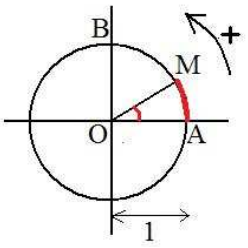
Angles, Orientation
Enveloppe convexe

On distingue deux types d'angles, les angles non orientés et les angles orientés. Un angle non orienté s'écrit le plus souvent en degrés, comme pour les angles d'un triangle (la somme de ses angles vaut 180°), et il est compris entre 0 et 180°. Dans ce contexte, en se plaçant dans un triangle rectangle, le cosinus d'un angle est le côté adjacent divisé par l'hypoténuse, et le sinus le côté opposé divisé par l'hypoténuse.

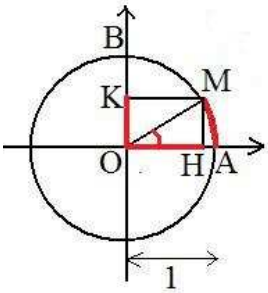
Un angle orienté s'écrit plutôt en radians et il se lit sur le cercle trigonométrique, un cercle de rayon 1 et de centre *O* (origine d'un repère orthonormé du plan), sur lequel est défini un sens positif (ou direct) qui est le sens inverse des aiguilles d'une montre.



L

Ainsi placé dans le cercle trigonométrique, l'angle orienté de vecteurs (**OA**, **OM**) a pour mesure en radians l'arc orienté *AM*. Mais lorsqu'on part de *A* pour arriver en *M* en circulant sur le cercle, on peut faire quelques tours complets supplémentaires dans un sens ou dans l'autre, et il existe une infinité d'arcs possibles. Comme la longueur du cercle est 2π, l'angle est défini à $2k\pi$ près (avec *k* entier positif ou négatif).

Rappels de trigonométrie



Appelons *a* l'angle (**OA**, **OM**). Avec le point *M* qui se projette en *H* et *K* sur les axes *Ox* et *Oy*, on a par définition :

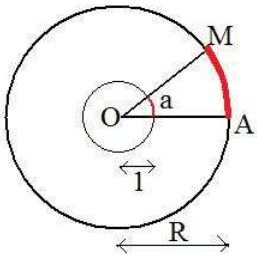
cos *a* = $\overline{OH}$ et sin *a* = $\overline{OK}$,

ces deux mesures algébriques étant comprises entre -1 et +1 (selon que *H* est à droite ou à gauche de *O*, et de même pour *K*). [1]

Dans ces conditions le point *M* a pour coordonnées : (cos *a*, sin *a*).

Nous verrons plus bas ce qu'est la tangente d'un angle.

Longueur d'un arc

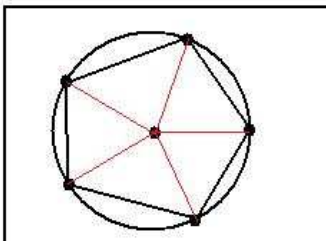


Grossissons le cercle trigonométrique en prenant un cercle de rayon *R*. Avec un angle *a* = (**OA**, **OM**), le point *M* a maintenant pour coordonnées (*R* cos *a*, *R* sin *a*). Et en supprimant les problèmes d'orientation, la longueur de l'arc *AM* est *R a*.

[1] En supprimant les problèmes de signe, on retrouve la définition du cosinus et du sinus pour un angle non orienté : il suffit de se placer dans le triangle rectangle *OHM*.

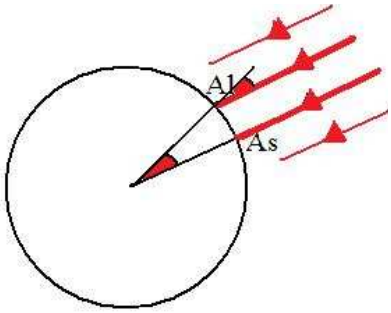
Exemple 1

Dans le repère de l'écran, on dispose un cercle de centre *O* (*xorig*, *yorig*) et de rayon *R*, et l'on place sur ce cercle des points régulièrement espacés de façon à obtenir un polygone régulier, comme ci-dessous pour *N* = 5 points. Pour avoir les coordonnées de ces points, on fait ce petit programme :



```
for(i=0 ; i<N ; i++)
  { x[i] = xorig + R*cos(2.*pi*i/(float)N ;
    y[i] = yorig – R*sin(2.*pi*i/(float)N ;
  }
```

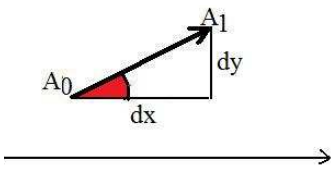
Exemple 2 : Eratosthène et la sphère terrestre



Voici comment, il y a deux mille trois ans en Egypte, Eratosthène mesura le rayon de la Terre. Il avait constaté qu'à Assouan (qui s'appelait Syène à l'époque), le jour du solstice d'été, à midi, le soleil était exactement à la verticale. En effet, à cet instant-là précisément, le soleil éclairait le fond d'un puits.

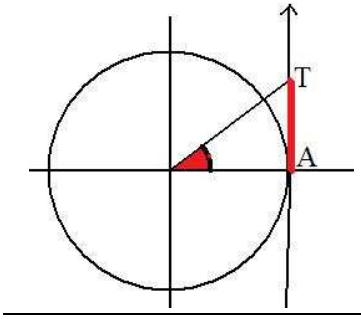
Au même moment, à Alexandrie, ville distante de quelques 800 km, et sur le même méridien qu'Assouan, le soleil était aussi au plus haut dans le ciel, mais faisait avec la verticale un angle *a* que l'on pouvait mesurer, soit 7,2° dans le cas présent, ou 7,2 π / 180 radians. Pour avoir le rayon *R* de la Terre, il suffisait de faire 800 = *Ra* et d'en déduire *R*. Eratosthène trouva ainsi la circonférence de la Terre, de l'ordre de 40 050 km, avec une erreur de l'ordre de 50 km. [2]

Angle d'un vecteur avec l'axe horizontal



Connaissant deux points A_0 et A_1 par leurs coordonnées dans un repère plan, comment avoir l'angle orienté que fait le vecteur $\mathbf{A_0A_1}$ avec l'axe horizontal, c'est-à-dire l'angle qui va de l'horizontale tracée à partir de A_0 jusqu'à $\mathbf{A_0A_1}$?

Pour le calculer, nous allons utiliser la tangente de l'angle, qui est liée au rapport entre la variation verticale *dy* (nombre positif ou négatif) et la variation horizontale *dx* correspondante (elle aussi positive ou négative selon les cas de figure.



Par définition, la tangente d'un angle *a*, soit tan *a*, est égale à :

sin *a* / cos *a*, ou encore égale à $\overline{AT}$ (voir dessin ci-contre du cercle trigonométrique de rayon 1). Lorsqu'un angle varie de $-\pi/2$ à $\pi/2$, sa tangente varie de $-\infty$ à $+\infty$. Inversement, pour une tangente donnée (c'est-à-dire un point *T* donné), il existe deux angles : si l'un est *a* l'autre est $a + \pi$.

[2] A ce sujet, et de bien d'autres, on pourra lire avec profit les trois volumes de la *Civilisation grecque*, d'André Bonnard, un des rares livres sur le sujet qui ne soit pas une apologie oiseuse de l'Occident.

La fonction *atan* (ou *Arctan*), inverse de la fonction tangente, prend comme variable un nombre qui est la tangente d'un angle et ramène l'angle correspondant, en choisissant celui qui est compris entre $-\pi/2$ à $\pi/2$. Par exemple *Arctan* 1 est l'arc (ou l'angle) dont la tangente vaut 1, d'où *Arctan* $1 = \pi/4$.

Nous allons calculer l'angle orienté de $\boldsymbol{A_0A_1}$ avec l'axe horizontal, en degrés de préférence. Sa valeur sera comprise entre 0° et 360°. Pour cela, on détermine la variation *dx* horizontale et la variation *dy* verticale. On traite d'abord le cas où la tangente de l'angle n'existe pas, lorsque $dx = 0$, ce qui donne soit 90° soit 270°. Puis on calcule dy/dx. Au cas où *dx* et *dy* sont positifs, dy/dx est l'arctangente de l'angle cherché, en radians (entre 0 et $\pi/2$). Il suffit de le multiplier par $180/\pi$ pour l'avoir en degrés. Mais si l'on a $dx > 0$ et $dy < 0$, l'arctangente précédent donne un angle négatif entre $-\pi/2$ et 0. On doit lui ajouter un tour complet de 360° pour avoir sa valeur dans la zone entre 0 et 360. Enfin si *dx* est négatif, on doit ajouter à l'arctangente (compris entre $-\pi/2$ et $+\pi/2$) un demi-tour de 180°. D'où le programme donnant l'angle en degrés :

```
float angle(int x0, int y0, int x1, int y1)
{   float a, dx, dy;
    dx=x1-x0; dy=y1-y0;
    if (dx==0.) if (dy>0) a=90.; else a = 270.;
    else a= atan(dy/dx)*180./pi;
    if (dx>0. && dy<0.) a+=360;
    else if (dx<0.) a+=180.;
    return a;
}
```

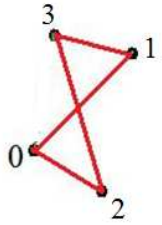
Exercice 1 : Construction d'un polygone simple à partir de N points

Un polygone simple est un polygone qui délimite une surface, ses côtés ne se recoupent pas, et il peut avoir des angles saillants ou rentrants. Dans le contexte de l'exercice, N points sont donnés. Trouver un chemin en forme de polygone qui passe par chacun de ces points une fois et une seule et revient au point de départ, sans jamais que les segments tracés entre les points successifs ne se recoupent. On appelle cela un chemin polytonal simple fermé.

