Université Jean Monnet

Faculté des Sciences et Techniques

Projet de Programmation fonctionnelle OCaml

Licence 1 MISPIC

2018

Encadrants

Émilie Samuel - emilie.samuel@univ-st-etienne.fr
Baptiste Jeudy - baptiste.jeudy@univ-st-etienne.fr

Département d'Informatique

Déroulement du projet

L'objet du projet est multiple :
— en premier lieu, il s'agit de programmer et comparer des fonctions de tris de listes ;
— en second lieu, il s'agit de réaliser un programme permettant de calculer et de visualiser des enveloppes convexes de points du plan.

Binôme

Vous devez faire ce projet en binôme. Un binôme est composé de deux personnes. Une personne seule n'est donc pas un binôme. Trois personnes ne forment pas un binôme non plus. Toute dérogation à cette règle nécessitera l'accord (exceptionnel) de votre chargé de TP. Plus clairement, si vous ne la respectez pas, vous serez pénalisés lors de l'évaluation du projet. D'autre part, il est impératif que chaque membre du binôme travaille, dans la mesure où l'examen final (écrit) comportera des questions portant sur ce projet. De plus, il n'est pas automatique que deux étudiants du même binôme aient la même note.

Les binômes devront être constitués d'élèves du même groupe de TD. La constitution de chaque binôme (noms des étudiants) devra être communiquée au chargé de TP correspondant.

Planning

Ce projet est relativement long à réaliser, et vous devez l'avoir mené à ben d'ici la fin du semestre. Il est donc impératif que vous travailliez dès le début, *en dehors* des heures de TP (voire pendant si vous avez terminé les TP), dans la salle en libre-accès du CIS (ou chez vous). Nombre d'étudiants n'ont pas suivi cette consigne les années précédentes ; ils ont fini par passer leurs jours et leurs nuits à écrire du OCaml (souvent faux), pour une note finale pas très bonne, du fait d'une mauvaise organisation de leur travail.

Organisation du projet

Ce projet est composé de deux parties. La première est une étude des tris de listes, en particulier de leur efficacité en pratique, et la seconde, qui ne repose que partiellement sur la première, vise le calcul des enveloppes convexes proprement dit. Chacune de ces parties est composée :

1. d'un certain nombre de connaissances théoriques en programmation et en algorithmique qu'il vous faut acquérir, de manière autonome, pour pouvoir travailler ;
2. d'un ensemble minimal de fonctions qu'on vous demande de réaliser ;
3. de plusieurs tests permettant de vérifier la correction et l'efficacité de vos fonctions.

Evaluation du projet

Concernant l'évaluation du projet, elle sera établie sur la base de votre travail, d'un rapport écrit et d'une soutenance sur machine :
— Le rapport présentera l'ensemble des résultats de la première partie du projet (uniquement). Vous devrez l'écrire en LaTeX en utilisant les connaissances acquises lors de votre

enseignement d'Outils Logiciels (un squelette de fichier LaTeX sera mis à disposition sur Claroline au besoin).
— La soutenance concernera l'ensemble du projet, parties 1 et 2. Elle se déroulera en binôme, sur machine et pour une durée de 10 à 15 minutes. Dans un premier temps, vous devrez faire une démonstration de votre travail (ce qui exige une vraie préparation avant la soutenance, ne venez pas les mains dans les poches !). Dans un deuxième temps, vous devrez répondre aux questions que les encadrants vous poseront sur votre projet.

Les soutenances auront lieu le 18/05 après-midi. Le planning des soutenances (heure de passage de chaque groupe) sera géré par votre encadrant de TP.

Rendu du projet

Vous devrez rendre votre projet sur Claroline (dans la partie dédiée au rendu), au plus tard le 13/05 inclus (aucune soumission tardive ne sera acceptée). Un seul rendu devra être fait par binôme (par l'un des 2 étudiants au choix). Les fichiers à rendre seront les suivants :
— votre rapport au format pdf ;
— `list_tri.ml` ;
— `list_tri.mli` ;
— `enveloppe_convexe.ml`.

Mise en œuvre

Sur un plan plus technique, tous les fichiers dont vous aurez besoin sont dans la partie Projet du cours Programmation fonctionnelle de Claroline. Créez donc un répertoire `Projet` dans votre arborescence dédiée à OCaml, et enregistrez-y : `hasard.ml`, `hasard.mli`, `point.ml`, et `point.mli`.

Vous êtes libres de travailler sur les ordinateurs des salles de TP, ou sur votre portable personnel. Ceci vaut également pour la soutenance, et vous devrez vous assurer avant celle-ci que votre travail fonctionne parfaitement sur la machine choisie.

Concernant le système d'exploitation, utilisez plutôt `Linux` (ou `Unix`) et non `Windows` : les implantations de OCaml sous `Windows` marchent souvent mal et sont donc difficile à utiliser. De plus, vous risquez d'avoir des problèmes avec les fonctions de génération aléatoire et les fonctions graphiques qui vous sont données (cf. première et seconde parties du projet).

Concernant, pour finir, votre propre répertoire de travail et vos fichiers, sachez qu'il est du plus mauvais effet, en soutenance, de montrer qu'on est archi-bordélique. Soyez rigoureux :
— faites du ménage dans votre répertoire, de temps en temps, en effaçant les fichiers inutiles (et seulement eux . . .) ;
— faites des fichiers aérés, bien indentés, commentés, agréables à lire ;
— utilisez impérativement les noms des fonctions qui vous sont donnés dans le sujet, et choisissez des noms significatifs pour les autres fonctions (par exemple, les fonctions auxiliaires).

Rôle des encadrants

Notre rôle est double :

1. Nous règlerons les gros problèmes techniques liés à une erreur dans le sujet ou dans un des fichiers qu'on vous donne. Cela arrivera peut-être : nous vous prions de nous en excuser par avance.
2. Nous évaluerons votre travail à la fin du projet. Mais il n'y a pas de raison que cette évaluation ne se fasse que dans un sens : si vous avez des suggestions d'amélioration de ce projet, faites les nous parvenir pour que nous les utilisions l'an prochain.

