TP sur la méthode de descente

Il s'agit de programmer en Java une méthode de descente dans le cas particulier suivant.
On considère une fonction f de **deux** variables réelles et un domaine D du plan **délimité par des fonctions affines**. On suppose que f est **convexe** et continûment dérivable sur un ouvert contenant le domaine D. Il s'agit de trouver le **minimum** de la fonction f sur D.
Le problème pourait être par exemple :

Minimiser la fonction $f(x, y) = \exp(x + y) + x^2 + 2\,y^2$
dans le domaine défini par : $-2x + y \leq 0$
$-y \leq 0$
$x + y \leq 4$

Il faudra compléter une classe d'un programme.

Les contraintes du problème sont modélisées sous la forme de **contraintes de négativité**, sous la forme $ax + by + c \leq 0$. Le vecteur gradient calculé par la méthode `getGradient()` de la classe est le vecteur (a, b).

Si vous souhaitez voir le résultat à atteindre, téléchargez descenteExecutable.jar. Vous pouvez alors exécuter le programme avec la commande
java -jar descenteExecutable.jar

Pour faire tourner l'application, il faut choisir un problème en indiquant un numéro compris entre 1 et 14 ; après avoir validé le problème, le domaine s'affiche ; il faut alors choisir un point de départ de la méthode de descente à l'intérieur du domaine puis appuyer sur le bouton "Demarrer". La méthode de descente s'exécute. Au cours de la méthode, pour chaque point obtenu au cours de la méthode, le vecteur gradient est dessiné en bleu et la prochaine direction à suivre est en vert. La trajectoire engendrée par la méthode de descente est tracée en rouge au fur et à mesure.

Il faut commencer par sauvegarder le fichier descenteIncomplet.jar sur votre ordinateur.

Après avoir lancé Eclipse, pour créer le projet :

- dans "File", faire New puis Java Project : cela ouvre une fenêtre ;
- dans le cadre "Project name", mettre le nom de son choix ;
- si le cadre "JRE" de la fenêtre affiche à droite "jdk-11.0.19", passer directement à la dernière instruction de cette liste ;
- sinon, faire tout ce qui suit :
- dans le cadre JRE, sélectionner "Use a project specific JRE" ; à droite, cliquer sur "configure JREs..." (sous les deux petits cadres) : cela ouvre une fenêtre appelée "Preferences"
- dans la fenêtre "Preferences", cliquer sur le bouton "Add" : cela ouvre une fenêtre appelée "Add JRE" ;
- dans la fenêtre "Add JRE", sélectionner "Standard VM" puis cliquer sur "Next" : cela ouvre une fenêtre de nouveau appelée "Add JRE" ;
- dans "JRE home", écrire /cal/softs/java/jdk-11.0.19/ puis cliquer sur Finish : cela ramène à la fenêtre "Preferences" ; cliquer alors sur "Apply and close" ; cela ramène à la fenêtre "New Java Project" ;
- dans le cadre JRE, au niveau de la ligne (sélectionnée) "Use a project specific JRE", choisir "jdk-11.0.19 à l'aide du triangle noir pointé vers le bas ;
- cliquer en bas sur "Finish" ; cela ouvre une fenêtre "New module-info.java" ; dans cette fenêtre, **choisir "Don't create" afin de NE PAS CREER DE MODULE**.

Si vous travaillez sur votre ordinateur, il faut peut-être signaler de ne pas utilliser de "module" dans la page où on donne le nom du projet.

Puis :

- cliquer sur **src** **sous le nom du projet** avec le bouton de droite puis choisir "import" ;
- dans la fenêtre obtenue, dans "General", choisir "Archive File" ;

- après avoir fait "Next", choisir le fichier descenteACompleter.jar ;
- appuyer sur "Finish".

Vous pouvez essayer d'exécuter la méthode `main` qui se trouve dans la classe `Main` dans le paquetage `descente`.

Le travail à effectuer consiste uniquement à compléter la classe `Descente` du paquetage `descente.modele`. Dans cette classe, six méthodes sont à compléter (voir les commentaires dans le fichier à compléter pour la description de ces méthodes ; ces commentaires précèdent chacune des méthodes à compléter) : pour la méthode `directionASuivreSiCoin`, des explications sont fournies plus bas dans cette page.