Polygone simple

Polygone non simple

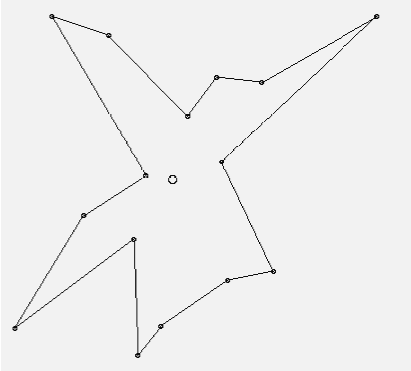
Voici une première proposition de méthode. On choisit un des points comme point de départ, soit A_0. Puis, en prenant touts les autres points A_i, on détermine l'angle des N - 1 vecteurs $\boldsymbol{A_0\ A_i}$ avec l'horizontale. En classant ces angles par ordre croissant, ainsi que les points A_i correspondants, la jonction de ces points dans cet ordre, commençant par A_0 pour revenir en A_0 donne un chemin polygonal simple. Mais cela est faux. Il peut en effet arriver que certains polygones se recoupent, comme dans l'exemple suivant :



A partir du point 0, les autres points classés selon leurs angles croissants avec l'horizontale (entre 0 et 360°) sont 1, 3, 2. Lorsque l'on joint les points dans l'ordre 0 1 3 2 0, on trouve un polygone non simple.

Rectifions l'algorithme. On est amené à prendre le centre de gravité *G* des *N* points, puis à classer les *N* vecteurs $\boldsymbol{GA_i}$ selon leurs angles avec l'horizontale. En tournant autour du centre de gravité dans l'ordre des angles croissants, il n'y a plus aucun recoupement.

```
cumulx=0; cumuly=0;
for(i=0;i<N;i++) { p[i]= i;    x[i]=200+rand()%400; y[i]=100+rand()%400;
                   cumulx+=x[i]; cumuly+=y[i];
                  }
 xg=cumulx/N; yg=cumuly/N;
 for(i=0;i<N;i++) alpha[i]=angle(xg,yg,x[i],y[i]);
for(i=0;i<N-1;i++) for(j=i+1; j<N; j++)
if(alpha[i]>alpha[j]) { aux=p[i]; p[i]=p[j];p[j]=aux;
                        auxf= alpha[i]; alpha[i]=alpha[j]; alpha[j]=auxf;
                       }
 for(i=0;i<N; i++) line(x[p[i]],y[p[i]],x[p[(i+1)%N]],y[p[(i+1)%N]],rouge);
```

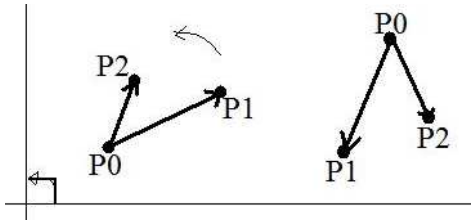


Comment distinguer la gauche de la droite ?

Dans un plan, lorsque l'on trace un vecteur, ou une droite orientée, comme un rayon lumineux, le plan est partagé en deux demi-plans, l'un à gauche et l'autre à droite. Pour faire la distinction entre gauche et droite, on utilise la règle du déterminant, que voici.

Soit trois points P_0 P_1 P_2, dans un repère orthonormé direct (on passe de Ox à Oy en tournant de $+\pi/2$). Le point P_2 est à gauche du vecteur $\mathbf{P_0P_1}$ -on regarde de P_0 vers P_1, si et seulement si le déterminant $dx1.\ dy2 - dy1.\ dx2$ des vecteurs $\mathbf{P_0P_1}$($dx1$, $dy1$) et $\mathbf{P_0P_2}$ ($dx2$, $dy2$) est positif, et à droite dans le cas contraire.[3] Rappelons qu'avec les points P_0 (x_0, y_0), P_1 (x_1, y_1), P_2 (x_2, y_2) donnés par leurs coordonnées, on a :

$dx1 = x_1 - x_0$, $dy1 = y_1 - y_0$ et $dx2 = x_2 - x_0$, $dy2 = y_2 - y_0$



Ici P_2 est à gauche de P_0P_1 (ou encore le triangle P_0 P_1 P_2 est direct, on parcourt le chemin $P_0P_1P_2$ en tournant dans le sens inverse des aiguilles d'une montre).

[3] Appelons a_1 l'angle que fait $\boldsymbol{P_0P_1}$ avec Ox et a_2 celui de $\boldsymbol{P_0P_2}$, et notons $d1$ et $d2$ les longueurs de ces deux vecteurs. P_2 est à gauche du vecteur $\boldsymbol{P_0P_1}$ ssi l'angle $a_2 - a_1$ est compris entre 0 et π, ou encore $\sin(a_2 - a_1)>0$. Or

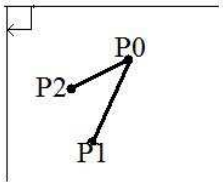
$$\sin(a_2 - a_1) = \sin a_2 \cos a_1 - \sin a_1 \cos a_2 = \frac{dy2}{d2}\frac{dx1}{d1} - \frac{dy1}{d1}\frac{dx2}{d2} = \frac{\text{déterminant}}{d1\,d2}.$$

Remarquons que le déterminant des deux vecteurs ($\mathbf{P_0P_1}$ suivi de $\mathbf{P_0}$ $\mathbf{P_2}$) est aussi le produit de leurs longueurs $d1$ et $d2$ par le sinus de l'angle qu'ils font entre eux (de $\mathbf{P_0P_1}$ vers $\mathbf{P_0P_2}$). Il s'agit aussi de l'aire (algébrique, avec un signe + ou -) du parallélogramme constitué par les deux vecteurs.

On en déduit la fonction *gauche*() qui ramène oui ou non selon que le triangle $P_0\ P_1\ P_2$ est direct:

```
int gauche (int x0, int y0, int x1, int y1, int x2, int y2)
  {  dx1=x1 –  x0 ; dy1= y1 – y0 ; dx2 = x2 – x0 ; dy2 = y2 – y0 ;
     det= dx1*dy2 – dy1*dx2 ;
     if (det>=0) return OUI;  /* ici l'alignement est considéré comme à gauche */
     return NON;
  }
```

Selon les besoins, ce programme sera raffiné pour tenir compte des problèmes d'alignement. Remarquons aussi que dans le repère écran qui est celui de nos ordinateurs, avec l'axe des *y* dirigé vers le bas, le résultat est inversé : avec le déterminant positif, le point P_2 est maintenant à droite de $\mathbf{P_0P_1}$, mais si l'on remplace le vecteur par la droite (non orientée) (P_0P_1), le point P_2 reste quand même à gauche de la droite (P_0P_1) lorsque P_1 est au-dessous de P_0 comme sur le dessin ci-contre.



Comment savoir si un point est à l'intérieur d'un triangle ?

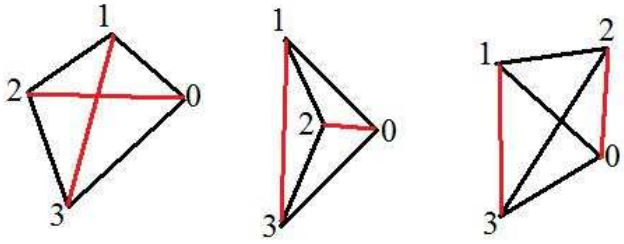
Considérons un triangle *ABC* supposé orienté dans le sens direct. Cela entraîne qu'en le parcourant dans le sens inverse des aiguilles d'une montre, son intérieur est toujours situé à gauche. Ainsi un point *I* sera intérieur au triangle s'il est à gauche du vecteur **AB**, à gauche de **BC** et à gauche de **CA**.

Comment savoir si deux segments se coupent ?

Deux segments [*AB*] et [*CD*] se coupent si *A* et *B* sont de part et d'autre de ***CD***, et si *C* et *D* sont de part et d'autre de **AB**. Remarquons qu'une seule de ces deux conditions n'est pas suffisante.

Comment savoir si un quadrilatère est convexe ?

Considérons un quadrilatère dont les sommets successifs sont numérotés 0 1 2 3. Il est dit convexe si ses angles sont tous saillants (entre 0 et 180°) ce qui signifie que ses diagonales 02 et 13, considérées comme des segments, se coupent. C'est là une propriété caractéristique. Il n'en est pas de même pour un quadrilatère non convexe, comme l'indiquent les dessins suivants :



(quadrilatères en noir, diagonales en rouge)

convexe simple non convexe non simple (croisé)

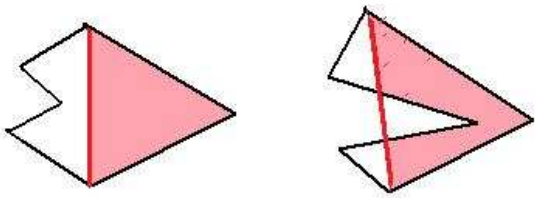
Voici un petit programme qui permet de construire un quadrilatère convexe au hasard (de sommets *xq*[] et *yq*[]), grâce à la fonction *convexe*() qui ramène oui (1) ou non (0) selon que les segments diagonaux se coupent ou ne se coupent pas :

```
/* programme principal */
 do   {for(i=0; i<4; i++) {xq[i]= rand()%200+100;  yq[i]=rand()%200+100; }
 while (convexe()==0);
 tracer le quadrilatère avec les lignes des points 0 à 1, de 1 à 2, de 2 à 3 et de 3 à 0

 _____
int convexe(void)
{  float det1,det2,det3,det4;
   det1=(xq[2]-xq[0])*(yq[1]-yq[0])-(yq[2]-yq[0])*(xq[1]-xq[0]);
   det2=(xq[2]-xq[0])*(yq[3]-yq[0])-(yq[2]-yq[0])*(xq[3]-xq[0]);
   det3=(xq[3]-xq[1])*(yq[0]-yq[1])-(yq[3]-yq[1])*(xq[0]-xq[1]);
   det4=(xq[3]-xq[1])*(yq[2]-yq[1])-(yq[3]-yq[1])*(xq[2]-xq[1]);
   if (det1*det2<0  && det3*det4<0) return 1;
   return 0;
}
```

Exercice 1 : Triangulation de polygones simples

Avec ses angles saillants et rentrants, un polygone simple possède une surface intérieure et c'est elle que nous allons découper en triangles. Une triangulation du polygone est un ensemble de triangles dont les sommets sont aussi des sommets du polygone et qui forment une partition de sa surface intérieure. L'idée de l'algorithme est d'enlever un triangle correspondant à un angle saillant, ses deux côtés étant des côtés du polygone, puis de recommencer avec le polygone simple amputé de ce triangle. Mais cette idée qui consiste à rogner les bosses, à couper tout ce qui dépasse, est fausse, dans la mesure où il peut arriver qu'un triangle possède un trou. Aussi doit-on raffiner l'algorithme. Nous allons définir la notion de protubérance.