Par contre, nous ne sommes pas là pour faire le projet à votre place. Cela signifie que nous *ne règlerons pas* vos problèmes de OCaml, à moins que vous ne bloquiez dessus *depuis plusieurs*

jours. Vous devrez vous débrouiller *seul* pour corriger vos fonctions : cela fait partie du travail d'un programmeur.

Un conseil, toutefois : si vous ne voulez pas avoir de problème, il faut *absolument* que vous testiez *rigoureusement* chacune de vos fonctions sur *de nombreux exemples*, afin d'être archi-sûr qu'elles fonctionnent correctement. Faites ce travail de débogage *chaque fois* que vous écrivez une nouvelle fonction et avant d'en écrire une autre. Sachez qu'il n'y a rien de plus difficile à faire que de corriger une erreur en fin de projet, quand cette erreur vient d'une fonction fausse qui a été écrite au tout début du projet.

1ère partie : sur les tris de listes

L'objectif de cette première partie du projet est double. Tout d'abord, elle consiste en une étude comparative des tris et de leur efficacité. Ensuite, elle vise à écrire un nouveau module de traitement de listes.

Les modules

Comme vous l'avez déjà vu en TP, OCaml fournit des bibliothèques de fonctions prédéfinies, qu'on appelle des modules, comme par exemple `String` et `List`. L'existence de ces modules présente de nombreux avantages. Par exemple, un programmeur peut utiliser ces fonctions quand il le désire, sans avoir à les (re-)programmer. De plus, l'implantation de ces fonctions est souvent d'une redoutable efficacité ; un programmeur a donc tout intérêt à les utiliser pour augmenter l'efficacité de ses propres programmes.

Pour utiliser une fonction définie dans un module, il suffit de préfixer le nom de cette fonction par le nom du module qui la contient. C'est ce que vous faites quand vous utilisez les fonctions `String.length` et `String.sub`. Cette écriture signifie que les fonctions `length` et `sub` sont définies dans le module `String`. À noter qu'il existe aussi une fonction `length` dans le module `List` permettant de calculer la longueur d'une liste. Pour l'utiliser, il suffira donc d'écrire `List.length`.

Pour éviter le préfixage de la fonction par le nom de son module, on peut aussi *ouvrir* le module avec la commande `open`. Mais il faut être prudent : un nom de fonction peut être défini dans plusieurs modules différents (comme `length`) sans désigner la même fonction pour autant. Cela peut donc introduire des confusions ...

```
# List.length [1; 2; 3] ;;
- : int = 3
# String.length "Je suis fan de caml" ;;
- : int = 19
# length [1; 2; 3] ;;
Unbound value length
# open List ;;
# length [1; 2; 3] ;;
- : int = 3
# open String ;;
# length [1; 2; 3] ;;
This expression has type 'a list but is here used with type string
```

Définir ses propres modules ...

En dehors des modules prédéfinis, OCaml permet aussi à un programmeur de *définir* ses propres modules et de les utiliser. C'est le cas du module `Hasard` que nous vous donnons (voir Annexe A).

Un module se compose toujours de deux parties :

1. Une partie *spécification*, qui regroupe les déclarations des types et des fonctions fournies par le module (bibliothèque de programmes).

2. Une partie *implantation*, qui contient les programmes associés aux fonctions du module.

Ces deux parties se trouvent le plus souvent dans deux fichiers distincts dont les noms sont significatifs. Ainsi, la spécification du module `Hasard` se trouve (nécessairement) dans le fichier `hasard.mli` et son implantation, dans le fichier `hasard.ml` (voir Annexe A). Si vous regardez ces deux fichiers de plus près, vous constatez que l'implantation contient bien du code en OCaml qui décrit chacune des fonctions déclarées dans la spécification. Ce fichier pourrait contenir d'autres fonctions non déclarées dans la spécification et qui seraient alors inutilisables en dehors du fichier. Cela permet de définir des fonctions auxiliaires qui sont invisibles à l'utilisateur du module.

Mais notez aussi que, sauf exception, vous n'êtes pas en mesure de comprendre parfaitement le code du fichier `hasard.ml`. C'est normal et ce n'est pas grave : si un module est bien conçu, que sa spécification est bien documentée, il n'est pas nécessaire qu'un programmeur sache comment les fonctions de ce module sont programmées pour les utiliser. Seule la spécification du module suffit. D'ailleurs, vous utilisez vous-mêmes des fonctions comme `String.length` et `String.sub` sans jamais vous être posé la question de savoir comment elles avaient été programmées.

Compilation et chargement d'un module

En OCaml, on utilise rarement des modules sans les avoir préalablement compilés. C'est le cas des modules prédéfinis `String` et `List`. Pour les autres modules, par contre, il faut effectuer cette opération de compilation "à la main", avec la commande `ocamlc -c`. Pour cela, on commence toujours par compiler la spécification du module, le `.mli`, ce qui génère un fichier `.cmi`. Il faut ensuite compiler son implantation, le `.ml`, ce qui génère un fichier `.cmo`. On peut ensuite utiliser le module dans l'interpréteur d'`OCaml`, en chargeant préalablement le fichier `.cmo` avec la commande [1] `#load`. Les modules prédéfinis sont, eux, automatiquement chargés lorsqu'on lance `OCaml`. On peut ensuite travailler :

```
meina.c2m.univ-st-etienne.fr> ls -l
-rw-r--r-- 1 samuel dep   765 janv. 11 14:06 hasard.ml
-rw-r--r-- 1 samuel dep   688 janv. 11 14:06 hasard.mli
meina.c2m.univ-st-etienne.fr> ocamlc -c hasard.mli
meina.c2m.univ-st-etienne.fr> ocamlc -c hasard.ml
meina.c2m.univ-st-etienne.fr> ls -l
-rw-r--r-- 1 samuel dep   339 janv. 11 14:08 hasard.cmi
-rw-r--r-- 1 samuel dep   855 janv. 11 14:08 hasard.cmo
meina.c2m.univ-st-etienne.fr> ocaml
        Objective Caml version 3.04
# #load "hasard.cmo" ;;
# open Hasard ;;
# init_random () ;;
- : unit = ()
# random_list 100 10 ;;
- : int list = [44; 53; 58; 16; 31; 10; 47; 45; 80; 86]
```

1. La commande `#use` permet de charger des fichiers *non* compilés, donc des fichiers `.ml`

Travail demandé

Notre objectif est de réaliser un nouveau module `List_tri` contenant de nouvelles fonctions sur les listes. Nous écrirons la spécification complète de ce module plus tard. En attendant, ouvrez un nouveau fichier `list_tri.ml` dans lequel vous devez absolument sauvegarder toutes vos fonctions.

