant pour P6 on remarque qu'on est dans la même situation que dans la figure 5 partie 5, les point P1 et P6 sont colinéaire, on va donc regarder qui se trouve le plus loin de P7, P1 est le point le plus loin de P7 donc rien ne change (partie 4). Ensuite on regarde pour P8, le point se trouve a gauche du segment donc, il devient le nouveau point temporaire et vu qu'il ne reste plus de point on va tracer en vert le segment $[P7, P8]$ (partie 6).

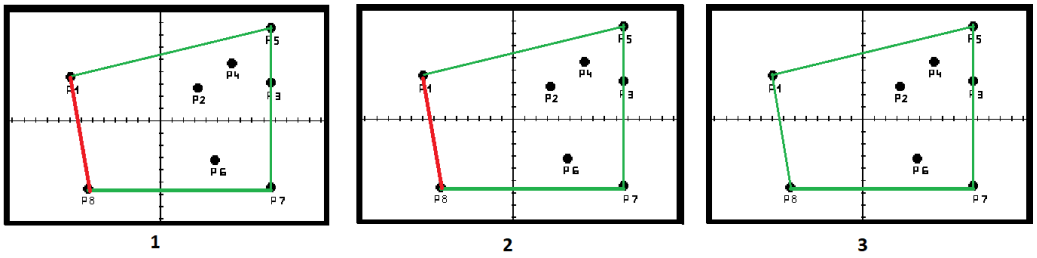


FIGURE 7 – Dernier étape de l'algorithme de Jarvis

Finalement dans la figure 7 on refait la même chose que précédemment. Le premier point est P1, on relie donc en rouge P8 à P1 (partie 1). Ensuite on regarde si les point qui

se trouvent dans le tableau S sont a gauche du segment, ici aucun point se trouve a gauche (partie 2) donc on peut relier définitivement en vert le segment $[P8, P1]$, et P1 devient le point de référence. Et vu que P1 est le point par lequel on a commencé cela veut dire que nous avons trouver les points qui forment l'enveloppe convexe.

Maintenant nous allons voir comment détecter si un point se trouve a gauche ou a droite d'un segment. Pour cela nous allons utiliser le produit vectoriel. Nous allons utiliser la figure 8 pour donner un exemple.

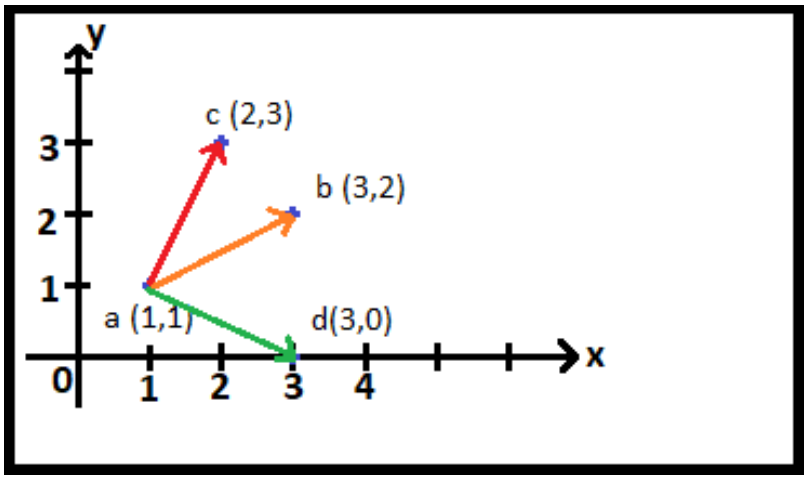


FIGURE 8 – Produit Vectoriel pour déterminer le point à gauche

On se met dans l'exemple où on a un tableau $S = [a, b, c, d]$ est on est au début du programme après avoir tracer le segment $[a, b]$ et on cherche a déterminer si le point c est a gauche ou a droite du segment. Pour le savoir on va faire un produit vectoriel entre les vecteurs $\overrightarrow{ab}$ et $\overrightarrow{ac}$.

$$\begin{aligned} x1 &= ax - bx = (1-3) = -2 \\ x2 &= ax - cx = (1-2) = -1 \\ y1 &= ay - by = (1-2) = -1 \\ y2 &= ay - cy = (1-3) = -2 \end{aligned} \tag{2}$$

$$\begin{aligned} \overrightarrow{ab} * \overrightarrow{ac} &= y_2 * x_1 - y_1 * x_2 \\ &= (-2) * (-2) - (-1) * (-1) \\ &= 4 - 1 \\ &= 3 \end{aligned} \tag{3}$$

Ici on obtiens que $\overrightarrow{ab} * \overrightarrow{ac} = 3 > 0$ donc le point c est a gauche de $[a, b]$.