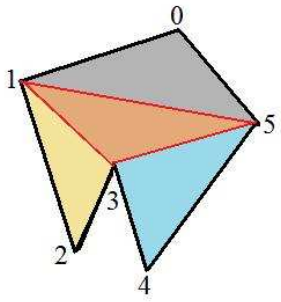
on peut enlever le triangle rose on ne peut pas l'enlever

Protubérance

Il s'agit d'un sommet *i* entouré de ses sommets voisins *i* - 1 et *i* + 1 (modulo le nombre *N* de sommets su polygone) tel que le triangle ainsi formé ait toute sa surface intérieure au polygone. Si un angle rentrant ne donne jamais une protubérance, un angle saillent ne donne pas forcément une protubérance, comme on vient de le voir. Un sommet du polygone est une protubérance si et seulement si l'angle est saillant et que le triangle associé ne contient aucun autre sommet du polygone. L'algorithme de triangulation consiste à couper les protubérances, les unes après les autres. On admettra les propriétés suivantes : tout polygone simple admet une triangulation. Si son nombre *N* de sommets est supérieur ou égal à 4, il admet au moins deux protubérances non consécutives. Toute triangulation d'un polygone simple à *N* sommets est formée de *N* – 2 triangles.

Algorithme

On part d'un polygone à *N* sommets, ceux-ci étant numérotés de 0 à *N* - 1 lorsque l'on parcourt le polygone dans le sens direct. Puisque la triangulation donne *N* - 2 triangles, la grande boucle du programme est répétée *N* - 2 fois. A chaque fois, en tournant sur le contour du polygone à partir du premier point, on cherche la première protubérance venue, ce qui donne un triangle, puis on l'enlève, et on recommence avec le nouveau polygone, qui possède un sommet de moins que le précédent. A chaque étape, on gère un tableau *A*[] contenant les sommets des polygones successifs, et c'est ce tableau de longueur *L* décroissante que l'on parcourt pour trouver la première protubérance, que l'on enlève ensuite. Comme dans l'exemple suivant où l'on voit l'évolution du tableau *A*[] :



étape 0 : 012345 $L = 6$ ($= N$)
étape 1 : 12345 $L = 5$
étape 2 : 1345 $L = 4$
étape 3 : 345 $L = 3$
étape 4 : 45 $L = 2$

Programmation

En conditions initiales on se donne les coordonnées *X*[] et *Y*[] des sommets du polygone, ceux-ci étant numérotés de 0 à N - 1, puis on remplit le tableau *A*[] avec les numéros des sommets du polygone, de 0 à N - 1, sur une longueur $L = N$. Les indices des cases du tableau *A*[] seront appelées indices de sommets. Le programme principal se réduit alors à :

```
for(i=0;i<N;i++) A[i]=i; L=N;
for(i=0;i<N-2;i++)   {  indiceprotu=chercherprotu();  creertriangle(); enleverprotu(); }
```

Passons aux fonctions :

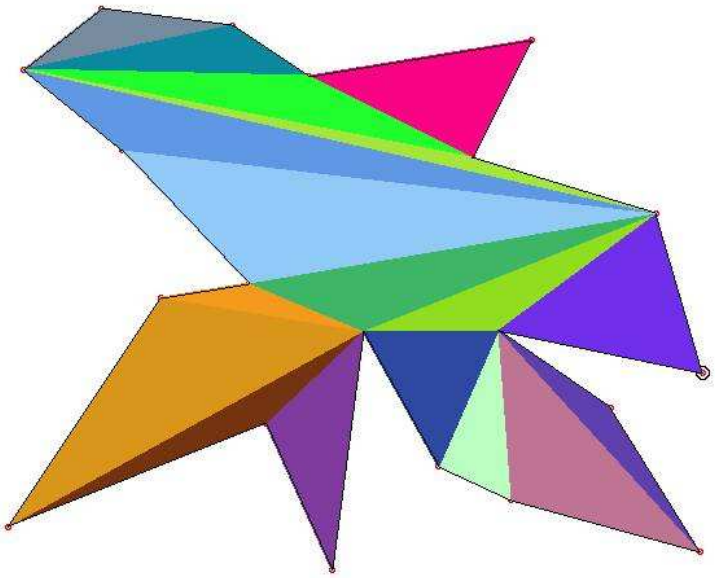
```
int chercherprotu(void)
{  int i;
   i=0;      while (protu(i)==NON && i<L) i++;
   return i;
}

int protu(int i)   /* i est un indice de sommet et non pas un sommet */
{ int a,b,c,j;
   a=A[(i-1+L)%L]; b=A[i]; c=A[(i+1)%L];   /* a, b, c sont des numéros de sommets */
   if (triangledirect(a,b,c)==NON) return NON;
   for(j=0;j<L;j++)  if (j!=i && j!=(i-1+L)%L && j!=(i+1)%L
                          && interieur(A[j],a,b,c)==OUI)
                          return NON;
   return OUI;
}

void enleverprotu(void)
{int k;
for(k=indiceprotu;k<L-1;k++) A[k]=A[k+1];
L--;
}

void creertriangle(void)
{
  xq[0]=X[A[(indiceprotu-1+L)%L]] ;yq[0]=Y[A[(indiceprotu-1+L)%L]] ;
  xq[1]=X[A[indiceprotu]];yq[1]=Y[A[indiceprotu]];
  xq[2]=X[A[(indiceprotu+1)%L]];yq[2]=Y[A[(indiceprotu+1)%L]];
  couleur=SDL_MapRGB(ecran->format,rand()%256,rand()%256,rand()%256);
  remplirtriangle(couleur);
}
```

La fonction *triangledirect*() est quasiment la même que la fonction *gauche*() vue plus haut. La fonction *interieur*() en est une conséquence immédiate, et la fonction *remplirtriangle*() sera donnée plus bas. Dans notre programme, nous avons pris l'axe vertical du repère dirigé vers le bas : dans ces conditions, le sens direct est inversé, c'est le sens des aiguilles d'une montre.



Un exemple de triangulation d'un polygone simple

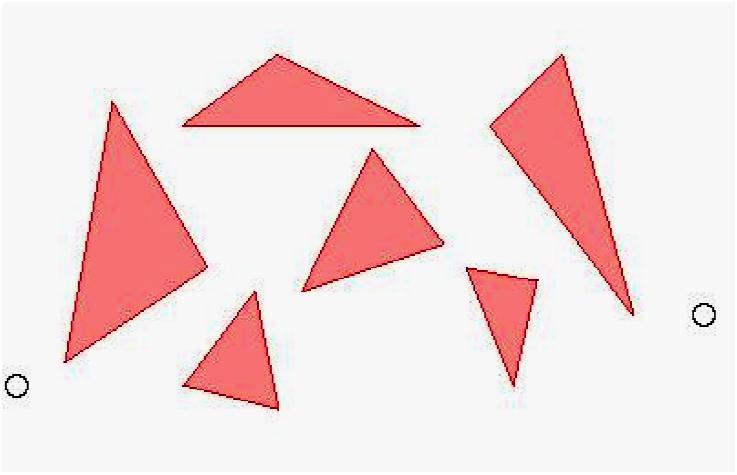
Exercice 2 : Déplacement d'un robot à travers des obstacles

Entre un point de départ D et un point d'arrivée A, un robot doit se frayer un chemin entre des obstacles en forme de triangles. L'objectif est de trouver le plus court chemin pour ce robot.

Si une ficelle joint les points *D* et *A* en passant entre certains obstacles bien précis, le fait de tendre cette ficelle donne un chemin plus court que les autres, et ce chemin se fait en ligne brisée, en s'appuyant sur certains sommets des triangles, voire en longeant certains côtés des triangles. La ficelle peut être tendue suivant plusieurs chemins entre *D* et *A*. Parmi tous ces chemins en ligne brisée, il s'agit de dégager celui qui est le plus court. Cela donne le principe de l'algorithme. Pour chaque sommet des triangles constituant les obstacles, ainsi que pour le point de départ et celui d'arrivée, on détermine les sommets qui sont visibles à partir de lui, c'es-à-dire les jonctions en ligne droite que l'on peut tracer sans passer au travers d'un obstacle. Cela donne un graphe dont les arcs sont des segments, et dont la longueur peut être calculée. Il reste ensuite à déterminer le plus court chemin de *D* à *A* dans ce graphe, ce que fait l'algorithme de Dijkstra. Passons à la programmation.

Plan d'ensemble

On commence par se donner le nombre de triangles (ici *nbtriangles* = 6), puis le nombre de points concernés, à savoir les sommets des triangles, allant de trois en trois à partir du sommet numéroté 1, ainsi que le point de départ *A*, numéroté 0, et le point d'arrivée *D* placé en dernier. Le nombre de ces points est *nbpoints* et il vaut 3**nbtriangles* + 2. Puis on entre les coordonnées *x*[] et *y*[] de ces points et on affiche le dessin, avec à gauche le point de départ (sous forme d'un rond) et à droite le point d'arrivée.