La principale fonction que va fournir le module `List_tri` est une fonction de tri de listes qui doit être la plus efficace possible. Pour cela, nous allons en écrire plusieurs, puis choisir la meilleure. Ces fonctions doivent être génériques : elles vont trier n'importe quel type de données, selon n'importe quel ordre. Ceci signifie que la plupart des fonctions suivantes seront paramétrées par une fonction de comparaison de type `'a -> 'a -> bool` qui permettra d'indiquer quel ordre est choisi lors de l'exécution de ces fonctions.

Tri par création du maximum

1. Écrire une fonction `selectionne_max : ('a -> 'a -> bool) -> 'a list -> 'a`.
   `selectionne_max comp l` calcule le maximum de la liste `l` par rapport à la fonction de comparaison `comp` (passée en paramètre). Par exemple :

   ```
   # selectionne_max (<) [2;8;4;1];;
   - : int = 8
   # selectionne_max (>) [2;8;4;1];;
   - : int = 1
   ```

2. Écrire une fonction `supprime : 'a -> 'a list -> 'a list`.
   `supprime x l` retourne la liste `l` sans la première occurence de `x` quand elle existe, et ne fait rien sinon. Par exemple :

   ```
   # supprime 2 [1;2;4;1];;
   - : int list = [1; 4; 1]
   # supprime 2 [1;2;4;2;1];;
   - : int list = [1; 4; 2; 1]
   # supprime 2 [1;4;1];;
   - : int list = [1; 4; 1]
   ```

3. Écrire une fonction `ajoute_fin : 'a -> 'a list -> 'a list`.
   `ajoute_fin x l` retourne la liste `l` dans laquelle on a ajouté `x` en dernière position. Par exemple :

   ```
   # ajoute_fin 5 [4;2;5;1];;
   - : int list = [4; 2; 5; 1; 5]
   # ajoute_fin 5 [];;
   - : int list = [5]
   ```

4. Écrire une fonction
   `tri_creation_max : ('a -> 'a -> bool) -> 'a list -> 'a list`
   `tri_creation_max comp l` trie une liste conformément à l'odre passé en paramètre. Pour cela, cette fonction sélectionne récursivement le premier maximum, et l'ajoute à la fin du reste de la liste déjà triée (c'est-à-dire à la fin du rappel de la fonction sur la liste privée de son maximum). Par exemple,

   ```
   # tri_creation_max (<) [2;4;1];;
   - : int list = [1; 2; 4]
   # tri_creation_max (>) [2;4;1];;
   - : int list = [4; 2; 1]
   ```

5. Utiliser le module `Hasard` pour tester la fonction précédente sur des listes courtes, puis plus longues, des listes avec peu de doublons et des listes avec beaucoup de doublons.

6. Sauver (sans quitter) votre fichier `list_tri.ml`.

Tri par partition-fusion

1. Écrire une fonction `partitionne : 'a list -> ('a list * 'a list)`
   `partitionne l` partage la liste `l` en deux listes de tailles équivalentes, et les renvoie sous forme de couple. Pour cela, on met tous les éléments de rang pair dans la première, et tous les éléments de rang impair dans la seconde (et lorsqu'il n'y a qu'un seul élément, dans l'une ou l'autre).
2. Écrire une fonction
   `fusionne : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list`
   `fusionne comp l1 l2` prend deux listes triées et les fusionne en une seule liste, tout en préservant l'ordre `comp` passé en paramètre.
3. Écrire enfin la fonction
   `tri_partition_fusion : ('a -> 'a -> bool) -> 'a list -> 'a list`.
   `tri_partition_fusion comp l` partage `l` en deux listes de tailles équivalentes, trie chacune de ces deux listes récursivement, puis fusionne les deux résultats.
4. De même qu'avant, tester la fonction précédente sur des listes courtes. Vous pouvez ensuite la tester sur des listes plus longues, des listes avec peu de doublons et des listes avec beaucoup de doublons.
5. Sauver (sans quitter) votre fichier `list_tri.ml`.

Tri par arbre binaire de recherche

Relisez votre CM d'OCaml sur les arbres binaires et les arbres binaires de recherche.

1. Définir le type de données `'a arbreBinaire` comme vu en CM.
2. Écrire une fonction
   `insere_noeud : ('a -> 'a -> bool) -> 'a -> 'a arbreBinaire -> 'a arbreBinaire`
   `insere_noeud comp x a` insère à sa place le noeud `x` dans l'arbre binaire de recherche `a` par rapport à la fonction de comparaison `comp` (passée en paramètre). Par exemple :

```
# let a = Noeud(4,
                Noeud(2,ArbreVide,ArbreVide),
                Noeud(7,
                      Noeud(5,ArbreVide,ArbreVide),
                      Noeud(9,ArbreVide,ArbreVide)));;
# insere_noeud (<) 6 a;;
- : int arbreBinaire =
Noeud(4,
      Noeud(2, ArbreVide, ArbreVide),
      Noeud(7,
            Noeud(5,
                  ArbreVide,
                  Noeud(6, ArbreVide, ArbreVide)),
            Noeud(9, ArbreVide, ArbreVide)))
# let a2 = Noeud(4,
                 Noeud(7,
                       Noeud(9,ArbreVide,ArbreVide),
                       Noeud(5,ArbreVide,ArbreVide)),
                 Noeud(2, ArbreVide, ArbreVide));;
# insere_noeud (>) 6 a2;;
- : int arbreBinaire =
Noeud (4,
       Noeud(7,
```

```
            Noeud(9, ArbreVide, ArbreVide),
            Noeud(5,
                  Noeud(6, ArbreVide, ArbreVide),
                  ArbreVide)),
      Noeud (2, ArbreVide, ArbreVide))
```

Remarque : il est évident que l'ordre de tri `comp` passé en paramètre n'a de sens que si l'arbre binaire de recherche dans lequel on insère l'élément est déjà "trié" dans cet ordre (comme dans les exemples ci-dessus). Par exemple, dans les ABR vus en CM, tous les éléments du sous-arbre gauche sont inférieurs à la racine, et tous les éléments du sous-arbre droit sont strictement supérieurs à la racine, ce qui correspond à l'ordre ($<$).