Maintenant on va voir pour le point d. Pour cela on va faire le produit vectoriel entre $\overrightarrow{ab}$ et $\overrightarrow{ad}$

$$\begin{aligned} x1 &= ax - bx = (1 - 3) = -2 \\ x2 &= ax - dx = (1 - 3) = -2 \\ y1 &= ay - by = (1 - 2) = -1 \\ y2 &= ay - dy = (1 - 0) = 1 \end{aligned} \tag{4}$$

$$\begin{aligned} \overrightarrow{ab} * \overrightarrow{ac} &= y_2 * x_1 - y_1 * x_2 \\ &= (1) * (-2) - (-1) * (-2) \\ &= -2 - 2 \\ &= -4 \end{aligned} \tag{5}$$

Ici on obtiens que $\overrightarrow{ab} * \overrightarrow{ac} = -4 < 0$ donc le point d est a droite de $[a, b]$.

2.2 Parcours de Graham

Le parcours de Graham est un algorithme pour le calcul de l'enveloppe convexe d'un ensemble de points dans le plan.

Avant de pouvoir être appliqué, le Parcours de Graham requiert un pré-traitement sur l'ensemble des points. La première étape de cet algorithme consiste à rechercher le point de plus petite ordonnée. S'il y a égalité entre un ou plusieurs points, l'algorithme choisit parmi eux le point de plus petite abscisse. Nommons P ce point. La complexité en temps de cette étape est en $O(n)$, n étant le nombre de points de l'ensemble.

L'ensemble des points (P compris) est ensuite trié en fonction de l'angle que chacun d'entre eux fait avec l'axe des abscisses relativement à P. N'importe quel algorithme de tri convient pour cela, par exemple le tri par tas (qui a une complexité de $O(n \log(n))$).

Algorithm 2 Parcours de Graham

```
1:  procedure GRAHAM
2:      N= number of points
3:      points[N+1]
4:      swap points[1] with the point with the lowest y-coordinate
5:      sort points by polar angle with points[1]
6:      points[0] = points[N]
7:      M = 1
8:      for i = 2 to N do                          ▷ Find next valid point on convex hull.
9:          while (CrossProduct(points[M-1], points[M], points[i]) <= 0) do
10:             if M > 1 then
11:                 M -= 1                          ▷ All points are collinear
12:             else if i == N then
13:                 break
14:             else
15:                 i += 1
16:             end if
17:         end while                  ▷ Update M and swap points[i] to the correct place
18:         M = M + 1
19:         swap points[M] with points[i]
20:     end for
21: end procedure
```

Nous allons appliquer l'algorithme de Graham sur le plan de la figure 9. Cf [3]