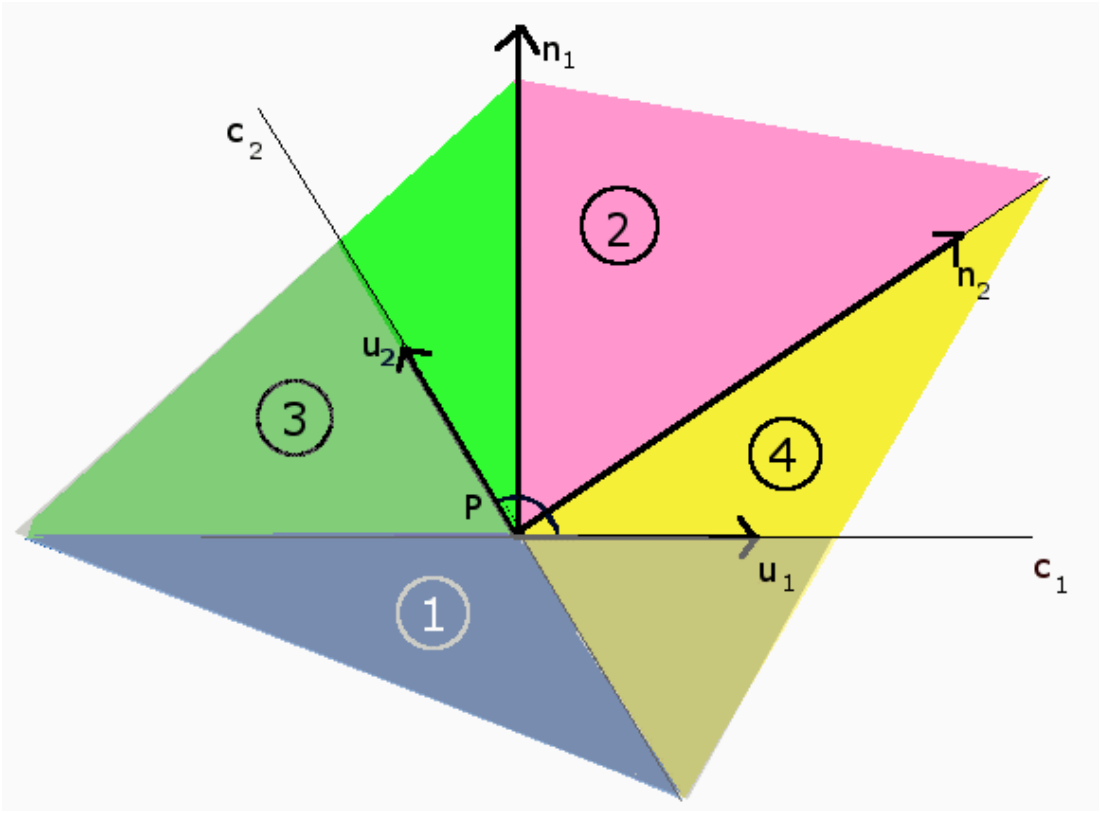
- la méthode `directionASuivreSiInterieur` ;
- la méthode `directionASuivreSiBord` ;
- la méthode `directionASuivreSiCoin` ;
- la méthode `chercheSecondPoint` ;
- la méthode `dichotomie` ;
- la méthode `KarushKuhnTucker`.

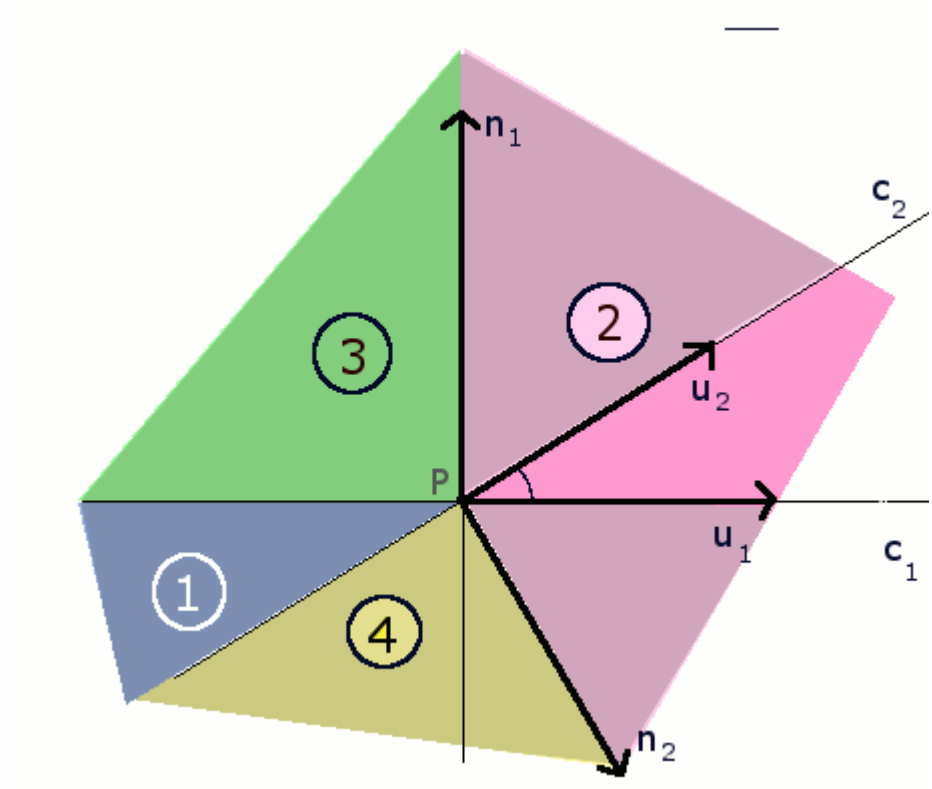
Si la norme du gradient de la fonction étudiée est inférieure à l'attribut `seuil` de la classe `descente` , la méthode de descente est considérée comme terminée ; d'autres cas de terminaison se produisent lorsqu'on est sur un bord ou en un coin du domaine.