3. Écrire une fonction
   `insere_liste_noeuds : ('a -> 'a -> bool) -> 'a list -> 'a arbreBinaire -> 'a arbreBinaire`
   `insere_liste_noeuds comp l a` insère récursivement à leur place chaque noeud de `l` dans l'arbre binaire de recherche `a` par rapport à la fonction de comparaison `comp` (passée en paramètre).
4. Écrire une fonction
   `parcours_arbre : 'a arbreBinaire -> 'a list`.
   `parcours_arbre a` parcourt `a` et crée la liste de ses noeuds. Le parcours doit permettre de rencontrer les noeuds dans l'ordre de tri représenté par l'arbre binaire de recherche. Par exemple :

   ```
   # parcours_arbre a;;
   - : int list = [2; 4; 5; 7; 9]
   # parcours_arbre a2;;
   - : int list = [9; 7; 5; 4; 2]
   ```
5. Écrire une fonction
   `tri_par_abr : ('a -> 'a -> bool) -> 'a list -> 'a list`.
   `tri_par_abr comp l` crée un arbre binaire de recherche avec les éléments de la liste en les insérant selon l'ordre de comparaison passé en paramètre, puis parcours dans le bon sens l'arbre pour constituer une liste triée.
6. De même qu'avant, tester la fonction précédente sur des listes courtes. Vous pouvez ensuite la tester sur des listes plus longues, des listes avec peu de doublons et des listes avec beaucoup de doublons.
7. Sauver (sans quitter) votre fichier `list_tri.ml`.

Choix d'une fonction de tri

Pour choisir la meilleure fonction de tri, vous devez tirer au hasard de grandes listes d'entiers, les trier avec vos 3 fonctions, puis déterminer l'algorithme qui vous semble le meilleur en terme d'efficacité. Attention, vous devrez montrer la pertinence de votre choix dans le rapport puis en soutenance. Soyez cohérents ...

Pour vous aider à faire ce choix, nous vous suggérons d'étudier le temps de calcul que nécessite chacun des algorithmes en fonction de la taille de la liste à trier. Pour faire ces mesures, utilisez le bout de code suivant ; il retourne le temps (en secondes) que met le processeur de l'ordinateur pour faire votre calcul :

```
# let temps_debut = Sys.time () in
  let _ = telle_fonction_de_tri ... in
  let temps_fin = Sys.time () in
  (temps_fin -. temps_debut) ;;
```

Une fois que vous avez choisi la meilleure fonction de tri, vous pouvez définir la fonction `tri : ('a -> 'a -> bool) -> 'a list -> 'a list` par un simple renommage (par exemple `let tri = tri_creation_max ;;`).

Définition du module `List_tri`

Avant de pouvoir définir le module `List_tri`, il va vous falloir ajouter deux autres fonctions dans `List_tri.ml`, dont nous nous servirons par la suite :

— une fonction `min_list : ('a -> 'a -> bool) -> 'a list -> 'a` qui retourne le plus petit élément d'une liste selon l'ordre passé en paramètre. Par exemple :
```
# min_list (<) [5;4;7;1;2];;
- : int = 1
# min_list (>) [5;4;7;1;2];;
- : int = 7
```

— une fonction `suppr_doublons : 'a list -> 'a list` qui prend une liste contenant d'éventuels doublons, et qui retourne cette liste sans ses doublons. Par exemple :
```
# suppr_doublons [1;2;3;1;2;4;2;1];;
- : int list = [1; 2; 3; 4]
```

Questions

1. Après avoir relu attentivement l'introduction précédente sur les modules, écrire le fichier `list_tri.mli` contenant la spécification du module `List_tri`. Il contiendra uniquement la spécification de 3 fonctions : `tri`, `min_list` et `suppr_doublons`. Compiler ce fichier puis vérifier que le fichier `list_tri.cmi` a bien été créé.
2. Compiler le fichier `list_tri.ml` puis vérifier que le fichier `list_tri.cmo` a bien été créé.
3. Relancer l'interpréteur `ocaml`, charger les modules `Hasard` et `List_tri`, puis vérifier une nouvelle fois que les trois fonctions du module `List_tri` fonctionnent bien correctement.

Rapport

De quatre à six pages, il doit reprendre l'essentiel de cette première partie du projet (et uniquement de cette partie).

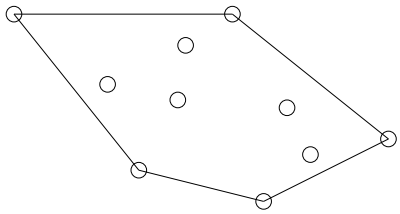
Quoiqu'il arrive, votre rapport ne contiendra aucun listing ni description linéaire des fonctions. Notez que vous pourrez l'utiliser comme support lors de votre soutenance. Dans ce rapport, vous devez justifier les choix que vous avez faits, et en particulier celui de votre fonction de tri. En conséquence, vous présenterez vos différentes expérimentations et les chiffres, tableaux, courbes, etc., qui en découlent.

Vous pouvez aussi étudier les propriétés des listes qui favorisent ou au contraire qui pénalisent vos fonctions de tri. Par exemple, comment se comportent-elles quand les listes qu'on leur donne sont déjà triées par ordre croissant ou par ordre décroissant ? Comment se comportent-elles en face de listes avec énormément de doublons ou avec très peu de doublons ? Pourquoi, selon vous ?

N'omettez surtout pas vos propres idées : plus elles seront originales, plus elles seront récompensées.

2nde partie :
Tracé d'enveloppes convexes

Les enveloppes convexes sont des objets géométriques qu'on rencontre fréquemment en informatique, par exemple, en recherche opérationnelle, en apprentissage automatique, en robotique, en statistiques inférentielles, etc. En voici la définition : soit E un ensemble de points du plan contenant au moins trois points non alignés ; on appelle *envelopppe convexe* de E le plus petit polygone convexe qui contient tous les points de E (voir figure). Clairement, les sommets de l'enveloppe convexe de E sont eux-mêmes des points de E.