Ce dessin correspond à l'entrée des données suivantes, toutes étant en entiers (mais on peut faire un *zoom* supplémentaire si on le désire)

```
D=0 ; A=nbpoints-1 ;
x[D]=10; y[D]=20+25*cas;  x[A]=300; y[A]=50;
x[1]=50; y[1]=140;   x[2]=30; y[2]=30;  x[3]=90; y[3]=70;x[4]=120; y[4]=160;
x[5]=80; y[5]=130;   x[6]=180; y[6]=130; x[7]=160; y[7]=120;  x[8]=130; y[8]=60;
            x[9]=190; y[9]=80; x[10]=200; y[10]=70;  x[11]=220; y[11]=20;  x[12]=230; y[12]=65;
x[13]=110; y[13]=60;  x[14]=80; y[14]=20;  x[15]=120; y[15]=10; x[16]=240; y[16]=160;
 x[17]=210; y[17]=130;  x[18]=270; y[18]=50;
dessin();
```

Pour la fonction *dessin*(), nous utilisons pour la première fois la fonction de remplissage *floodfill*() qui permet de colorier l'intérieur de chaque triangle. Cette fonction prend comme variable un point (pour nous le centre de gravité *xg*, *yg* d'un triangle) et elle se charge de colorier la périphérie de ce point jusqu'à la rencontre de la bordure du triangle (ayant la couleur *rougebordure*). Tout se passe comme s'il y avait un écoulement d'un liquide coloré à partir d'une source, l'écoulement se faisant avec la couleur *rouge2* dans le cas présent). Le programme simplifié de cette fonction *floodfill*() se trouve dans le chapitre suivant (*Remplissage de surfaces*)

```
void dessin(void)
{  int i,xg,yg;
   cercle(zoom*x[D],480-zoom*y[D],6,noir); cercle(zoom*x[A],480-zoom*y[A],6,noir);
   for(i=0;i<nbtriangles;i++)
     {  xg=0.333*(float)(x[3*i+1] + x[3*i+2]+ x[3*i+3]);
        yg=0.333*(float)(y[3*i+1] + y[3*i+2] + y[3*i+3]);
        ligne(zoom*x[3*i+1],480-zoom*y[3*i+1],
              zoom*x[3*i+2],480-zoom*y[3*i+2],rougebordure);
        ligne(zoom*x[3*i+2],480-zoom*y[3*i+2],
              zoom*x[3*i+3],480-zoom*y[3*i+3],rougebordure);
        ligne(zoom*x[3*i+3],480-zoom*y[3*i+3],
              zoom*x[3*i+1],480-zoom*y[3*i+1],rougebordure);
       floodfill(zoom*xg,480-zoom*yg,rouge2,rougebordure);
     }
}
```

Graphe de visibilité

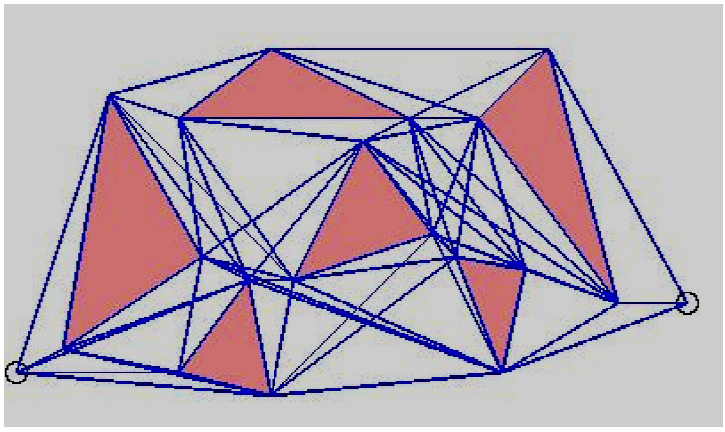
A partir de chacun des points *i* (départ, arrivée, et sommets des triangles), on lance des rayons vers tous les autres points *j*, et l'on garde les lignes de jonction lorsque celles-ci ne traversent aucun côté des triangles. Pour cela on commence par enregistrer les numéros des points extrémités de chaque côté

des triangles, soit *dc*[*k*][*kk*] et *fc*[*k*][*kk*] où *k* est le numéro du triangle concerné, et *kk* un nombre entre 0 et 2, correspondant à chacun des trois côtés. Puis le test d'intersection franche entre la ligne (*i j*) et le côté [*k, kk*] se fait par le biais de la fonction *intersection*() qui réutilise la fonction *gauche*() :

```
int intersection (int i,int j, int k, int kk)
{  if (gauche(i,j,k)*gauche(i,j,kk)<0  && gauche(k,kk,i)*gauche(k,kk,j)<0)   return 1;
   else return 0;
}
int gauche(int i,int j,int k)
{  int v1x,v1y,v2x,v2y,det;
   v1x=x[j]-x[i];v1y=y[j]-y[i];  v2x=x[k]-x[i];v2y=y[k]-y[i];  det=v1x*v2y-v2x*v1y;
   if (det>0) return 1;
   else if (det==0) return 0;
   else return -1;
}
```

A noter que si l'intersection se fait en une extrémité d'un des segments on fait *det* = 0 et *gauche*() ramène 0. En refusant cela comme une intersection franche, le fait que la fonction *intersection*() ramène 0 indique que l'on garde les lignes qui ne traversent pas les côtés, mais aussi celles qui peuvent les longer. Cela s'intègre dans la partie suivante du programme principal :

```
for(i=0;i<nbtriangles;i++)
  {  dc[i][0]=3*i+1;fc[i][0]=3*i+2;  dc[i][1]=3*i+2;fc[i][1]=3*i+3;
     dc[i][2]=3*i+3;fc[i][2]=3*i+1;
  }
for(i=0;i<nbpoints;i++) for(j=0;j<nbpoints;j++)
   {a[i][j]=3000; if(i==j) a[i][j]=0;}
for(i=0;i<nbpoints;i++)  for(j=0;j<nbpoints;j++) if (j!=i)
 { flag=0;
    for(k=0;k<nbtriangles;k++)  for(kk=0;kk<3;kk++)
    if (intersection(i,j,dc[k][kk],fc[k][kk])==1)
          { flag=1;   kk=3;k=nbtriangles; }
    if (flag==0)  {a[i][j]=longueur(i,j);
                    ligne(zoom*x[i],480-zoom*y[i],zoom*x[j],480-zoom*y[j] ,bleu);
                  }
 }
```



Par la même occasion, nous avons construit la matrice d'adjacence *a*[][] pondérée du graphe de visibilité. Les a[*i*][*i*] en diagonale sont mises à 0. Lorsqu'entre *i* et *j* il n'y a pas de jonction, on donne à *a*[*i*][*j*] une valeur élevée (ici 3000), et en cas de jonction, on met dans *a*[*i*][*j*] la longueur du segment de jonction, via la fonction *longueur*() :

```
 int longueur(int a, int b)
{  int vx,vy;
   vx=x[b]-x[a];vy=y[b]-y[a];   return sqrt((float)vx*(float)vx+(float)vy*(float)vy);
}
```

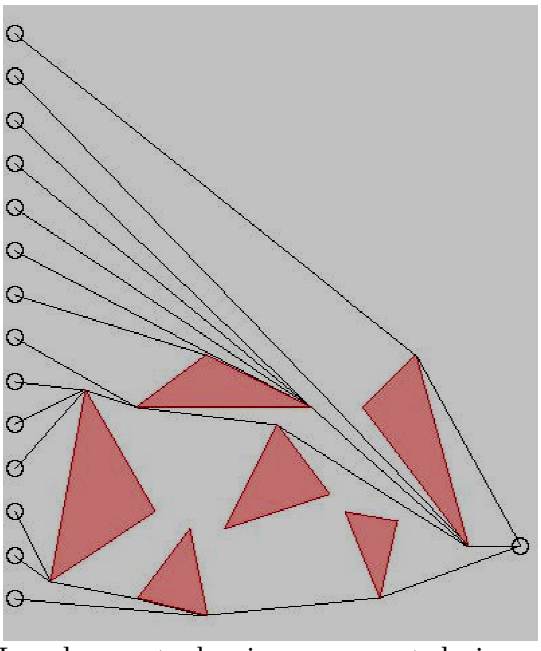
Algorithme du plus court chemin

C'est à partir de la matrice *a*[][] précédente que l'on peut appliquer l'algorithme de Dijkstra pour avoir le plus court chemin. Cet algorithme étant expliqué dans le livre *Combien* ? (*tome 3*) nous nous contentons d'en donner ici le programme adapté au problème présent.

```
longueure=1; e[0]=0;    longueurre=nbpoints-1;
for(i=0;i<nbpoints-1;i++) re[i]=i+1;
for(i=1;i<nbpoints;i++) d[i]=a[0][i];
for(i=1;i<nbpoints;i++) if (d[i]<3000) pred[i]=0;
for(compteur=1;compteur<nbpoints; compteur++)
  {   dmin=3000;
      for(j=0;j<longueurre;j++)   if (d[re[j]]<dmin) {dmin=d[re[j]]; min=re[j];jmin=j;}
      e[longueure++]=min;
      for(j=jmin;j<longueurre;j++) re[j]=re[j+1];
      longueurre--;
      for(i=0;i<longueurre;i++)
        {  u=re[i]; newd=dmin+a[min][u];
           if(newd<d[u]) {d[u]=newd;pred[u]=min;}
        }
  }
dessin();   dessinchemin();
```

Avec la fonction associée :

```
void dessinchemin(void)
{   int i,oldi;
    i=nbpoints-1;
    while (i>0)
      {  oldi=i;i=pred[i];  line(zoom*x[oldi],480-zoom*y[oldi],zoom*x[i],480-zoom*y[i],noir); }
}
```



Les plus courts chemins, en prenant plusieurs points de départ possibles, à gauche

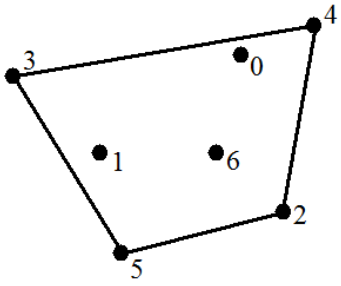
Enveloppe convexe de *N* points dans un plan

Il s'agit d'un polygone convexe (angles saillants) qui a pour sommets certains des *N* points et tel que les *N* points sont tous soit à l'intérieur soit sur la frontière de ce polygone.