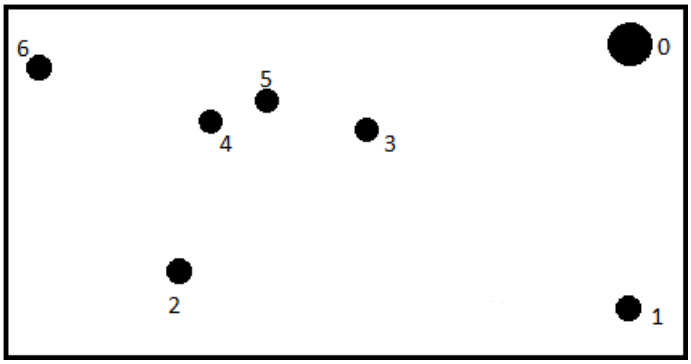


FIGURE 9 – Plan trié pour l'algorithme de Graham

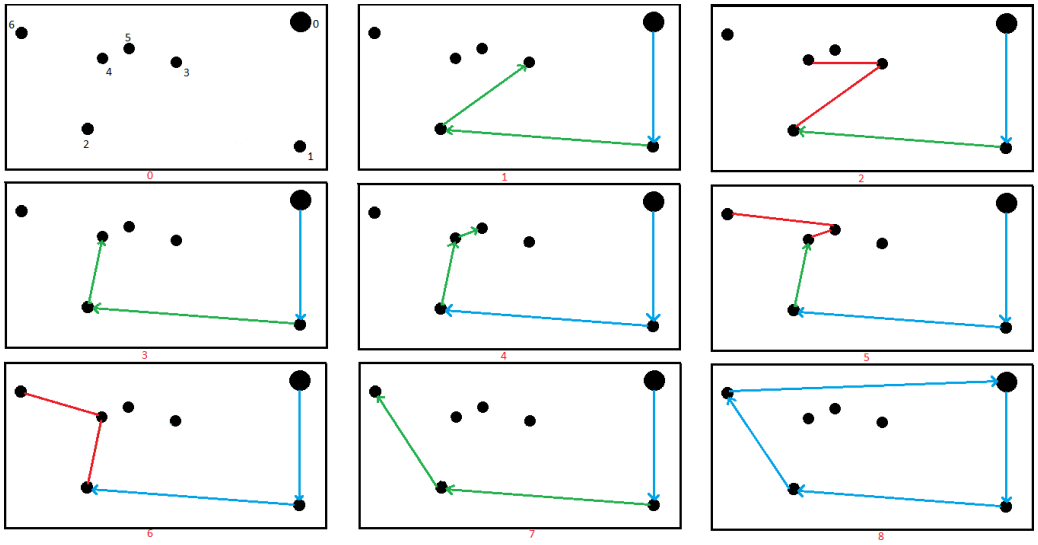


FIGURE 10 – Exe mple d'application de Graham

Comme on peut le voir dans la figure 10, on part du premier point et on avance dans la liste, trié en fonction des angles. On considère ensuite successivement les séquences de trois points contigus dans le tableau , vus comme deux couples successifs. Pour chacune de ces paires de couples, on décide si passer du premier couple au second constitue un « tournant à gauche » ou un « tournant à droite ». Si c'est un « tournant à droite », cela signifie que l'avant dernier point considéré (le deuxième des trois) ne fait pas partie de l'enveloppe convexe, et qu'il doit être rejeté du tableau.

2.3 Autres algorithmes

Il existe aussi d'autres algorithmes comme par exemple l'algorithme de Chan qui combine le parcours de Graham et la marche de Jarvis, l'aventage de cette algorithme est qui est rapide et s'étend facilement à la dimension 3. Il existe aussi l'algorithme Quickhull qui utilise le principe du diviser pour régner avec une approche similaire a l'algorithme du tri rapide, on divise le plan en 2 et on applique l'algorithme sur les 2 parties du plan.

3 Programmation

Dans cette partie nous allons aborder de l'implémentation d'un programme C++ avec la librairie OpenGL, qui permet de calculer l'enveloppe convexe d'un ensemble de point.

3.1 Installation des différents dépendances du projet

Pour pouvoir compiler et exécuter le programme il faut d'abord installer les dépendances nécessaires.

```
build-essential freeglut3-dev libjpeg-dev libxi-dev libxmu-dev
```

Après l'instalation de OpenGl il suffit exécuter le commande **make** pour compiler le programme ou **make run** pour compiler et lancer directement le programme, on peut également lancer le programme après avoir compiler en exécutant la commande **./convexe**.

Le code complet sera fournis en pièce joint a ce document mais il peut également le consulter sur cette page github

3.2 Ajout des classes

Avant de commencer le programme. On va créer des classes pour simplifier l'écriture du code. Pour cela nous avons 2 classes. Une classe dPoint et une classe ColorPoint qui hérite de dPoint, comme on peut le voir dans le figure 11. Vu que pour créer un ColorPoint on doit spécifier a chaque fois la couleur et la taille, on va créer un constructeur avec des attributs par défaut pour simplifier la création d'un point.