L'étude des enveloppes convexes est un sous-domaine de la *géométrie algorithmique*. De nombreux algorithmes de construction ont été proposés, et l'objectif de ce projet est d'en implanter un qui a été développé par Barber et al. en 1996, et qui se nomme QUICKHULL.

L'algorithme Quickhull

On considère une liste L de points du plan sans doublon :

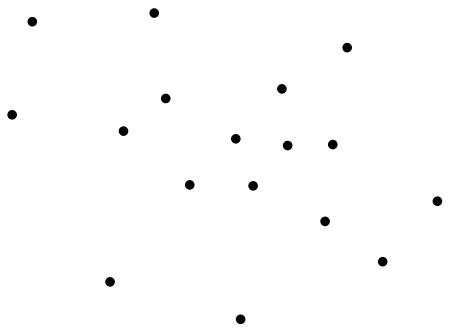


FIGURE 1 – `env = []`

L'enveloppe convexe sera une sous-liste de ces points, ordonnée de façon à ce qu'on puisse tracer dans l'ordre de rencontre chacun des segments de l'enveloppe. Au début, l'enveloppe convexe est égale à la liste vide, `env = []`.

1. Soit p_a le point de plus petite abscisse dans L, et p_b le point de plus grande abscisse dans L (si plusieurs points ont cette même plus petite/grande abscisse, peu importe celui qui est choisi). Ces points feront forcément partie de l'enveloppe convexe et sont donc mis dans la liste correspondante, `env =` $[p_a; p_b]$.

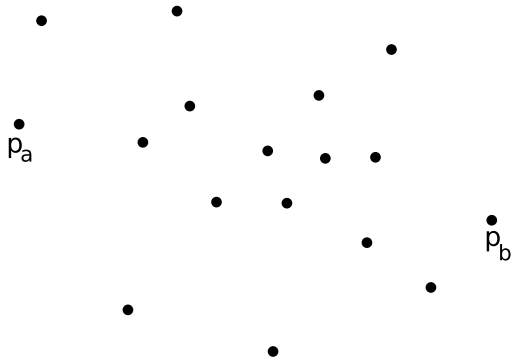
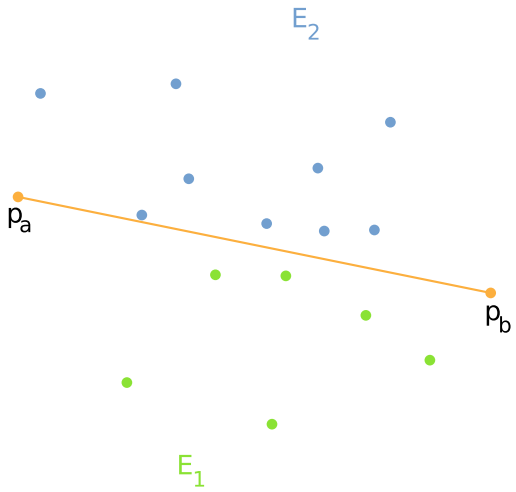


FIGURE 2 – `env =` $[p_a; p_b]$

2. la droite $(p_a p_b)$ partage les points du plan en 2 sous-ensembles E_1 et E_2. E_1 contient tous les points de L qui sont à droite de la droite orientée de p_a à p_b, et E_2 contient tous les points de L qui sont à droite de la droite orientée de p_b à p_a. Si des points sont sur la droite, ils peuvent être ignorés car ils ne feront jamais partie de l'enveloppe convexe) :



3. On considère ensuite les points de E_1. Il faut :
   (a) chercher le point de E_1 le plus éloigné de la droite orientée correspondante (donc celle allant de p_a à p_b). Soit p_c ce point, on l'ajoute dans la liste de l'enveloppe convexe, après le point de départ de la droite (donc après p_a), `env =` $[p_a; p_c; p_b]$.

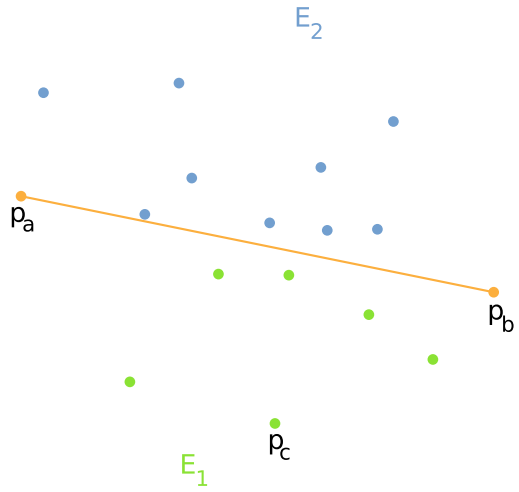
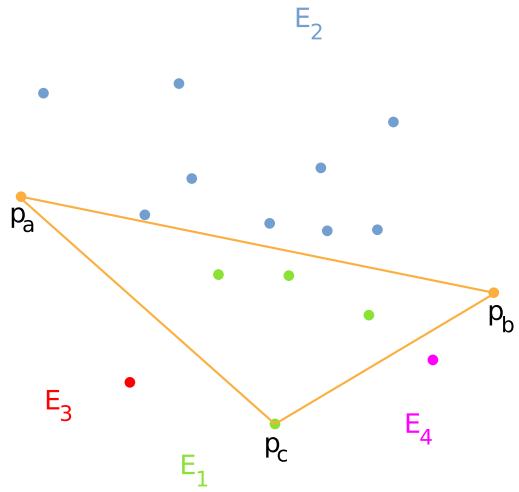


FIGURE 3 – `env` = $[p_a; p_c; p_b]$

(b) calculer l'ensemble E_3 des points de E_1 à droite de la droite orientée de p_a vers p_c, et l'ensemble E_4 des points à droite de la droite orientée de p_c vers p_b :



(c) répéter récursivement les étapes (a) et (b) pour E_3. On arrête la récursion quand l'ensemble considéré est vide. Ensuite, répéter ces mêmes étapes pour E_4.

