

L2 Info - 2022-2023

Programmation C

Projet 2^o partie

Enveloppe Convexe dans le plan (suite)

Cette seconde partie est un prolongement direct de la partie 1.

Elle propose deux options (au choix)

- ① si vous préférez la programmation : Enveloppes Convexes Emboîtées
- ② si vous préférez les mathématiques : Evaluation de Complexité

N'attaquez pas cette seconde partie tant que la première n'est pas parfaitement opérationnelle !

Option 1 - Enveloppes Convexes Emboîtées

L'idée ici est de construire, à la volée, une structure géométrique de polygones (enveloppes) emboîtés de sorte que **chaque point du nuage soit un sommet d'une et une seule enveloppe**.

On va donc être amené à gérer une **liste chaînée** d'Enveloppes Convexes.

Concrètement cela se déroule comme suit pour le traitement séquentiel des points de \mathcal{E}_n :

- ① les 3 premiers points traités initialisent la 1^o enveloppe de la liste.
- ② pour un nouveau point,
 - soit il est à l'**extérieur** de cette enveloppe et il vient s'intégrer dans son polygone,
 - soit il est à l'**intérieur** et il devient le 1^o sommet d'une nouvelle enveloppe (à un seul sommet) ajoutée à la liste, et qui va se développer à l'*intérieur* de la première. Bien sûr cette nouvelle enveloppe ne prendra réellement corps que lorsqu'elle sera constituée d'au moins 3 sommets
- ③ le point suivant pourra alors être
 - Ⓐ à l'**extérieur** de la 1^o enveloppe ➡ il s'intègre à celle-ci
 - Ⓑ à l'**intérieur** de la 1^o mais à l'**extérieur** de la seconde ➡ il s'intègre à la seconde
 - Ⓒ à l'**intérieur** de la 2^o ➡ il initialise une 3^o enveloppe, emboîtée dans la seconde
- ④ et ainsi de suite pour chaque point de l'ensemble :
 - soit il devient un sommet d'une enveloppe existante,
 - soit il déclenche la création, en fin de liste, d'une nouvelle enveloppe (dont il est l'unique sommet), emboîtée dans la plus petite.

⚠ Bien sûr, comme dans la première partie du projet, l'insertion d'un point dans une enveloppe EC_k peut conduire à la disparition d'un (ou plusieurs) sommet(s) (**vertex**).

Le point correspondant à un **vertex** supprimé passe alors à l'intérieur de EC_k et doit alors être re-traité pour être intégré à l'enveloppe suivante EC_{k+1} .

On voit que cela déclenche potentiellement une *réaction en chaîne* qui peut modifier **toutes** les enveloppes et conduire à la création d'une nouvelle enveloppe, au centre du nuage, qui vient s'insérer en fin de liste.

Il devient alors judicieux de mettre en oeuvre une version **récursive** de la fonction de traitement d'un point

