

# Master Maths 1 2006-2007

## Optimisation

### TP2 : simplexe

Pour le déroulement de cette deuxième séance, vous devez tout d'abord finir les exercices de la première feuille, surtout ceux sur l'écriture de fonctions en langage scilab et plus particulièrement celui portant sur la génération de P.L. qui vous serviront à tester votre code de simplexe (en plus des exemples vus en TD).

Ce TP représente un assez gros travail et se fera sur 2 séances 1/2 (cad aujourd'hui + 2 séances supplémentaires).

## 1 Rappels sur l'algorithme du simplexe (révisé ou non)

On considère le P.L. en forme standard :

$$\left\{ \begin{array}{l} \min \quad cx \\ x \in \mathbb{R}^n \\ Ax = b \\ x \geq 0 \end{array} \right. \quad \text{avec} \quad \begin{array}{l} A \in \mathbb{R}^{m \times n} \\ b \in \mathbb{R}^m \\ c \in \mathbb{R}^{n^*} = \mathbb{R}^{1 \times n} \end{array}$$

dont on suppose connue une solution de base réalisable.  $J$  désignera l'ensemble des indices de base et  $K$  (noté  $\bar{J}$  dans le cours) l'ensemble des indices hors base. D'autre part  $A^J$  désignera la sous-matrice  $m \times m$  correspondant à la base (obtenue par juxtaposition des colonnes d'indices de base) et  $A^K$  celle formée des colonnes hors base. On utilisera le même genre de découpage par blocs pour  $c$  et  $x$ , la contrainte  $Ax = b$  s'écrivant alors :

$$A^J x_J + A^K x_K = b \iff x^J = (A^J)^{-1}(b - A^K x_K)$$

et le coût :

$$f(x) = c^J x_J + c^K x_K$$

Remarquons que la solution de base actuelle (celle impliquée par le choix de  $J$ ), notée  $\underline{x}$  se décompose en  $\underline{x}_J = (A^J)^{-1}b$  et  $\underline{x}_K = 0$ .

Pour progresser ou s'arrêter dans l'algorithme, on se pose toujours les questions suivantes :

1. Quelle variable a-t-on intérêt à faire entrer dans la base de façon à faire diminuer le coût ? Pour cela il faut l'exprimer en fonction des variables libres (hors base), ce qui est très simple :

$$\begin{aligned} f(x) &= c^J (A^J)^{-1}(b - A^K x_K) + c^K x_K \\ &= c^J (A^J)^{-1}b + (c^K - c^J (A^J)^{-1}A^K)x_K \\ &= f(\underline{x}) + c_R x_K \end{aligned}$$

Maintenant si tous les coûts réduits sont positifs ou nuls, il n'est alors plus possible de diminuer le coût en augmentant une variable hors base, l'algorithme se termine **normalement** ; dans le cas contraire cela est *a priori* encore possible et la stratégie usuelle consiste à choisir la variable hors base qui a le coût réduit le plus petit possible (cad le plus négatif possible !). On notera  $i \in K$  l'indice de cette variable "entrante".

2. une fois  $i$  choisi, la nouvelle question est : “Quelle variable quitte la base ?” Intuitivement, on s’attend à ce qu’en augmentant  $x_i$ , au moins une variable de  $x_J$  diminue. Si c’est le cas, on augmentera alors  $x_i$  jusqu’à annuler une des variables de base qui sera alors la variable “sortante”. Répondre à cette question est assez facile puisque maintenir la relation  $Ax = b$  en ne s’autorisant qu’à augmenter  $x_i$  (et pas les autres variables hors base), s’écrit :

$$A^J x_J + A^i x_i = b \iff x_J = (A^J)^{-1}(b - A^i x_i) \iff x_J = \underline{x}_J - (A^J)^{-1} A^i x_i = \underline{x}_J - z x_i$$

où  $z = (A^J)^{-1} A^i \in \mathbb{R}^m$ . Il est clair que seules les variables de base correspondant à des valeurs de  $z$  strictement positives vont “aller” vers zéro. Ainsi si le vecteur  $z$  a toutes ses composantes négatives ou nulles, on peut alors augmenter  $x_i$  *ad vitam eternam*, ce qui implique que la fonction coût n’est pas bornée inférieurement sur l’ensemble admissible (il faut alors arrêter l’algorithme en renvoyant cette information). Sinon, la variable qui sort est celle pour laquelle le rapport :

$$\frac{x_k^J}{z_k}, \quad k = 1, \dots, m, \quad z_k > 0$$

est le plus petit possible<sup>1</sup>. On notera  $q$  l’indice de la variable sortante.