Sur notre exemple, p_d est le point de E_3 le plus éloigné de la droite orientée de p_a vers p_c, il est ajouté à l'enveloppe convexe après p_a, `env` = $[p_a; p_d; p_c; p_b]$.

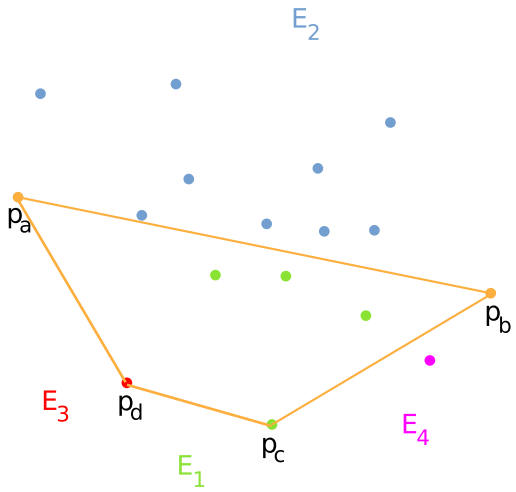


FIGURE 4 – `env` = $[p_a; p_d; p_c; p_b]$

On calcule l'ensemble des points à droite de la droite orientée de p_a vers p_d. Ici, il est vide. On passe donc à l'ensemble des points à droite de la droite orientée de p_d vers p_c. Il est vide également. On peut maintenant traiter E_4.

On cherche le point de E_4 le plus éloigné de la droite orientée de p_c vers p_b. Soit p_e ce point, on l'ajoute à l'enveloppe convexe après p_c, `env` = $[p_a; p_d; p_c; p_e; p_b]$.

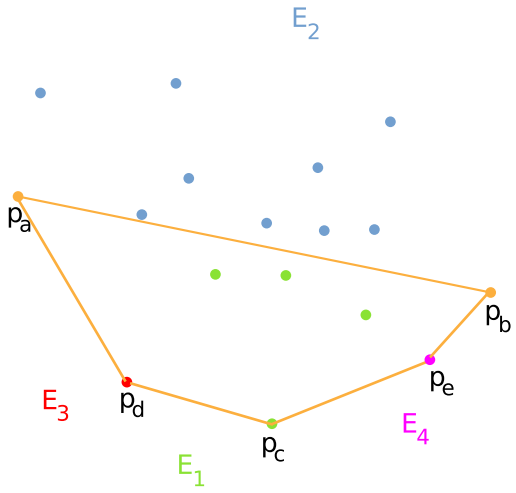


FIGURE 5 – `env` = $[p_a; p_d; p_c; p_e; p_b]$

On calcule l'ensemble des points à droite de la droite orientée de p_c vers p_e. Ici, il est vide. On passe donc à l'ensemble des points à droite de la droite orientée de p_e vers p_b. Il est vide également.

4. On répète l'ensemble du processus pour les points de E_2.

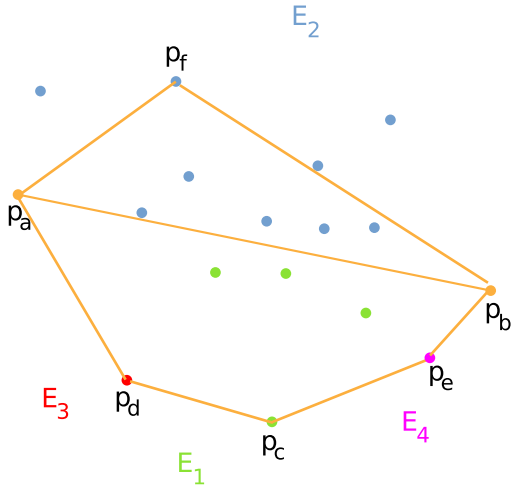


FIGURE 6 – `env =` $[p_a; p_d; p_c; p_e; p_b; p_f]$

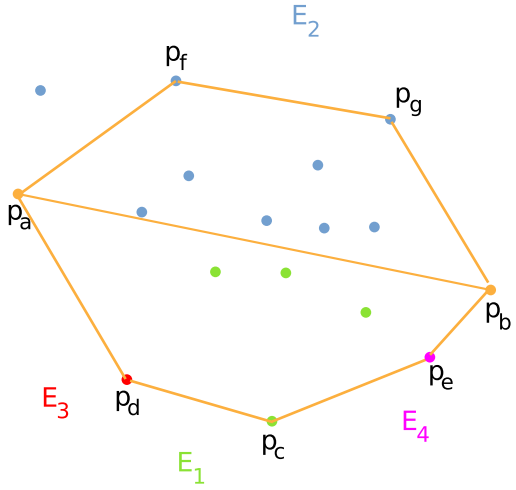


FIGURE 7 – `env =` $[p_a; p_d; p_c; p_e; p_b; p_g; p_f]$

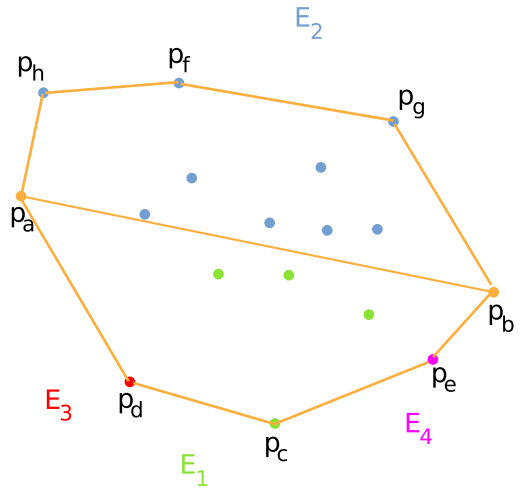


FIGURE 8 – `env` = $[p_a; p_d; p_c; p_e; p_b; p_g; p_f; p_h]$

Au final, les points contenus dans `env` = $[p_a; p_d; p_c; p_e; p_b; p_g; p_f; p_h]$ sont tous les points de l'enveloppe convexe. De plus, ils sont correctement ordonnés, et on peut directement tracer cette enveloppe en créant chaque segment (avec une fonction que l'on vous fournira).