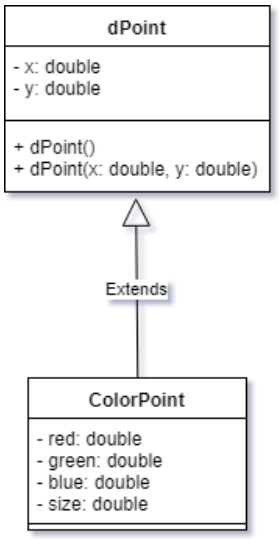


FIGURE 11 – Diagramme de classe des Points

```
ColorPoint(double _x, double _y, double _red = 0.0, double _green = 0.0,
    double _blue = 0.0, double _size = 10.0);
```

Ensuite on va créer un namespace ConvexeHullAlgorithms qui pourra contenir tous les algorithmes qui seront implémenter pour calculer l'enveloppe convexe. L'entête du namepsace est disponible dans ci-dessus.

```
namespace ConvexeHullAlgorithms
{
  /**
   * @description: Prend un tableau de Point, calcule et return les points
    qui appartiennent a l'enveloppe convexe en utilisant l'algorithme du
    Marche de Jarvis
   * @params:
   *    vector<ColorPoint> pointList : Liste des points dont il faut calculer
    l'enveloppe convexe
   * @return:
   *  un tableau de point qui font partie de l'envelopppe convexe
  */
  vector<ColorPoint> Jarvis(vector<ColorPoint> pointList);
};
```

Code 1 – Classe ConvexeHullAlgorithms

Pour finir nous allons aussi créer un namespace Utility pour placer des fonctions qui pourraient être utile pour le programme, comme par exemple une fonction qui calcule la distance entre deux point ou encore une fonction qui fait le produit vectorielle entre 3 point.

```
namespace Utility
{

  int distance2D(ColorPoint p1, ColorPoint p2);

  int crossProduct(ColorPoint p1, ColorPoint p2, ColorPoint p3);

  int getIntRandomNb(int min, int max);

  float getFlatRandNb(float min, float max);

  void printArrayPoints(std::vector<ColorPoint> points);

  void writeConvexePointsToFIle(std::vector<ColorPoint> points);

  void writePointsFormatedForGeogebra(std::vector<ColorPoint> points);

};
```

Code 2 – Utility fonctions

3.3 Explication du code de l'algorithme implémenté

Nous avons choisi d'implémenter l'algorithme **Marche de Jarvis**, en le modifiant pour mieux correspondre rependre a la problématique.

```
1  vector<ColorPoint> ConvexeHullAlgorithms::Jarvis(vector<ColorPoint> pointList
       )
2  {
3    vector<ColorPoint> result;
4    int length = pointList.size();
5    int startIndex;
6
7    // Trouver le point le plus a gauche
8    ColorPoint firstPts = pointList.at(0);
9    startIndex = 0;
10   for (int i = 1; i < length; i++)
11   {
12     if (pointList.at(i).getX() < firstPts.getX())
13     {
14       firstPts = pointList.at(i);
15       startIndex = i;
16     }
17     else if(pointList.at(i).getX() == firstPts.getX()) // si 2 pts ont le
       meme x alors on compare les Y et on prend le plus petit
18     {
19       if(pointList.at(i).getY() < firstPts.getY())
20       {
21         firstPts = pointList.at(i);
22         startIndex = i;
23       }
24     }
25   }
26   result.push_back(firstPts);
27
28   int precIndex = startIndex;
29   int qi;
30
31   do
32   {
33     qi = (precIndex + 1) % length;
34
35     for (int i = 0; i < length; i++)
36     {
37       // ne pas tester quand on doit prendre 2 fois le meme point
38       if (i == precIndex || i == qi)
39       {
40         continue;
41       }
42
43       if (Utility::crossProduct(pointList.at(precIndex), pointList.at(i),
```

```
        pointList.at(qi)) < 0)
44          {
45            qi = i;
46          }
47          if (Utility::crossProduct(pointList.at(precIndex), pointList.at(i),
        pointList.at(qi)) == 0) // les point sont colinéaires
48          {
49            if (Utility::distance2D(pointList.at(precIndex),pointList.at(i)) >
        Utility::distance2D(pointList.at(precIndex),pointList.at(qi))) // on
        prend le point qui est le plus loin du point de rotation
50            {
51              qi = i;
52            }
53          }
54        }
55
56        result.push_back(pointList.at(qi));
57
58        precIndex = qi;
59
60      } while (precIndex != startIndex);
61
62      int memeCoord=0;
63      int colli=0;
64      for (int i = 0; i < result.size()-1; i++)
65      {
66        if(result.at(i) == result.at(i+1))
67        {
68          memeCoord++;
69        }
70      }
71
72      // si tous les points ont la meme coordonées OU si tous les points sont
          alignés
73      if(memeCoord == result.size()-1 || (result.size()-1 <= 2))
74      {
75        result.clear();
76        cout << "Tous les point ont le meme coordonnée ou sont tous alignés" <<
          endl;
77      }
78
79      return result;
80    }
```