Pour le construire, imaginons que les points sont des clous plantés sur une planche. Il suffit de prendre une ficelle, de l'attacher à un point, puis de la serrer autour du paquet des points, comme on fait pour emballer un paquet. Cette construction prouve par la même occasion l'existence de l'enveloppe convexe.[4]

Comment programmer cela ? Commençons par choisir le point le plus bas. A partir de lui traçons un segment vers le point tel que tous les autres points soient à gauche de ce segment, ce qui revient à choisir, parmi les $N - 1$ segments lancés à partir du point bas, celui qui fait l'angle le plus petit avec l'horizontale. On utilise pour cela la fonction *angle*() vue au début de ce chapitre. On trouve ainsi un deuxième point de l'enveloppe. Puis l'on recommence à partir de ce point. A chaque étape, on part du point que l'on vient de trouver et l'on prend parmi les points restants celui qui donne l'angle le plus faible, sous réserve que cet angle soit supérieur à celui trouvé à l'étape précédente. Cette restriction est indispensable, car lorsqu'on arrive vers la fin du tour complet avec un angle qui s'approche des 360°, certains des segments lancés vers les points restants peuvent avoir des angles de quelques dizaines de degrés, alors qu'ils dépassent en fait les 360°, car la fonction *angle*() ramène tous les angles entre 0 et 360°. Il s'agit d'éviter de choisir de tels points qui ne sont pas du tout sur l'enveloppe convexe, d'où la nécessité d'imposer un angle supérieur à l'angle précédent à chaque nouvelle étape, ce qui contraint les angles à rester inférieurs ou égal à 360°. En fait on fait à chaque étape le plus petit virage à gauche possible, et la somme des virages atteint 360° au terme du tour complet.

Algorithme de l'emballage de paquet



Prenons par exemple les 7 points de ce dessin, numérotés de 0 à 6, et faisons marcher l'algorithme. Les numéros de ces points sont placés dans un tableau *a*[] déclaré sur une longueur $N + 1$ et non pas *N*. Au départ ce tableau contient 0 1 2 3 4 5 6 (et n'importe quoi dans la dernière case). Commençons par déterminer le point le plus bas, ici 5. On échange les points 5 et 0 dans le tableau, de façon que 5 soit en position 0, et l'on ajoute 5 en dernière position, ce qui donne le tableau 5 1 2 3 4 0 6 5. On vient d'obtenir le premier morceau de l'enveloppe, souligné ci-dessus, en position $k = 0$, les autres points étant en attente, et notamment le point 5 car il faudra le retrouver à la fin de la boucle. A chaque étape, un point va s'ajouter à l'enveloppe convexe et *k* augmente de 1. Ainsi le deuxième point trouvé est 2, il est placé par échange en position $k = 1$, ce qui donne le tableau 5 2 1 3 4 0 6 5. A chaque étape, le tableau *a*[] contient jusqu'au point en position *k* une partie de l'enveloppe convexe, et au-delà les points en attente. A la fin, quand on retombe sur le point 5, on a toute l'enveloppe convexe, et le reste du tableau *a*[] contient les points qui n'en font pas partie.[5] Cela donne l'évolution suivante :

$k = 0$ 5 1 2 3 4 0 6 5
$k = 1$ 5 2 1 3 4 0 6 5
$k = 2$ 5 2 4 3 1 0 6 5
$k = 3$ 5 2 4 3 1 0 6 5

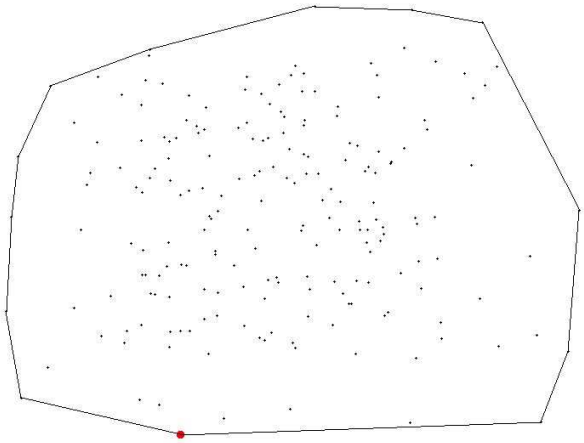
[4] Cet algorithme est attribué à R.A. Jarvis (1973). Il avait auparavant été découvert par Chand et Kapur en 1970.

[5] Cet algorithme peut avoir un problème. En effet, le test d'arrêt se produit lorsque l'on retombe sur le point de départ, à savoir le point le plus bas. Mais il peut arriver qu'il existe plusieurs points les plus bas, auquel cas on risque de retomber sur un autre point que le point de départ après un tour complet, et le programme ne s'arrête pas comme on le désirerait.

$k = 4$ 5 2 4 3 5 0 6 1 (le fait de retrouver 5 est le test d'arrêt).

On en déduit le programme :

```
mise en place des N points (coordonnées en pixels écran) et dessin
ymin=1000;  /* point le plus bas */
for(i=0;i<N;i++)
if (y[i]<ymin) {ymin=y[i]; imin=i;}
tampon=a[0];a[0]=a[imin];a[imin]=tampon;     a[N]=a[0];
oldangle=0.;
for(k=0;k<N;k++) /* tracé de l'enveloppe convexe point par point */
  { minangle=1000.;
    for(j=k+1;j<N+1;j++)
      { alpha=angle(x[a[k]],y[a[k]],x[a[j]],y[a[j]]);
        if (alpha>=oldangle && alpha<minangle)
          { minangle=alpha; jmini=j; }
      }
    tampon=a[k+1];a[k+1]=a[jmini];a[jmini]=tampon;
    oldangle= minangle;
    line(xorig+x[a[k]],yorig-y[a[k]], xorig+x[a[k+1]],yorig-y[a[k+1]],black);
    if (a[k+1]==a[0]) break;
  }
```



Enveloppe convexe pour $N = 200$ points, le point bas initial étant en rouge

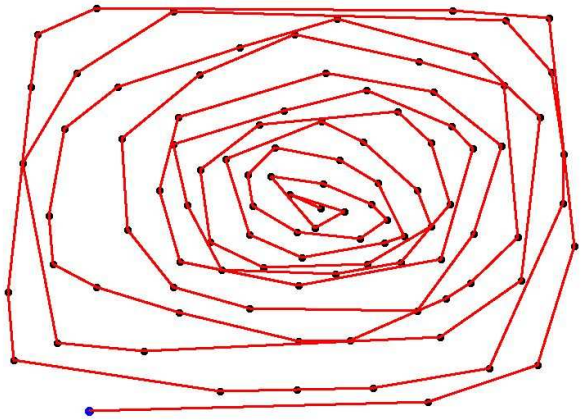
Exercice 3 : Jonction des points en spirale, comme lorsque l'on pèle un fruit

Partir du point le plus bas, et faire en sorte que les N points se succèdent sur une spirale.

Après être parti du point le plus bas, il suffit d'aménager l'algorithme précédent. Ce dernier s'arrêtait après un tour complet. Maintenant on utilise un drapeau *flag*. Tant que celui-ci est à 1, on continue le tour et l'on enregistre les points successifs dans le tableau *spir*[]. Mais chaque fois que l'on a effectué un tour complet, le drapeau est mis à 0, le temps de remettre les pendules à zéro (en faisant *oldangle* = 0), et de trouver le premier point du nouveau tour, le *flag* étant alors remis à 1. On s'arrête lorsque les *N* points sont sur la spirale.

```
points();SDL_Flip(screen);pause();  /* placement des N points et dessin */
    ymin=1000;  /* recherche du point le plus bas, qui sera placé dans spir[0] */
for(i=0;i<N;i++) if (y[i]<ymin) {ymin=y[i]; imin=i;}
filldisc(xorig+x[imin],yorig-y[imin],5,blue);
spir[0]=imin; fait[imin]=1;
fini=0;
```

```
while(fini==0)
  { fini=1;
    oldangle=0.;
    for(k=0;k<N;k++)
      { minangle=1000.;
        flag=0;
        for(j=0;j<N;j++)  if (fait[j]==0)
          { alpha=angle(x[spir[k]],y[spir[k]],x[j],y[j]);
            if (alpha>=oldangle && alpha<minangle)
                { minangle=alpha; jmini=j; flag=1; }
          }
        if (flag==0)
          { oldangle=0.;
            for(j=0;j<N;j++)  if (fait[j]==0)
              { alpha=angle(x[spir[k]],y[spir[k]],x[j],y[j]);
                if (alpha>=oldangle && alpha<minangle)
                    { minangle=alpha; jmini=j; flag=1; }
              }
          }
        if (flag==1)
          { oldangle= minangle; spir[k+1]=jmini;
            linewithwidth(xorig+x[spir[k]],yorig-y[spir[k]], xorig+x[spir[k+1]],yorig-y[spir[k+1]],1,red);
            fait[spir[k+1]]=1;
          }
      }
    for(i=0;i<N;i++) if (fait[i]==0) {fini=0; break;} /* test de fin */
    if (fini==1) break;
  }
```