Il faudra faire attention aux attributs suivants de la classe `Descente` :

- l'attribut `directionASuivre` devra contenir la direction que doit suivre la méthode de descente à partir du point courant ;
- l'attribut `fini` devra être positionné à `true` lorsqu'on considère que la méthode est terminée.

Explications pour la fonction `directionASuivreSiCoin`





Sur les dessins, deux cas sont représentés : le cas d'un coin obtus et le cas d'un coin aigu. Dans les deux cas, le point courant P se trouve dans le coin. Les droites frontières des contraintes sont désignées par c_1 et c_2. Le domaine réalisable est compris entre ces deux droites. Le domaine non réalisable est légèrement grisé.

Le vecteur n_1 est égal à $-\nabla g_1(P)|$ si la première contrainte s'écrit $g_1(x, y) \leq 0$; le vecteur n_2 est égal à $-\nabla g_2(P)|$ si la seconde contrainte s'écrit $g_2(x, y) \leq 0$.

Les vecteurs u_1 et u_2 sont les vecteurs unitaires des bords situés du côté du domaine réalisable ; **u_1 doit être orienté vers le côté négatif de la contrainte c_2** (il faut faire en sorte que le produit scalaire de u_1 avec n_2 soit positif), **u_2 doit être orienté vers le côté négatif de la contrainte c_1** (il faut faire en sorte que le produit

scalaire de u_2 avec n_1 soit positif).

Notons ∇f le gradient de f au point P.

- *Premier cas* : ∇f appartient à la zone 1, en bleu grisé. La direction admissible de plus grande pente est $-\nabla f$. La décomposition de ∇f sur les vecteurs u_1 et u_2 permet de détecter ce cas.

- *Deuxième cas* : ∇f appartient à la zone 2, en rose grisé ou non, la condition de Karush, Kuhn et Tucker est vérifiée. La décomposition de ∇f sur les vecteurs n_1 et n_2 permet de détecter ce cas.

- *Troisième cas* : on n'est pas dans les deux cas précédents et donc ∇f appartient à la zone 3, verte, ou à la zone 4, jaune, grisées ou non. Dans ce cas, ce sera un des deux vecteurs u_1 ou u_2 qui donne la direction de descente admissible de plus grande pente. On rappelle qu'une direction d descend d'autant plus qu'elle fait un plus grand angle avec ∇f, c'est-à-dire que le produit scalaire de d avec ∇f est plus petit.

ATTENTION : ne rien modifier en dehors de la classe `Descente`.

Vous disposez d'une documentation.

Les classes et méthodes utiles pour le travail à faire sont en particulier les suivantes :

- La classe `descente.modele.Descente` modélise la descente ; c'est dans cette classe que se trouvent les six méthodes à compléter ; des explications se trouvent dans le fichier `Descente.java` à compléter .

- La classe `descente.modele.Couple` modélise un couple de deux éléments de type `double` ; elle sert à modéliser un point du plan réel ou un vecteur de ce même plan. Les méthodes suivantes de cette classe peuvent être utiles :

  - la méthode `double produitScalaire(Couple v)` : si `v1` et `v2` sont deux vecteurs de type `Couple`, alors `v1.produitScalaire(v2)` donne le produit scalaire de `v1` avec `v2` ;

  - la méthode `double norme()` : si `v` est un vecteur de type `Couple`, alors `v.norme()` donne la norme de `v` ;

  - la méthode `boolean estPerpendiculaire(Couple v)` : si `v1` et `v2` sont deux vecteurs de type `Couple`, alors `v1.estPerpendiculaire(v2)` vaut `true` si `v1` et `v2` sont perpendiculaires, `false` sinon ;