Code 3 – Utility fonctions

Tout d'abord la fonction prend en paramètre un tableau de ColorPoint et retourne un tableau qui contiens les points appartenant à l'enveloppe convexe. Dans un premier temps

on parcours tout le tableau et on recherche le point qui a le plus petit abscisse, en cas ou 2 point ont le même abscisse, on regarde leurs ordonné et on prend le point qui a le plus petit ordonné. Une fois le point trouvé on note son indice et on l'ajoute au tableau final qui va être retourné.

Pour mieux comprendre ce qui ce passe dans la boucle while, on va illustrer les première étapes de l'exécution de l'algorithme sur l'exemple simple présent dans la figure 12. Les nombres en haut des points représentent leur place dans le tableau passé en paramètre.

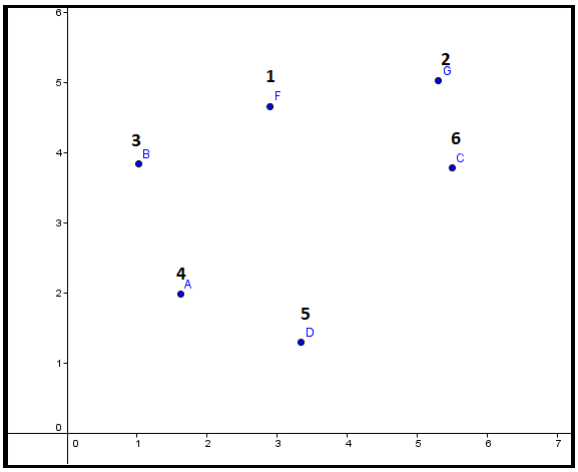


FIGURE 12 – Exemple d'illustration pour l'algorithme

Donc si on exécute l'algorithme on va arriver a cette étape. On a l'index du point le plus a gauche qui est stocké dans la variable ***precIndex***. La varible ***qi*** va contenir le point suivant dans la liste, ici 4 car $qi = (3+1)\%6$, en se trouve dans la situation de la figure 13 .On va ensuite entrer dans la boucle for et parcourir tous les points dans l'ordre. Pour i = 1 on calcule le produit vectorielle entre le point ***precIndex*** le point ***i*** et le point ***qi***. Si la valeur est négative donc le point se trouve a gauche, on va changer la valeur de ***qi*** et l'affecter la valeur de ***i***, comme on peut le voir dans le figure 14. Ainsi de suite jusqu'à la fin de la boucle for. Le pont qui se trouve à l'index qui a la fin de la boucle sera ajouté dans le tableau résultat et la valeur de ***precIndex*** devient ***qi***. Et on recommence les tests en incrémentant la valeur de qui ***qi*** tant que la variable ***precIndex*** est différent de ***startIndex***.