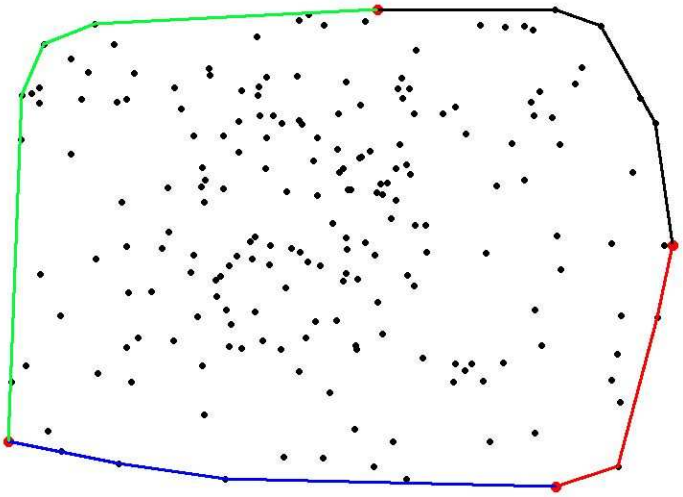


Amélioration de l'algorithme de l'emballage de paquet

Nous allons maintenant tracer l'enveloppe convexe en quatre morceaux, délimités par les points le plus bas, le plus à droite, le plus haut et le plus à gauche. Pour le morceau situé entre le point le plus bas et le point le plus à droite, la pente reste positive, tout en croissant : il suffit de faire les tests sur les segments orientés de pente positive, avec une abscisse et une ordonnée positives. On évite ainsi le calcul de l'*arctangente* utilisé précédemment dans la fonction *angle*(). En fait la fonction *angle*() devient inutile. On fait de même avec les trois autres quarts d'enveloppe, il suffit de changer les points extrêmes et de modifier légèrement les variations *dx* et *dy* associées aux segments, par interversion ou changement de signe. D'où le programme :

```
Se donner les N points numérotés de 0 à N – 1 par leurs coordonnées x[i] et y[i]
ptbas= 0; pthaut=0; ptdroite=0;ptgauche=0;  /* numéros des points extrêmes */
  for(i=1;i<N;i++)
    { if(y[i]<y[ptbas]) ptbas=i;   if(y[i]>y[pthaut]) pthaut=i;
      if(x[i]>x[ptdroite]) ptdroite=i;  if(x[i]<x[ptgauche]) ptgauche=i;
```

```
    }
  filldisc(x[ptbas],y[ptbas],5,red);filldisc(x[ptdroite],y[ptdroite],5,red);
  filldisc(x[pthaut],y[pthaut],5,red);filldisc(x[ptgauche],y[ptgauche],5,red);
/************************* premier quart      *************************/
  envconv[0]=ptbas; d=ptbas; k=1;
  if (ptbas!=ptdroite)
  while (f!=ptdroite)
   { pentemini=10000000000000000000000000000.;
    for(i=0;i<N;i++)
     { dx=x[i]-x[d];dy=y[i]-y[d];
       if (dx>0 && dy>=0)
         { pente=(float)dy/(float)dx;
           if (pente<pentemini) {pentemini=pente;f=i;}
         }
      }
    linewithwidth(x[d],y[d],x[f],y[f],1,black);
    envconv[k++]=f;   d=f;
   }
/************************* deuxième quart      ******************/
  d=ptdroite;
  if (ptdroite!=pthaut)
  while (f!=pthaut)
   { pentemini=10000000000000000000000000000.;
     for(i=0;i<N;i++)
      { dy=-(x[i]-x[d]);dx=y[i]-y[d];
        if (dx>0 && dy>=0)
          { pente=(float)dy/(float)dx;    if (pente<pentemini) {pentemini=pente;f=i;} }
      }
     linewithwidth(x[d],y[d],x[f],y[f],1,red); envconv[k++]=f;   d=f;
   }
/********************* troisième quart      ***********************/
 d=pthaut;
 if(pthaut!=ptgauche)
 while (f!=ptgauche)
  { pentemini=10000000000000000000000000000.;
    for(i=0;i<N;i++)
     { dx=-(x[i]-x[d]);dy=-(y[i]-y[d]);
       if (dx>0 && dy>=0)
         { pente=(float)dy/(float)dx; if (pente<pentemini) {pentemini=pente;f=i;}}
     }
    linewithwidth(x[d],y[d],x[f],y[f],1,blue);  envconv[k++]=f; d=f;
  }
/********************** quatrième quart      ********************/
 d=ptgauche; f=ptgauche;
 if (ptgauche!=ptbas)
 while (f!=ptbas)
  { pentemini=10000000000000000000000000000.;
    for(i=0;i<N;i++)
     { dy=x[i]-x[d];dx=-(y[i]-y[d]);
       if (dx>0 && dy>=0)
         { pente=(float)dy/(float)dx; if (pente<pentemini) {pentemini=pente;f=i;}}
     }
    linewithwidth(x[d],y[d],x[f],y[f],1,green); envconv[k++]=f;  d=f;
  }
```



Robustesse et performance des algorithmes d'enveloppe convexe

Un algorithme est considéré comme robuste s'il réussit à s'adapter à toutes les configurations initiales qu'on lui impose. Les algorithmes géométriques, en particulier ceux portant sur les enveloppes convexes, ont toujours des difficultés à fonctionner dans tous les cas de figure. Ils peuvent marcher à de multiples reprises, et tout à coup se bloquer ou donner un résultat faux dans un cas particulier. Ces cas particuliers sont ici de deux ordres, du moins quand les points sont donnés avec des coordonnées entières. D'une part, dans le nuage des points, plusieurs points peuvent être confondus. Cela suffit à perturber le résultat du programme. Le remède est simple : il convient de faire un prétraitement pour ne conserver qu'un seul représentant de ces points confondus. D'autre part, il arrive que trois points ou plus soient alignés. Ce problème est plus délicat, et mérite d'être analysé en détail. Nous allons prendre deux exemples d'algorithmes relatifs aux enveloppes convexes.

L'algorithme le plus simple

Prenons le moyen le plus naïf pour construire l'enveloppe convexe d'un ensemble de points dans le plan. Il découle directement de la définition. On prend un par un chaque couple de points, ou segment orienté, et l'on regarde si tous les autres points sont situés à sa gauche. Si oui, ce segment orienté appartient à la frontière de l'enveloppe, sinon il n'y appartient pas. On obtient ainsi tous les côtés du polygone de l'enveloppe. Il reste ensuite à les classer dans l'ordre où ils se succèdent sur l'enveloppe lorsqu'on en fait le tour. Cet algorithme n'est pas très performant, puisqu'il est en N^3, N étant le nombre des points. En effet, le temps mis pour prendre tous les couples est en N^2, et comme à chaque fois on fait des tests sur d'autres points, au nombre de $N - 2$ au plus, cela donne une performance en N^3. Le classement final est de son côté opéré en N^2, ou même en $N \log N$, comme les tris habituels. Il reste finalement un comportement en N^3. Par rapport aux algorithmes en N log N, comme celui que l'on va voir ci-dessous, cela semble peu performant, encore que dans la réalité, avec un nombre N allant jusqu'à quelques centaines, le temps d'attente des résultats est quasiment le même pour tous les types d'algorithmes. Cet algorithme a du moins l'avantage d'être facile à trouver et à implanter, et le programme est le suivant:

```
/* On se donne un repère d'origine (xorig, yorig) avec l'axe des y vers le haut. Les coordonnées des points
sont en pixels écran. Les N points sont mis en place au hasard sur l'écran. En même temps, on détermine leur
centre de gravité (xg,yg), ce point étant toujours à l'intérieur de l'enveloppe convexe */
cumulx=0;cumuly=0;
for(i=0;i<N;i++)
    { x[i]=random(550); y[i]=random(450);  cumulx+=(float)x[i];cumuly+=(float)y[i];   }
xg=cumulx/(float)N;yg=cumuly/(float)N;
dessiner les points
```

```
/* Au lieu de prendre chaque couple de points, on prend chaque paire, puis parmi ses deux orientations
possibles, on choisit la seule qui puisse prétendre être valable, celle qui met le centre de gravité à gauche du
segment orienté */
  n=0;      /* n sera le nombre de côtés ou de points de l'enveloppe convexe */
  for(i=0;i<N-1;i++)     for(j=i+1;j<N;j++)
    {  debut=i;fin=j;     /* on va éventuellement intervertir le début et l'extémité du segment */
       if (gauche(xg,yg,x[debut],y[debut],x[fin],y[fin])==NON)     { debut = j;  fin= i; }
       flag=OUI;
       for(k=0;k<N;k++)  if (k!=debut && k!=fin)    /* on prend tous les autres points */
       if (gauche(x[k],y[k],x[debut],y[debut],x[fin],y[fin])==NON)    flag=NON;
       if (flag==OUI)        /* on va tracer le segment qui est sur l'enveloppe */
         { envd[n]=debut;envf[n++]=fin;
           line(xorig+x[debut],yorig-y[debut],xorig+x[fin],yorig-y[fin]);
         }
    }
  /* Il reste à classer tous les côtés de l'enveloppe dans l'ordre où ils se succèdent */
  for(i=1; i<n; i++)    for(j=i; j<n; j++)   if (envd[j] == envf[i-1])
    {  tampon=envd[i];envd[i]=envd[j];envd[j]=tampon;
       tampon=envf[i];envf[i]=envf[j];envf[j]=tampon;
    }
  envd[n]=envd[0];envf[n]=envf[0];
  for(i=0;i<n;i++)   /* on redessine le polygone dans le bon ordre */
  line(xorig+x[envd[i]],yorig-y[envd[i]],xorig+x[envf[i]],yorig-y[envf[i]]);

  /* fonction gauche()  */
  int gauche(int x2,int y2, int x0,int y0, int x1, int y1)
  { int v1x,v1y,v2x,v2y,det;
    v1x=x1-x0;v1y=y1-y0;  v2x=x2-x0;v2y=y2-y0;
    det=v1x*v2y-v2x*v1y;
    if (det>=0) return OUI;
    return NON;
  }
```