## 2 simplexe classique versus simplexe révisé

Dans la méthode classique du simplexe (cf TDs), les calculs précédents sont simplifiés du fait qu’on s’arrange toujours pour que “ $A^J = I$ ”, en fait en multipliant le système  $Ax = b$  par  $(A^J)^{-1}$ . Bien sûr c’est une vue algébrique des choses et dans la pratique on maintient en fait “l’identité sous la base” à l’aide de transformations de Gauss. De même pour chaque itération, on déduit les nouveaux coûts réduits assez facilement en fonction des anciens...

Cette méthode a le défaut de cumuler les erreurs d’arrondi si on travaille en arithmétique flottante. D’un point de vue complexité (nombre d’opérations) elle perd aussi vis à vis de la méthode du simplexe révisé du moins dans les cas usuels (cad quand la matrice  $A$  comporte beaucoup de zéros, ou encore lorsque  $m \ll n$ ).

Dans le simplexe révisé (qui est conceptuellement plus simple), on reste très proche des relations précédentes. Enumérons les “gros” calculs correspondant à une étape du simplexe :

1. calcul de la solution de base : il faut résoudre le système linéaire  $A^J \underline{x}_J = b$
2. calcul des coûts réduits : il faut calculer  $\tilde{c}^J = c^J (A^J)^{-1}$ , ce qui peut se voir comme, “résoudre le système linéaire”  $(A^J)^\top (\tilde{c}^J)^\top = (c^J)^\top$  puis ensuite on a :  $c_R = c^K - (\tilde{c}^J) A^K$ . Attention il faut bien faire les opérations dans cet ordre, cad  $(c^K - (c^J (A^J)^{-1}) A^K)$  et pas dans cet ordre  $(c^K - c^J ((A^J)^{-1}) A^K)$ , pourquoi ? ;
3. la variable sortante : il faut calculer le vecteur  $z$ , cad résoudre le système linéaire  $A^J z = A^i$ .

Soient donc 3 systèmes linéaires à résoudre de même matrice<sup>2</sup>  $A^J$ . Si vous vous souvenez un peu de vos cours d’analyse numérique de base, cela fait penser à “factorisation LU”. C’est exactement ce que font les codes modernes de simplexe. Les 3 systèmes linéaires sont résolus en utilisant une telle factorisation. D’autre part, la factorisation n’est pas recalculée complètement à chaque fois mais simplement “mise à jour” : d’une étape à l’autre seule une colonne diffère dans  $A^J$  et cette propriété est utilisée pour calculer à moindre coût la factorisation de la nouvelle matrice  $A^J$ .

<sup>1</sup>ce rapport étant exactement la valeur à donner à  $x_i$  de manière à annuler la variable “sortante”

<sup>2</sup>la matrice du deuxième système est  $(A^J)^\top$  mais si on connaît une factorisation de  $A^J$  il est aussi facile de résoudre un système linéaire dont la matrice est  $(A^J)^\top$ .

Néanmoins cette technique de mise à jour de factorisation est plus ou moins délicate et nous allons simplifier le problème en calculant une fois l'inverse de  $A^J$  (ou pas du tout si cet inverse est fourni initialement) puis cet inverse sera mis à jour à chaque fin d'étape. Eventuellement la matrice inverse sera parfois recalculée "complètement" si on s'aperçoit que l'on "dérive" de trop (par exemple si  $\|A^J \underline{x}_J - b\|/\|b\|$  est jugé trop grand).

### 3 mise à jour de l'inverse

Supposons que la colonne  $A^i$  rentre dans la base, elle prend la place de la colonne  $A^q$ . Dans  $J$ ,  $i$  prend la place de  $q$ . D'un point de vue informatique, on va utiliser une liste d'indices et pas réellement un ensemble d'où le terme de place. Supposons donc que l'indice  $q$  était en position  $k$  dans la liste  $J$ , si on appelle  $J'$  la nouvelle liste, on a  $J'_l = J_l$  pour  $l = 1, \dots, m, l \neq k$  et  $J'_k = i$ , alors que  $J_k = q$ . De même avec ces notations (qui se traduiront presque directement en langage scilab) on a :

$$A^J = [A^{J_1} | \dots | A^{J_m}]$$

On suppose donc que l'on connaît l'inverse de  $A^J$  (noté  $B^J$ ), peut-on calculer rapidement l'inverse de  $A^{J'}$  ? On a :

$$\begin{aligned} A^{J'} &= A^J - (A^q - A^i)(e^k)^\top \\ A^{J'} &= A^J(I - B^J(A^q - A^i)(e^k)^\top) \end{aligned}$$

d'où :

$$\begin{aligned} B^{J'} &= (I - B^J(A^q - A^i)(e^k)^\top)^{-1} B^J \\ &= (I - w(e^k)^\top)^{-1} B^J, \text{ avec } w = B^J(A^q - A^i) \end{aligned}$$