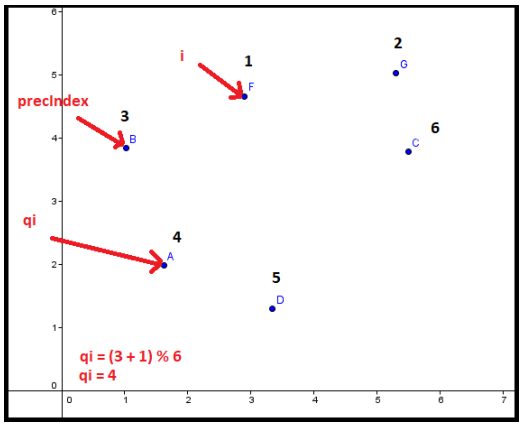


FIGURE 13 – Situation initiale

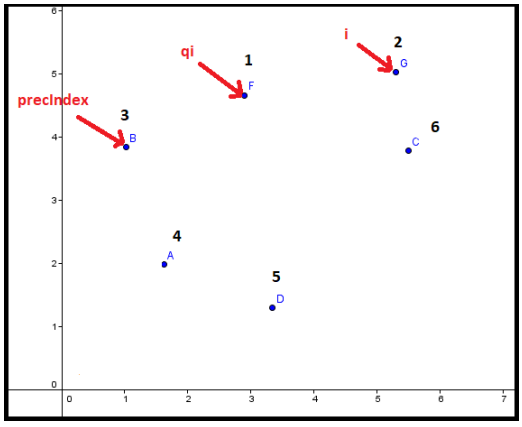


FIGURE 14 – Changement de point

Comment sont géré les cas particuliers dans l'algorithme ?

Dans la cas ou 3 points sont aligné, on regarde la distance entre le point de départ et les deux autres points et on prend celui qui est le plus point du point de départ (dans le programme le point a l'index precIndex).

Ensuite dans le *Code 3 a la ligne 64*, on compte le nombre de points qui ont les même coordonnés. Ensuite on test si tous les points entrés par par l'utilisateur ont les même coordonnés ou si l'utilisateur a entré que des points qui sont alignés, si c'est le cas alors on vide le tableau et on affiche un message.

3.4 Résultats

Au lancement du programme l'utilisateur a le choix entre 2 options. Générer les poins de manière aléatoire ou saisir les point soit même. La premier option va générer 150 points avec des coordonnées aléatoire entre -130 et 130. Pour la seconde option il faut entrer le nombre de points qu'on souhaite saisir et ensuite entrer les coordonnées des points un par un, comme on peut le voir dans la figure 15.

```
Choisir une option
1 - Points génerés aleatoirement
2 - Entrer les points a la main
2
Combien de points voulez vous saisir (> 3)
5
Saisir les coordonnées de cette façon : #Point#i: x espace y
- Par exemple pour saisir un point avec les coordonées x=2.3 et y=4.5 il faut faire: 2.3 4.5
Point #0:
```

FIGURE 15 – Menu de lancement

Après le calcule de l'enveloppe convexe, les points sont affichés à l'écran et les points

qui appartient à l'enveloppe sont stockés dans le fichier **convexePoints.txt**, qui est réinitialiser au début de chaque programme. Voici quelques résultats obtenu avec e programme, avec quelques cas particuliers.

— Exemple de enveloppe convexe calculé avec l'option 1 qui génère 150 points de manière aléatoire. Le résultat se trouve dans la figure 16 et le contenu du fichier convexePoints.txt .