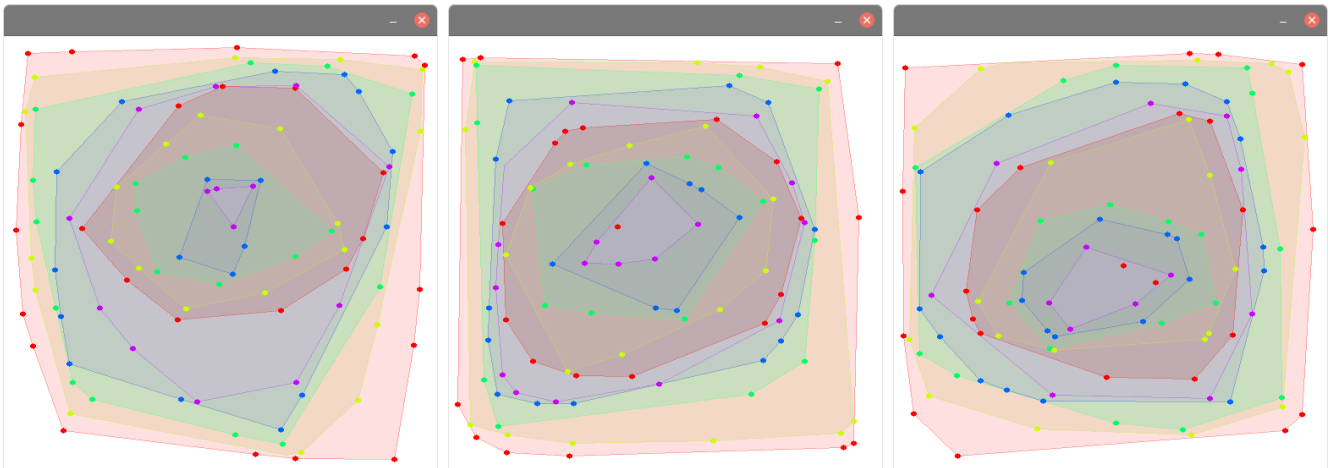
- pour un point situé à l'intérieur d'une enveloppe EC_k , on relance le traitement de ce même point avec l'enveloppe suivante EC_{k+1}
- de même pour un **vertex** supprimé de EC_k lors de l'insertion du point courant : le point correspondant à ce vertex est re-traité avec EC_{k+1} .

La récursivité se termine naturellement par l'insertion du point courant dans une des enveloppes existantes ou par la création d'une nouvelle enveloppe en fin de liste.

En fin de traitement, la dernière liste, au centre, peut être réduite à 1 ou 2 sommets comme dans certains des exemples ci-dessous, pour des distributions rectangulaires à 100 points.

La gestion des couleurs et de tous les aspects graphiques et "esthétiques" est laissé à votre appréciation. Vos choix doivent néanmoins permettre une visualisation claire, pertinente et robuste (i.e. cohérente, en toute situation). L'utilisation du **canal alpha** des couleurs RGBA (transparence) peut s'avérer utile.

De même, et comme pour la première partie du projet, un menu graphique ou des options en ligne de commande seront fort appréciées : nombre de points, forme du nuage, point par point au clic souris, animation.... (bref, tout ce qui peut mettre en valeur votre travail).



Pour cette seconde partie vous allez être amenés à modifier (légèrement) les structures de données et les fonctions utilisées pour la 1^o partie. Vous devrez donc créer une **version.2** de votre code (copie de la **version.1**, à conserver, bien sûr !) et rendre les 2 versions en même temps. Chaque version devra être compilable et exécutable indépendamment de l'autre.

L'ensemble des consignes présentées en fin de première partie (conditions de rendu) sont bien sûr valables également pour cette seconde partie.

Option 2 - Evaluation de la complexité moyenne

Il s'agit ici d'étudier la complexité et les performances de l'algorithme dans différentes configurations du nuage de points (cas les plus favorables, défavorables).

Pour cette étude il n'y a plus besoin de visualisation de l'enveloppe, donc plus d'utilisation de la `libMLV`. En revanche il faudra pouvoir tester la méthode dans de nombreuses configurations et avec un très grand nombre de points (plusieurs milliers, voire millions) et en extraire des informations statistiques.

☞ Quelques légères adaptations du programme de la partie 1 seront nécessaires, mais c'est essentiellement sur la base d'un **rapport** que se fera l'évaluation.

Complexité théorique

Pour évaluer la complexité⁽¹⁾ il faut déterminer combien d'opérations sont nécessaires, en moyenne, pour traiter un point de l'ensemble \mathcal{E}_n .

Considérons une situation où on a déjà construit une enveloppe convexe à p sommets et étudions le coût de traitement du point courant P_k ($0 < p < k < n$).

- S'intègre-t-il dans le polygône ou est-il à l'intérieur ?
 - quelle est la situation la plus favorable ?
 - quelle est la situation la plus **défavorable** ?
- Si il est à l'intérieur, il n'y aura plus rien à faire. Mais si il s'intègre au polygône, peut-on évaluer le coût de cette insertion (au moins le borner) ?