Si on connaît l'inverse de la matrice  $I - w(e^k)^\top$  on a donc gagné. Cette matrice est une matrice identité sur laquelle on soustrait en colonne  $k$  le vecteur  $w$ . On peut s'apercevoir que cette matrice est effectivement inversible si et seulement si  $1 - w_k \neq 0$  et que son inverse (le vérifier) est :

$$(I - w(e^k)^\top)^{-1} = I + \frac{1}{1 - w_k} w(e^k)^\top$$

On a donc finalement :

$$B^{J'} = B^J + \frac{1}{1 - w_k} w(e^k)^\top B^J = B^J + \frac{1}{1 - w_k} w B_k^J$$

(avec  $B_k^J$  la  $k$  ème ligne de  $B^J$ ). Si vous faites le compte des opérations, vous verrez que ce calcul coûte  $O(m^2)$  opérations, à comparer à  $O(m^3)$  pour un calcul direct de l'inverse.

### 4 considérations pratiques

On va supposer dans la suite que l'on fournit à notre code de simplexe, les listes d'indices  $J$  et  $K$  ainsi que la matrice  $B = (A^J)^{-1}$  en plus des données  $A$ ,  $b$  et  $c$ . Dans la suite, je vous donne quelques lignes de scilab qui vous permettrons de coder de façon synthétique, par exemple, comment calculer les coûts réduits ? C'est très simple puisqu'en scilab (ou Matlab, ou octave, ou Freemath,...) vous pouvez presque copier les expressions matricielles :

```
cr = c(:,K) - (c(:,J)*B)*A(:,K) // calcul des coûts réduits
```

La suite du code peut s'écrire :

```

[crmin, imin] = min(cr)
if crmin >= 0 then, info = "convergence", break, end
i = K(imin) // indice global de la variable qui rentre

```

Quelques précisions :

- la fonction `min` renvoie la composante la plus petite d'un vecteur mais aussi son indice<sup>3</sup> (2<sup>ème</sup> argument renvoyé par la fonction); ici l'indice obtenu est celui du numéro de la composante dans le vecteur `cr`, l'indice global (cad celui dans la matrice initiale) étant obtenu par `i = K(imin)`.
  - ensuite vous voyez le test correspondant à la sortie "normale" du simplexe; la variable `info` qui sera renvoyée par votre fonction `simplexe` nous sert à enregistrer les différents cas de figures; parmi ceux-ci, on pourra utiliser :
    - `info = "fct coût non bornée"`
    - `info = "trop d'iterations"` car en dehors des cas de cyclages (rarement rencontrés dans la pratique), on peut avoir des problèmes de cyclage "numérique" et d'autres problèmes (matrice non inversible, etc....). Il est ainsi bon de prévoir un nombre maximum d'itérations pour éviter d'attendre trop longtemps (on observe en général une moyenne d'environ  $3m$  itérations, un choix raisonnable consiste à prendre  $itermax = 6m$ ).
 mais d'autres pourront se rajouter peut être par la suite.
  - l'instruction `break` permet de sortir d'une boucle `while` ou d'une boucle `for`.
- Voici maintenant l'entête et le squelette que vous donnerez à votre super fonction :

```

function [xo,pt,J,K,info,co,pt,B,iter] = simplexe(c, A, b, J, K, itermax, B, verb)
// J le vecteur des indices de base, B inverse de A(:,J)
// si verb vaut %t on affichera des informations lors du déroulement
// (pratique pour déboguer...) comme le coût, la sol de base, J, etc...
// initialisations
while %t // boucle infinie
// calcul solution de base
// calcul coût réduits
// calcul variable sortante
// mises à jour diverses
end
endfunction

```

Pour les tests, utilisez un P.L. en forme canonique  $Ax \leq b$  avec  $b \geq 0$  puisqu'en rajoutant les variable d'écart, on dispose d'une solution de base réalisable. Ainsi si  $m$  est le nombre d'inéquations et  $n$  le nombre de variables initiales, on se retrouve avec  $n + m$  variables, avec  $K = [1, 2, \dots, n]$  et  $J = [n + 1, \dots, n + m]$  ce qui s'écrit `K = 1:n; J = (n+1):(n+m)` en scilab.

Voilà, à vous de jouer, des informations supplémentaires seront données lors des 2 séances. Une dernière aide : la fonction `find` est très pratique pour éviter d'écrire une boucle, par exemple, `ind = find(x>0)` donne les indices des composantes strictement positives du vecteur  $x$  (on obtient `ind = []` si  $x_i \leq 0, \forall i$ ).

---

<sup>3</sup>en cas de minimum multiple, l'indice donné est celui de la première composante où le min est atteint.