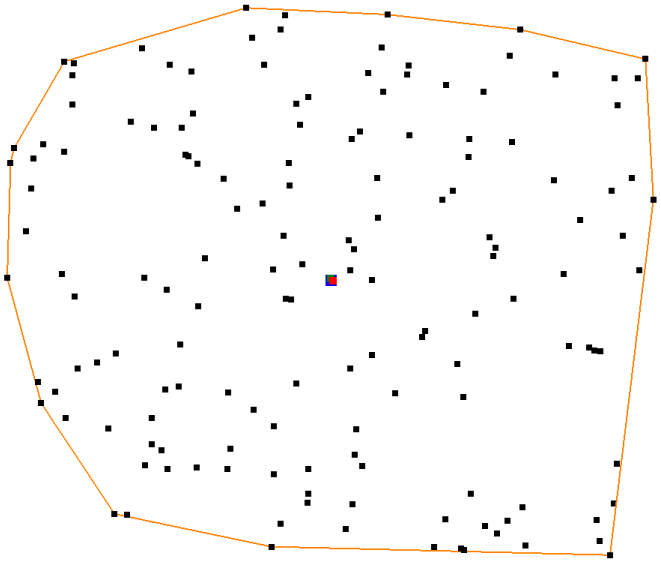


FIGURE 16 – Enveloppe convexe d'un ensemble de points généré aléatoirement

```
Les points qui appartiennent a l'enveloppe convexe :
#0 : x = -129.598 y= 1.35942
#1 : x = -115.984 y= -57.885
#2 : x = -86.6366 y= -110.24
#3 : x = -23.8471 y= -125.738
#4 : x = 111.864 y= -129.694
#5 : x = 129.234 y= 38.3084
#6 : x = 126.067 y= 104.831
#7 : x = 75.7893 y= 118.729
#8 : x = 22.7697 y= 125.861
#9 : x = -33.8889 y= 129.004
#10 : x = -106.655 y= 103.683
#11 : x = -126.883 y= 62.6505
#12 : x = -128.254 y= 55.745
```

— Dans la figure 17 on peut trouver le cas ou 3 points sont colinéaires. Ici les points P(2,0), P(3,0) et P(6,0). Et comme on peut voir en sortie dans le fichier convexe-Points.txt, le point P(3,0) ne fait pas partie de l'enveloppe convexe.

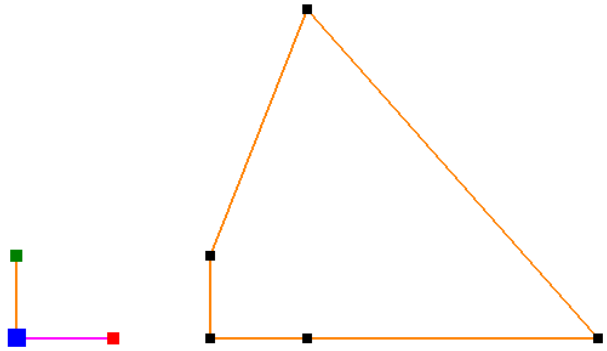


FIGURE 17 – Cas particulier de points colinéaires

```
Les points qui appartiennent a l'enveloppe convexe :
#0 : x = 2 y= 0
#1 : x = 6 y= 0
#2 : x = 3 y= 4
#3 : x = 2 y= 1
```

— Le cas ou tous les points entrés ont les même coordonnées présent dans la figure 19. Ici on a pris 5 poins avec les coordonnées (1,1). En sortie on a une erreur dans la console, qui nous dit que tous les points ont les même coordonnées.

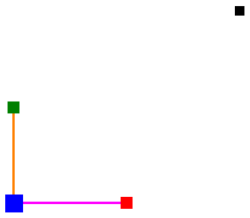


FIGURE 18 – Cas particulier de points avec tous les même coordonnées

— Le cas ou tous les points sont alignés

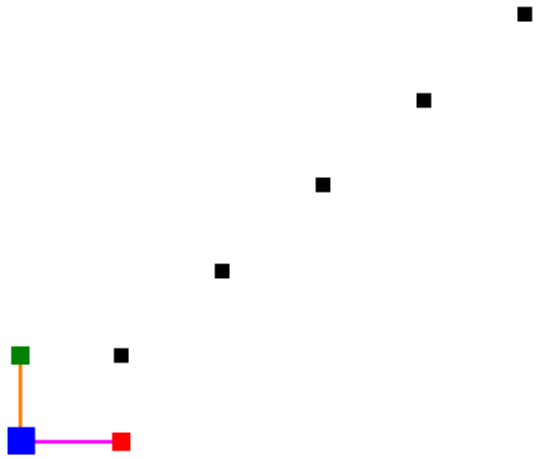


FIGURE 19 – Cas particulier des points tous alignés

Références

[1] Frédéric Holweck et Jean-Noël Martin. *Géométries pour l'ingénieur* (2013)

[2] Cours Lionel GARNIER, ESIREM

[3] Convex Hull : Lien du Pdf

[4] Cours de l'université Angers : Lien du Pdf

[5] Cours prépas Dupuy de Lôme : Lien site