A partir de ces différentes situation, essayez de donner une évaluation théorique de la complexité de cet algorithme⁽²⁾.

Influence de la forme de la distribution de points

On considère ici que les n points de l'ensemble \mathcal{E}_n sont stockés dans un tableau et traités dans l'ordre, du premier au dernier.

Il vous a été demandé, dans la première partie, de mettre en oeuvre au moins 2 façons de distribuer ces points dans la fenêtre : dans un disque et dans un carré de centre C et de "rayon" (ou demi-côté) r .

Et pour chacune de ces formes, une version où le "rayon" r évolue de manière croissante : les premiers points sont concentrés autour du centre et les suivants s'en éloignent petit à petit.

☞ Expliquez pourquoi et comment ces différentes configurations impactent le déroulement de l'algorithme.

Vérification expérimentale

Pour cette partie, il va falloir mesurer et enregistrer certaines informations caractéristiques de l'algorithme telles que la longueur (nombre de sommets) de l'enveloppe et le nombre d'opérations "élémentaires" à différentes étapes du calcul, puis afficher les données collectées.

Quelles informations enregistrer ?

D'après la structure `ConvexHull` proposée en première partie, cet objet dispose d'un champs permettant d'enregistrer, à chaque étape (i.e. pour chaque point), la longueur courante de l'enveloppe ainsi que sa longueur maximale. Cette donnée est donc déjà accessible.

Pour mesurer la complexité nous nous contenterons de comptabiliser le nombre *opérations élémentaires* réalisées, sans trop chercher à détailler.

⁽¹⁾cf. https://fr.wikipedia.org/wiki/Complexité_en_moyenne_des_algorithmes

⁽²⁾cf. https://fr.wikipedia.org/wiki/Calcul_de_l'enveloppe_convexe

Nous distinguerons ici trois opérations élémentaires, chacune s'effectuant à *coût constant* :


- ① calcul d'orientation d'un triplet de point (le point courant et deux points de l'enveloppe) :
- ② ajout d'un sommet dans l'enveloppe
- ③ suppression d'un sommet de l'enveloppe.

Vous devrez donc adapter votre programme pour compter, à chaque étape, ce nombre d'opérations. Chacune compte pour 1 et on utilisera un seul compteur.

L'objectif est évidemment d'étudier comment évolue ce nombre d'opérations en fonction du nombre de points de l'ensemble, de la forme de la distribution et de le comparer à l'évolution de la longueur de l'enveloppe.

Comment les enregistrer ?

Pour enregistrer ces informations (longueurs d'enveloppes / nombres d'opérations), nous passerons par un fichier texte, rempli à la volée, constitué de 3 colonnes : le nombre de points traités, la longueur courante de l'enveloppe, le nombre d'opérations "consommées" pour chaque point.

Ce fichier texte ressemblera donc à quelque chose comme ça  pour une distribution en "rectangle croissant" à 20 points.

Sur un tel exemple, rien de bien significatif n'émerge de ces données. C'est normal.

Pour voir apparaître des informations pertinentes il faudra **beaucoup** plus de points et des données *moyennisées*.

	PNT	LEN	OPE			
	0	1	1		10	5 10
	1	2	1		11	6 4
	2	3	2		12	6 6
	3	4	6		13	5 9
	4	5	7		14	6 8
	5	6	8		15	7 8
	6	7	9		16	7 7
	7	7	7		17	7 7
	8	7	11		18	6 8
	9	6	8		19	6 6