  - la méthode `Couple mult(double t)` : si `v` est un vecteur de type `Couple` et si `t` est un double, alors `v.mult(t)` retourne le vecteur de type `Couple` obtenu en multipliant `v` par `t` ; `v` n'est pas modifié par cette multiplication ;

  - la méthode `Couple ajoute(Couple v)` : si `v1` et `v2` sont deux vecteurs de type `Couple`, alors `v1.ajoute(v2)` retourne le vecteur de type `Couple` obtenu en ajoutant `v1` à `v2` ; `v1` n'est pas modifié par cet ajout ;

  - la méthode `statique Couple decompose(Couple v, Couple v1, Couple V2)` : si `v`, `v1` et `v2` sont trois vecteurs de type `Couple`, alors `Couple.decompose(v, v1, v2)` retourne les deux composantes de `v` sur le repère formé par `v1` et `v2` sous la forme d'un `Couple` ; si `v1` et `v2` sont parallèles, la méthode retourne `null`.

- La classe `descente.modele.Contrainte` modélise une contrainte affine qui s'écrit : `coeffx * x + coeffy * y + constante <= 0` ; cette classe modélise aussi un demi-plan. La droite

d'équation : `coeffx * x + coeffy * y + constante = 0` est la droite frontière du demi-plan. Les méthodes suivantes de cette classe peuvent être utiles :

- la méthode `Couple getGradient()` : si `c` est de type `Contrainte`, alors `c.getGradient()` retourne le gradient de la fonction `(x, y) -> coeffx * x + coeffy * y + constante`, c'est-à-dire le vecteur de composantes `coeffx` et `coeffy` ;
- la méthode `Couple getVecteurUnitaireBord()` : si `c` est de type `Contrainte`, alors `c.getVecteurUnitaireBord()` donne un vecteur unitaire parallèle à la droite frontière.

- La classe `descente.modele.Domaine` modélise un domaine défini par un ensemble de contraintes affines. On suppose qu'il n'existe pas trois droites déterminant les contraintes qui se coupent en un même point (sinon le code pourrait ne pas fonctionner). Les méthodes suivantes de cette classe peuvent être utiles :
  - la méthode `Contrainte estSurBord(Couple P)` : si `D` est de type `Domaine` et si `P` est un point de type `Couple`, alors `D.estSurBord(P)` renvoie :
    - si `P` est sur un bord du domaine, la méthode retourne une contrainte (de type `Contrainte`) correspondant à ce bord ; si P est en un coin du domaine, la méthode retourne une des deux contraintes correspondant à ce coin ;
    - si `P` n'est pas sur un bord du domaine, la méthode renvoie la valeur `null` ;
  - la méthode `Contrainte[] estCoin(Couple P)` : si `D` est de type `Domaine` et si `P` est un point de type `Couple`, alors `D.estCoin(P)` renvoie :
    - si `P` est en un coin du domaine, les deux contraintes correspondant à ce coin (sous forme d'un `tableau de type Contrainte[]` à deux cases) ;
    - si `P` n'est pas en un coin, la valeur `null`.
- La classe abstraite `probleme.Pb` modélise un problème, avec la définition de la fonction *f* à minimiser et l'ensemble des contraintes définissant le domaine. Tous les problèmes sont définis par des classes héritant de la classe `probleme.Pb`. Les méthodes suivantes de cette classe pourront être utilisées :
  - la méthode `double phiDerivee(Couple P0, Couple d, double t)` : si `pb` est de type `Pb`, si t -> P0 + td (t > 0) est l'équation paramétrique d'une demi-droite, alors `pb.gPrime(P0, d, t)` retourne la valeur de la dérivée en `t` de la fonction `g : t -> g(t) = pb.f(P0 + td))`.
  - la méthode `Couple gradientf(Couple P)` : si `pb` est de type `Pb`, si `P` est un point de type `Couple`, alors `pb.gradientf(P)` renvoie le gradient de la fonction `f` au point `P`.

Quand vous aurez implémenté correctement la méthode `directionASuivreSiInterieur`, vous pourrez regarder ce qui se passe avec le problème 1 pour voir si cela commence bien ; après avoir implémenté `directionASuivreSiBord`, vous pourrez regarder ce qui se passe avec le problème 1 pour voir si cela se passe bien sur un bord ; après avoir implémenté `directionASuivreSiCoin`, vous pourrez vérifier ce qui se passe avec les problèmes 1, 4 et 6.
Après avoir implémenté la méthode `chercherSecondPoint`, vous pourrez regarder ce qui se passe avec le problème 3 (qui n'atteint pas de minimum dans le domaine).
Après avoir implémenté la méthode `dichotomie`, vous pourrez tester tous les problèmes.