Mais ce programme a des problèmes en cas d'alignement de points. Dans la fonction *gauche*(), l'alignement est assimilé au fait d'être à gauche. Cela a plusieurs répercussions. Si trois points de la frontière sont alignés, par exemple *A, B, C* dans cet ordre, il y a trois segments orientés **AB**, **AC**, **BC** qui seront considérés comme valables, alors que le seul côté du polygone frontière que l'on désire conserver est *AC*. L'autre écueil est dans le classement des segments successifs du polygone. La partie correspondante du programme précédent ne fonctionne plus, car elle suppose que toute extrémité d'un segment est le début du suivant, et cela de façon unique. Il convient donc de rectifier la fonction *gauche*(). Un point sera encore considéré comme à gauche d'un segment orienté **PQ** s'il est dans l'alignement mais à condition qu'il soit situé entre *P* et *Q*. D'où la nouvelle fonction:

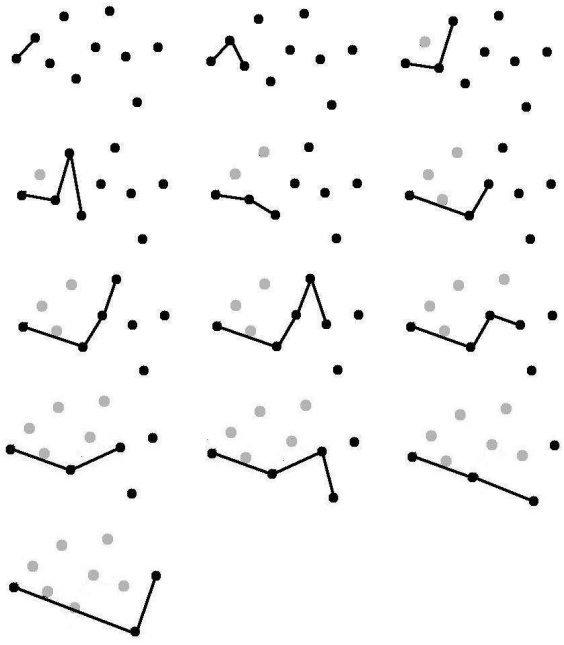
```
int gauche(int x2,int y2, int x0,int y0, int x1, int y1)
{
v1x=x1-x0;v1y=y1-y0;   v2x=x2-x0;v2y=y2-y0;
det=v1x*v2y-v2x*v1y;
if (det>0) return OUI;
if (det==0)
  {   if (v1x!=0) kk=(float)v2x/(float)v1x; else kk=(float)v2y/(float)v1y;
      if (kk>0. && kk<1.) return OUI;
      else return NON;
  }
return NON;
}
```

L'algorithme initial était simple, mais il a fallu un certain raffinement pour qu'il puisse fonctionner dans tous les cas. La robustesse de l'algorithme est à ce prix.

L'algorithme incrémental (*monotone chain algorithm*)

Prenons maintenant un autre algorithme, considéré comme bien plus performant, l'algorithme dit incrémental. Commet tracer la moitié basse de l'enveloppe, entre le point le plus à gauche et celui le plus à droite, étant entendu qu'il en sera de même pour la partie haute? On commence par classer les points selon leurs abscisses croissantes, et en cas d'égalité, selon leurs ordonnées croissantes. Il s'agit là d'un classement de type lexicographique (alphabétique). Si le nombre N de points est inférieur à 1000, il suffit de fabriquer pour chaque point i le nombre $z[i]=1000*x[i] + y[i]$ et de classer les points suivant les valeurs croissantes de $z[]$.

Puis on prend ces points ainsi classés l'un après l'autre. Les deux premiers sont mis d'office dans l'enveloppe. Puis on prend le point suivant, que l'on met aussi d'office dans l'enveloppe, mais l'on regarde si les trois points ainsi obtenus font un virage à gauche ou à droite. S'ils font un virage à droite, on élimine le point intermédiaire de l'enveloppe, qui ne conserve que deux points pour le moment. Et l'on continue de même. Supposons que l'on en soit arrivé au $k^{ème}$ point de l'ensemble trié des points, et que l'on ait construit la partie de l'enveloppe basse correspondant à ces points. On prend alors le $k+1^{ème}$ point de l'ensemble, et on l'ajoute dans l'enveloppe. S'il est à gauche du dernier segment orienté de l'enveloppe, on le garde ainsi que ceux qui le précèdent. Par contre, si les trois derniers points de l'enveloppe (le dernier des trois étant le $k+1^{ème}$ point) forment un virage à droite, on élimine le point intermédiaire que l'on remplace par ce $k+1^{ème}$ point. Puis l'on recommence avec les trois derniers points de l'enveloppe, et l'on élimine le point intermédiaire s'il y a un virage à droite. On continue ainsi en marche arrière jusqu'à ce qu'il n'y ait plus de virage à droite, et en s'arrêtant aussi s'il ne reste plus que deux points dans l'enveloppe. On fait de même jusqu'au dernier point trié de l'ensemble des points. Voici un exemple de construction :



Pour résumer, on prend chaque point un par un dans l'ordre croissant des abscisses. On l'ajoute d'office comme nouveau point appartenant à l'enveloppe. Mais on élimine les points précédents placés dans l'enveloppe et qui provoquent des virages à droite. Ainsi à chaque étape, à défaut d'avoir une partie de l'enveloppe finale, on a une succession convexe de points. Quand on arrive au dernier point, et que l'on élimine éventuellement certains points précédents, il reste l'enveloppe convexe basse.

Dans le cas présent, qu'en est-il des problèmes d'alignement de points ? D'abord, lors du tri des points selon les abscisses, seul l'alignement vertical nous concerne, et c'est la raison pour laquelle on fait un classement selon les ordonnées croissantes quand les abscisses sont les mêmes. Ensuite, quand on supprime les virages à droite sur trois points pour obtenir la bonne enveloppe convexe, on s'arrange

aussi pour supprimer les alignements de trois points, en éliminant le point intermédiaire. On fait comme s'il s'agissait d'un virage à droite, et la fonction *droite*() est facile à implanter, plus que dans l'algorithme précédent avec sa fonction *gauche*(). On en déduit le programme de l'enveloppe basse (on fait quasiment de même pour l'enveloppe haute):

```
points();  /* construction et dessin des points (x[i], y[i]) ici avec Oy dirigé vers le bas */
for(i=0;i<N;i++)   /* points mis dans l'ordre alphabétique */
     { a[i]=i;   z[i]=1000*x[i]+y[i]; }
for(i=0;i<N-1;i++)  for(j=i+1;j<N;j++) if (z[i]>z[j])
     { ltampon=z[i];z[i]=z[j],z[j]=ltampon;
       tampon=a[i];a[i]=a[j];a[j]=tampon;  /* a[] contient les numéros des points dans l'ordre alphabétique */
     }
/*********    enveloppe basse     *********/
env[0]=a[0];env[1]=a[1]; k=1;filldisc(x[env[0]],y[env[0]],5,red);
for(i=2;i<N;i++)
  { k++;
    env[k]=a[i];
    while(k>1 && gauche(env[k],env[k-2],env[k-1])==OUI  )
       { env[k-1]=env[k];
         k--;
       }
   }
for(i=0;i<k;i++)
{
linewithwidth(x[env[i]],y[env[i]],x[env[i+1]],y[env[i+1]],1,red);
filldisc(x[env[i]],y[env[i]],5,black);
}
filldisc(x[env[k]],y[env[k]],5,black);
int K;       /*****************   enveloppe haute   *********************************/
env[k]=a[N-1];env[k+1]=a[N-2]; K=k;k=k+1;
for(i=N-3;i>=0;i--)
  { k++;
    env[k]=a[i];
    while( k>K+1 && gauche(env[k],env[k-2],env[k-1])==OUI )
        { env[k-1]=env[k];  k--; }
       }
for(i=K;i<k;i++)
  { linewithwidth(x[env[i]],y[env[i]],x[env[i+1]],y[env[i+1]],1,blue);
    filldisc(x[env[i]],y[env[i]],5,black);
  }
filldisc(x[env[k]],y[env[k]],5,black);
for(i=0;i<k;i++) filldisc(x[env[i]],y[env[i]],5,red);
```

Avec ces fonctions :

```
/**********     construction et dessin des points     **********/
void points(void)
{ int i,j,R;  /* on fait en sorte que les points soient séparés d'une distance au moins égale à R */
  for(i=0;i<N;i++)      /* axe des y dirigé vers le bas */
   {do  { x[i]=20+rand()%760;y[i]=20+rand()%560;
          if (x[i]>400 && x[i]<500 && y[i]>300  && y[i]<400) R=5;
          else if (x[i]>300 && x[i]<600 && y[i]>300  && y[i]<400) R=10;
          else R=30;
        }
   while (getpixel(x[i],y[i]) == red);
   filldisc(x[i],y[i],R,red);
   }
 for(i=0;i<800;i++) for(j=0;j<600;j++)  /* on supprime les disques rouges */
 if (getpixel(i,j)==red) putpixel(i,j,white);
}
```

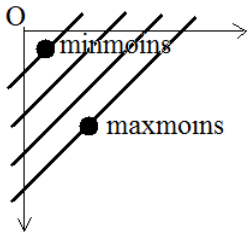
```
/*********     fonctions  droite et gauche     **********/
int droite(int i2,int i0,int i1)
{int v1x,v1y,v2x,v2y,x2,y2,x1,y1,x0,y0; long int det;
x2=x[i2];y2=y[i2];  x1=x[i1];y1=y[i1];  x0=x[i0];y0=y[i0];
v1x=x1-x0;v1y=y1-y0;  v2x=x2-x0;v2y=y2-y0;
det=v1x*v2y-v2x*v1y;
if (det<=0) return OUI;
return NON;
}
int gauche(int i2,int i0,int i1)
{int v1x,v1y,v2x,v2y,x2,y2,x1,y1,x0,y0; long int det;
x2=x[i2];y2=y[i2]; x1=x[i1];y1=y[i1]; x0=x[i0];y0=y[i0];
v1x=x1-x0;v1y=y1-y0;  v2x=x2-x0;v2y=y2-y0;
det=v1x*v2y-v2x*v1y;
if (det>=0) return OUI;
return NON;
}
```

En termes de performance, le tri préliminaire peut être réalisé en *N log N*. Puis l'on effectue un parcours de l'ensemble des points, de l'ordre de *N*, mais à chaque fois il peut y avoir plusieurs sous-étapes de suppression de points (dans la boucle *while*). Cette partie du programme reste de l'ordre de *N* sauf dans le pire des cas où devrait supprimer à chaque fois la plupart des points déjà trouvés. Finalement, on obtient une performance en *N log N*, nettement meilleure que celle de l'algorithme précédent.