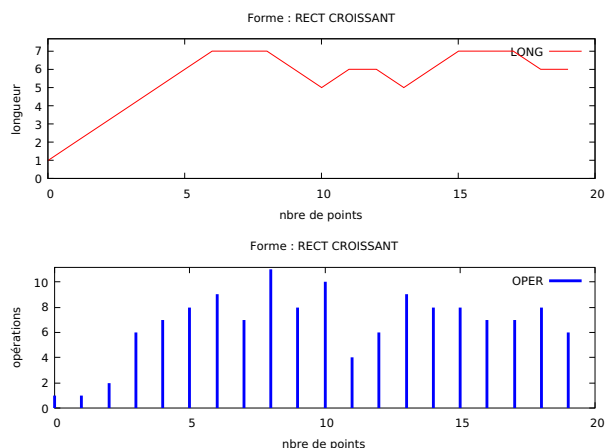
Comment les visualiser ?

Pour visualiser ces données vous pourrez utiliser l'application **gnuplot**⁽³⁾ disponible sur les machines de l'université et très facile à installer sous **linux**, **windows** et **OSX**. C'est une librairie graphique de *visualisation scientifique*, permettant de tracer des courbes, des surfaces... Son utilisation est très simple pour ce qui vous est demandé ici.

Le plus simple est de créer, à partir de votre programme, un fichier script (appelons-le **script.gp**) correctement formaté et de lancer son exécution, directement depuis votre programme, grâce à la commande `system("gnuplot script.gp")`⁽⁴⁾.


Sur l'exemple précédent, les données (enregistrées dans un fichier texte **stats.data**) donneraient le graphique ci-contre, avec le script (auto-généré) suivant :

```
# script.gp
set title 'Forme : RECT CROISSANT'
data = 'stats.data'
set xrange [0:20]
set yrange [0:7]
set multiplot layout 2,1 scale 1.,1.
set ylabel 'longueur'
plot data using 'PNT': 'LEN' with lines lc rgb 'red'
set ylabel 'opérations'
set yrange [0:11]
plot file using 'PNT': 'OPE' with impulses lc rgb 'blue'
pause mouse 'Cliquer pour quitter'
```



Un exemple de programme réalisant quelque chose de semblable (création d'un fichier de données, écriture d'un script formaté et exécution de **gnuplot** sur ce script) vous est fourni sur la plateforme **eLearning**. C'est un exemple très primaire et sans grand intérêt. Vous pouvez certainement faire beaucoup mieux....

⁽³⁾ cf. <http://www.gnuplot.info/> + un petit tour sur **google**...

⁽⁴⁾  `man 3 system`

Des données pertinentes

Vos graphiques devront mettre en valeur d'une part l'influence de la *forme* de la distribution et d'autre part la corrélation existant entre les longueurs d'enveloppes et les quantités de calculs nécessaires (la complexité...)

Pour obtenir des données pertinentes il faudra bien sûr travailler sur des distributions de points beaucoup plus volumineuses⁽⁵⁾ et extraire des statistiques moyennes. Il pourra alors être judicieux de travailler avec des réels.

On pourra par exemple déterminer des longueurs et nombres d'opérations moyens sur des paquets de points plus ou moins gros : pour 10 000 points, on extrait une statistique moyenne tous les 50 points, pour obtenir des graphiques à 200 échantillons.

On pourra également tracer, sur un même graphique, l'évolution moyenne de ces données pour les différentes formes de distributions. Ces distributions étant aléatoires (simplement contraintes par une *forme globale*), plusieurs "tirages" moyennés seront nécessaires pour mettre en valeur les éléments pertinents. Par exemple, lancer 20 simulations de distributions rectangulaires à 10 000 points, traités par paquets de 100.

Dans cette option, il n'y a donc que très peu d'aspects "programmation" mais un travail d'analyse du problème et de visualisation scientifique pour *faire parler* l'algorithme. Le rapport sera donc tout aussi important (si ce n'est plus) que le code.

L'ensemble des consignes présentées en fin de première partie (conditions de rendu) sont bien sûr valables également pour cette seconde partie.

⁽⁵⁾il peut se passer des choses intéressantes au delà de 100 000 points...