Inutile de vous pendre . . .

. . . nous allons vous guider pas à pas pour réaliser ce programme. Nous vous fournirons même quelques fonctions.

Travail préparatoire

Comme nous l'avons dit, l'objectif de ce projet est d'implanter l'algorithme QUICKHULL. Pour cela, deux modules sont nécessaires : `List_tri` et `Point`. Vous trouverez la spécification de ce dernier dans l'Annexe B. Prenez le temps de la lire...

L'intérêt du module `Point` est de définir les types de base qui vont vous permettre de travailler, à savoir, le type `point`, le type `nuage` (ensemble de points) et le type `polygone` (liste ordonnée des sommets d'un polygone). En outre, ce module offre des fonctions permettant de générer aléatoirement des nuages de points et de les dessiner sur une fenêtre graphique.

Il vous faut maintenant compiler les deux fichiers du module `Point` (vous trouverez le `.ml` et le `.mli` sur Claroline, dans la partie dédiée au projet).

Ensuite, ouvrez un fichier `enveloppe_convexe.ml` qui contiendra toutes les fonctions du programme principal que vous programmerez par la suite.

Écrire directement au début de ce fichier les lignes vous permettant de charger le module de l'interface graphique et les modules `Point` et `List_tri` :

```
#load "graphics.cma";;
#load "point.cmo";;
open Point;;
#load "list_tri.cmo";;
open List_tri;;
open Graphics;;
```

Vous êtes maintenant prêts pour travailler.

Le cœur du programme

Calcul de p_a et p_b

Par définition, p_a (respectivement p_b) est le point de plus petite (respectivement plus grande) abscisse dans l'ensemble des points (si plusieurs points ont cette plus petite (respectivement grande) abscisse, alors on choisit le premier d'entre eux). Comme vous disposez de la fonction `min_list` du module `List_tri`, il suffit de définir les ordres `min_point` et `max_point` qui induisent cette notion de minimum (respectivement maximum).

Soient $p_1(x_1, y_1)$ et $p_2(x_2, y_2)$ deux points.
p_1 est plus petit que p_2 si $x_1 < x_2$.
p_1 est plus grand que p_2 si $x_1 > x_2$.

Questions

1. Écrire la fonction de comparaison `min_point : point -> point -> bool` qui implante l'ordre du plus petit point.
2. Écrire la fonction de comparaison `max_point : point -> point -> bool` qui implante l'ordre du plus grand point.
3. Écrire la fonction `points_depart : point list -> point * point` qui prend une liste de points et renvoie le plus petit et le plus grand, sous forme de couple. Cette fonction se sert de `min_list`.

Testez vos fonctions pour vérifier qu'elles sont correctes !

Ensemble de points à droite d'une droite orientée

Connaissant p_a et p_b, il faut maintenant constituer E_1 et E_2. E_1 contient tous les points de l'ensemble qui sont à droite de la droite orientée de p_a à p_b, et E_2 contient tous les points de l'ensemble qui sont à droite de la droite orientée de p_b à p_a.

Soient $p_1(x_1, y_1)$, $p_2(x_2, y_2)$ et $p_3(x_3, y_3)$ trois points. Posons

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

Si $c < 0$, alors p_3 est à droite de la droite orientée de p_1 vers p_2.

Question

Écrire la fonction `points_droite : point list -> point -> point -> point list`.
`points_droite l p1 p2` renvoie la liste des points de `l` qui sont à droite de la droite orientée de p_1 vers p_2.

Point le plus éloigné d'une droite

Il nous faut maintenant trouver le point de E_1 le plus éloigné de $(p_a p_b)$.

Questions

1. Commençons par déterminer l'équation cartésienne d'une droite passant par 2 points $p_1(x_1, y_1)$ et $p_2(x_2, y_2)$. Cette équation est de la forme $ax + by + c = 0$. Il nous faut donc trouver les valeurs de a, b et c.
   Nous savons que $\overrightarrow{p_1 p_2}$ est un vecteur directeur de $(p_1 p_2)$, et que

   $$\overrightarrow{p_1 p_2} = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} = \begin{pmatrix} -b \\ a \end{pmatrix}$$

   De plus, comme p_1 appartient à $(p_1 p_2)$, on peut alors remplacer les valeurs de a, x, b, y dans $ax + by + c = 0$ et trouver la valeur de c.

   Écrire la fonction `equation_droite : point -> point -> int * int * int` qui calcule l'équation de la droite passant par deux points, i.e. qui renvoie les valeurs de a, b et c sous forme de triplet.
2. Calculons maintenant la distance d'un point à une droite. Soient une droite d'équation $ax + by + c = 0$ et un point $p(x, y)$. La distance de p à la droite est

   $$\frac{|ax + by + c|}{\sqrt{a^2 + b^2}}$$

   Écrire la fonction `distance_droite : int -> int -> int -> point -> float`.
   `distance_droite a b c p` calcule la distance de p à la droite d'équation $ax + by + c = 0$.
3. Un point p_a est plus éloigné d'une droite $(p_1 p_2)$ qu'un point p_b si la distance de p_a à la droite est supérieure à la distance de p_b à la droite.

   Écrire la fonction de comparaison `distance_max : point -> point -> point -> point -> bool` qui implante l'ordre du point le plus éloigné.
   `distance_max p1 p2 pa pb` renvoie vrai si p_a est plus éloigné de $(p_1 p_2)$ que p_b.
4. Écrire la fonction `point_eloigne : point -> point -> point list -> point`.
   `point_eloigne p1 p2 l` renvoie le point de `l` le plus éloigné de la droite $(p_1 p_2)$. Cette fonction se sert de `min_list`.

Ajouter un point dans une liste à sa place

Il nous manque encore une fonction permettant d'ajouter un point dans l'enveloppe convexe à sa place, c'est-à-dire après un autre point.

Questions

1. Deux points sont égaux s'ils ont la même abscisse et la même ordonnée. Écrire la fonction de comparaison `points_egaux : point -> point -> bool` qui renvoie vrai si les deux points sont égaux.
2. Écrire la fonction `ajoute_list_apres : point -> point -> point list -> point list`.
   `ajoute_list_apres p1 p l` ajoute p dans la liste l, en le plaçant juste après le point p_1.