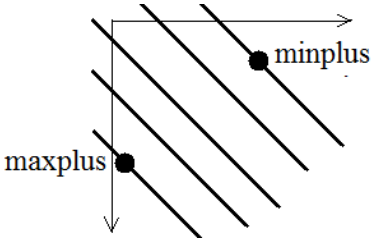
Mais tout cela doit être relativisé. Parmi les *N* points, il est possible d'en éliminer préalablement un grand nombre, dont on est sûr qu'ils ne sont pas sur l'enveloppe convexe. Ce procédé d'élagage permet de réduire fortement le nombre *N* des points dont on cherche l'enveloppe. Avec ce nombre de points réduit de 75% voire plus, il peut n'en rester que quelques dizaines ou quelques centaines, et dans ces conditions tous les algorithmes se valent, même ceux considérés théoriquement comme plus lents. Mais quels points supprimer ?

Elagage dans l'ensemble des points

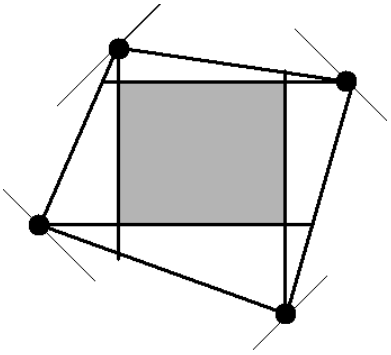
Nous allons nous inspirer de la méthode proposée par A. Marietta en 2012 (cf. *the fastest convex hull* ...)



Prenons les droites d'équation $y = x + c$ dans le repère de l'écran, ou encore $y - x = c$. Plus la constante c est grande, plus la droite est éloignée de l'origine *O* du repère. Parmi les *N* points déterminons le point *i* tel que la différence $y[i] - x[i]$ soit la plus petite possible, ce qui donne le point numéro *iminmoins*. Prenons ensuite celui pour lequel la différence est la plus grande possible, soit le point *imaxmoins*.



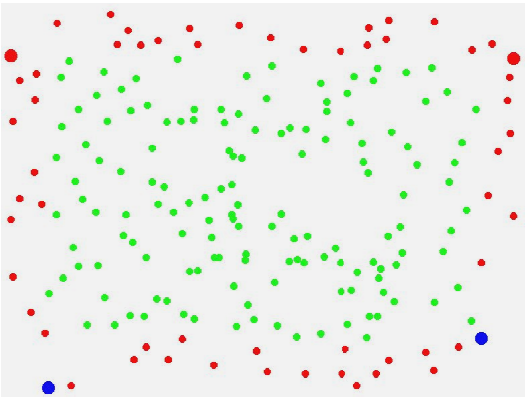
Recommençons avec les droites d'équation $y = - x + c$, ou bien $y + x = c$. On obtient comme précédemment deux points de l'ensemble des *N* points, celui dont la somme $y[i] + x[i]$ est la plus petite possible, soit le point *iminplus*, et celui dont la somme est la plus grande possible, soit le point numéro *imaxplus*.



Les quatre points ainsi obtenus forment un quadrilatère. Prenons maintenant le carré à côtés horizontaux et verticaux délimité par les abscisses et les ordonnées minimales et maximales des quatre points, soit *xmin* et *xmax*, ainsi que *ymin* et *ymax*. Cela donne le carré en gris sur le dessin. On peut affirmer que les points qui sont à l'intérieur de ce carré ne sont pas sur l'enveloppe convexe. Ceux-ci peuvent être supprimés. On en déduit le programme qui permet d'éliminer les points superflus, avec la fonction suivante.

```
void elagage(void)
{ int i,ii,jj,xmin,xmax,ymin,ymax;
   int minmoins,minplus,maxmoins,maxplus,flag,diff,sum,iminmoins,iminplus,imaxmoins,imaxplus;
  minmoins=10000;maxmoins=-10000; minplus=10000; maxplus=-10000;
  for(i=0;i<N;i++)
    {   filldisc(x[i],y[i],5,red);
        flag=0;
        diff=y[i]-x[i]; sum=x[i]+y[i];
        if (diff<minmoins) {minmoins=diff; iminmoins=i;}
        if (diff>maxmoins) {maxmoins=diff;imaxmoins=i;}
        if (sum<minplus) {minplus=sum; iminplus=i; }
        if (sum>maxplus) {maxplus=sum; imaxplus=i; }
    }
  filldisc(x[iminmoins],y[iminmoins],9,red);   filldisc(x[iminplus],y[iminplus],9,red);
  filldisc(x[imaxmoins],y[imaxmoins],9,blue);   filldisc(x[imaxplus],y[imaxplus],9,blue);

   if (x[imaxmoins]<x[iminplus])   xmin=x[iminplus]; else xmin=x[imaxmoins];
   if (x[imaxplus]<x[iminmoins]) xmax= x[imaxplus] ;else xmax=x[iminmoins];
   if (y[imaxplus]<y[imaxmoins]) ymax=y[imaxplus]; else ymax=y[imaxmoins];
    if (y[iminplus]<y[iminmoins]) ymin= y[iminmoins]; else ymin=y[iminplus];
    for(i=0;i<N;i++)
    if (x[i]>xmin && x[i]<xmax && y[i]>ymin && y[i]<ymax)
    filldisc(x[i],y[i],5,green);
}
```



Les points dessinés en vert peuvent être supprimés, seuls restent les points rouges pour déterminer l'enveloppe convexe. Dans cet exemple, sur les N = 200 points, environ 75% des points sont supprimés.

Exercice 4 : Pelures d'oignons

Prendre l'enveloppe convexe de N points, puis l'enveloppe convexe des points restants -ceux qui ne sont pas sur l'enveloppe convexe précédente, et continuer ainsi jusqu'à épuisement.

Les enveloppes successives, numérotées à partir de 0, sont enregistrées dans le tableau *sousenv*[*numero*][], chacune avec sa couleur indexée par son numéro.

```
Placer les N points x[], y[]
for(i=0;i<N;i++) {fait[i]=0;a[i]=i;  }
nombre=N; boucle=0; fini=NON;
while(fini==NON)
  {  boucle++;
     for(i=0;i<nombre;i++)   {  z[i]=1000*x[a[i]]+y[a[i]]; }
     for(i=0;i<nombre-1;i++)  for(j=i+1;j<nombre;j++) if (z[i]>z[j])
       { ltampon=z[i];z[i]=z[j],z[j]=ltampon;
         tampon=a[i];a[i]=a[j];a[j]=tampon;
       }
     if (nombre==1) { fait[a[0]] = boucle; sousenv[boucle][0]=a[0];nb[boucle]=2;
                      boucle++;      break;}
     env[0]=a[0];env[1]=a[1]; k=1; /* enveloppe basse */
     for(i=2;i<nombre;i++)
       { k++;   env[k]=a[i];
         while(k>1 && gauche(env[k],env[k-2],env[k-1])==OUI  )    { env[k-1]=env[k];   k--; }
        }
     for(i=0;i<k;i++)
       { linewithwidth(x[env[i]],y[env[i]],x[env[i+1]],y[env[i+1]],1,red[boucle]);
         filldisc(x[env[i]],y[env[i]],5,black) ; fait[env[i]]=boucle;
         sousenv[boucle][nb[boucle]++]=env[i];
       }
     filldisc(x[env[k]],y[env[k]],5,black); fait[env[k]]=boucle;
     sousenv[boucle][nb[boucle]++]=env[k];
     env[k]=a[nombre-1];env[k+1]=a[nombre-2]; K=k; k=k+1; /* partie haute de l'enveloppe */
     for(i=nombre-3;i>=0;i--)
       { k++;
         env[k]=a[i];
        while( k>K+1 && gauche(env[k],env[k-2],env[k-1])==OUI )
           { env[k-1]=env[k];  k--; }
       }
     for(i=K;i<k;i++)
       { linewithwidth(x[env[i]],y[env[i]],x[env[i+1]],y[env[i+1]],1,red[boucle]);
         filldisc(x[env[i]],y[env[i]],5,black);
         fait[env[i]]=boucle; if (i>K) sousenv[boucle][nb[boucle]++]=env[i];
       }
     filldisc(x[env[k]],y[env[k]],5,black);fait[env[k]]=boucle;
     sousenv[boucle][nb[boucle]++]=env[k];
     n=0;
     for(i=0;i<N;i++) if (fait[i]!=0) filldisc(x[i],y[i],5,rouge);
     else aa[n++]=i;
     nombre=n;    for(i=0;i<nombre;i++) a[i]=aa[i];
     for(i=0;i<N;i++) if (fait[i]==0) { fini=NON; break;} /* test d'arrêt de la boucle while */
     if (i==N) fini=OUI;
     boucle++;
  }
```