Fonction principale

Vous y êtes ! Il reste à faire la fonction principale, qui prend une liste de points et renvoie la liste des points de son enveloppe convexe.

Celle-ci est un peu complexe, alors nous avons décidé de vous la donner. Elle se décompose en deux fonctions, que vous pouvez recopier dans votre fichier `enveloppe_convexe.ml`. Attention, cependant : tâchez de comprendre ce que font ces deux fonctions, en les lisant attentivement et en reprenant les explications de fonctionnement de l'algorithme. Des précisions pourraient vous être demandées lors de la soutenance...

```
let rec findhull l p q enveloppe =
match l with
[] -> enveloppe
| x::r -> let c = point_eloigne p q l in
          let enveloppe = ajoute_list_apres p c enveloppe in
          let enveloppe = findhull (points_droite l p c) p c enveloppe in
          findhull (points_droite l c q) c q enveloppe;;


let quickhull l =
let l = suppr_doublons l in
let (pa,pb) = points_depart l in
let enveloppe = pa::(pb::[]) in
let enveloppe = findhull (points_droite l pa pb) pa pb enveloppe in
findhull (points_droite l pb pa) pb pa enveloppe;;
```

Expérimentations

Testez toutes les fonctions ci-dessus, sur plusieurs exemples, pour vous assurer qu'elles fonctionnement correctement. C'est seulement après avoir fait cela que vous pourrez les utiliser pour construire et visualiser des enveloppes convexes. Vous chargerez votre fichier `enveloppe_convexe.ml` dans l'interpréteur OCaml avec la commande `#use "enveloppe_convexe.ml";;` Ceci doit charger vos fonctions sans générer une seule erreur. Vous utiliserez ensuite les fonctions de génération aléatoire d'ensembles de points du module `Point` pour créer vos ensembles de points sur lesquels calculer l'enveloppe convexe.

La fonction suivante vous simplifiera sans doute la vie (vous pouvez l'ajouter dans votre fichier) :

```
# let enveloppe_convexe g n =
let l = g n in
(vider ();
set_color red;
tracer_nuage l ;
set_color blue;
tracer_polygone(quickhull l));;
val enveloppe_convexe : ('a -> nuage) -> 'a -> unit = <fun>
```

Et ensuite vous pourrez tester votre fonction finale (par exemple) :

```
# initialiser();;
# enveloppe_convexe gen_soleil 1000;;
```

Soutenance

Elle se déroulera en binôme, sur machine, pendant 10 à 15 minutes. Vous devrez présenter l'ensemble de votre projet, sur la base du rapport d'une part (vous pouvez faire une démonstration de vos fonctions de tri), puis en montrant que vos fonctions de la seconde partie permettent effectivement de construire des enveloppes convexes. *Préparez votre démonstration* car c'est vous qui mènerez la barque pendant la soutenance. Essayez de mettre en valeur les atouts de votre projet. En un mot, essayez de vous vendre, ou mieux, essayez de nous surprendre [2]. Préaprez-vous aussi pour répondre à d'éventuelles questions.

2. Pour un encadrant, c'est très long, une journée complète de soutenances, surtout quand on a l'impression de voir encore et toujours les mêmes choses ...

Annexe A

Module Hasard

A.1 Fichier `hasard.mli`

```
(* hasard.mli *)

      (**********************************)
      (*                                *)
      (* Spécification du module Hasard *)
      (*                                *)
      (**********************************)

(************************************************)

val init_random : unit -> unit

(* (init_random ()) permet d'initialiser
   le générateur aléatoire. On ne l'utilise
   qu'une fois en début d'une session d'ocaml. *)

(************************************************)

val random_list : int -> int -> int list

(* (random_list b n) retourne une liste de
   n entiers tirés au hasard entre 0 et (b - 1). *)

(************************************************)
```

A.2 Fichier `hasard.ml`

```
(* hasard.ml *)

        (*********************************)
        (*                               *)
        (* Implantation du module Hasard *)
        (*                               *)
        (*********************************)

open Sys ;;

(************************************************)

let init_random () =
  let s = "/tmp/la_date_de_" ^ (getenv "LOGNAME") in
  let _ = command ("date +\"%M%H%j\" > " ^ s) in
  let c = open_in s in
  let n = int_of_string (input_line c) in
  ( close_in c ; remove s ; Random.init n ) ;;

(************************************************)

let rec random_list b n =
  if n <= 0
  then []
  else (Random.int b)::(random_list b (n - 1)) ;;

(************************************************)

let je_ne_suis_pas_déclarée_dans_la_spécification =
"donc on ne peut m'utiliser en dehors de ce fichier" ;;

(************************************************)
```

Annexe B

Module Point

Fichier `point.mli`

```
(* point.mli *)

              (*******************************)
              (*                             *)
              (* Spécification du module Point *)
              (*                             *)
              (*******************************)

(*************************************************************)

type point = { x : int ; y : int }

type nuage = point list

type polygone = point list

(* un point est un enregistrement de coordonnées ; un nuage est
   un ensemble de points rangés dans une liste ; un polygône est
   représenté par la liste (ordonnée) de ses sommets consécutifs *)

(*************************************************************)

val gen_rectangle   : int -> nuage
val gen_cercle      : int -> nuage
val gen_papillon    : int -> nuage
val gen_cerf_volant : int -> nuage
val gen_soleil      : int -> nuage
val gen_poisson     : int -> nuage

(* génèrent aléatoirement des nuages de points ; le paramètre
   des fonctions précédente est le nombre de points du nuage ;
   chacune d'elles produit un nuage d'une forme différente ;
   les nuages générés peuvent comporter des doublons *)

(*************************************************************)
```

```
(* Fonctions graphiques : *)

val init     : unit -> unit
val vider    : unit -> unit
val terminer : unit -> unit

val tracer_point : point -> unit
(* dessine un point sur la fenêtre graphique *)

val tracer_nuage : nuage -> unit
(* dessine tous les points d'un nuage sur la fenêtre graphique *)

val tracer_polygone : polygone -> unit
(* trace un polygône sur la fenêtre graphique *)

(*****************************************************************)
```