Programmation de la méthode du simplexe

Partie 1 : Introduction et choix du langage de programmation

Programmer l'algorithme du simplexe est à première vue difficile, mais il n'en est rien. Les technologies et les langages ont extrêmement évolués, permettant de créer des algorithmes de plus en plus puissants et de plus en plus facilement.

Il existe une multitude de langages disponibles et utilisables pour programmer l'algorithme du simplexe. On pourrait utiliser des langages puissants comme le C ou le C++, ou des langages objets comme le Java ou l'Objective-C. On pourrait même utiliser la programmation web et associer l'algorithme à une interface graphique évoluée. Cependant, une interface n'est pas notre objectif.

Afin de bien choisir le langage, nous allons nous appuyer sur 3 critères :
- La portabilité : avoir un langage portable nous permettra de faire fonctionner le programme sur n'importe quel ordinateur sans trop d'efforts
- La puissance : le langage doit être puissant, et donc avoir une rapidité d'exécution assez élevée
- Le niveau : le niveau d'un langage est sa proportion à être proche du langage humain. Plus un langage est haut niveau, plus il est proche du langage humain. À l'inverse, plus un langage est bas niveau, plus il est proche du langage machine et donc difficile à mettre en œuvre

On va comparer 3 langages de programmation : C, Java et Python.
Pour le C :
- La portabilité : le langage n'est pas du tout portable, car c'est un langage compilé. Le code source doit être recompilé pour chaque machine à chaque fois.
- La puissance : le langage C est un monstre de puissance, et a été amélioré en ce sens. Il est le langage le plus puissant des trois.
- Le niveau : bien que le C soit considéré comme un langage de haut niveau, il demande une gestion de la mémoire de l'ordinateur qui peut être inutile et chronophage pour ce programme

Pour le Java :
- La portabilité : le langage Java est très portable, c'est un langage hybride (compilé-interprété). Il ne demande qu'un simple logiciel d'exécution (graphique ou en ligne de commande)
- La puissance : la nature du Java le rend moins puissant que le C, mais existant depuis plus de 20 ans, il a été grandement amélioré. Il reste cependant moins rapide, surtout que l'environnement d'exécution Java peut être lourd à traiter pour un ordinateur.

- Le niveau : le Java est un peu plus haut niveau que le C, et il n'y a pas la gestion mémoire derrière le programme. Cependant, le Java est un langage orienté objet strictement, et notre programme ne fait pas vraiment appel à ce paradigme

Pour le Python :
- La portabilité : le Python est un langage exclusivement interprété. Donc comme le Java, il est très portable. Mais il est encore plus portable que le Java. Le Java nécessite un environnement de variables et d'exécution, ainsi que la machine virtuelle Java, là où le Python ne demande qu'un simple interpréteur Python. Donc le Python est le plus portable de nos langages
- La puissance : longtemps considéré comme peu puissant et lent, le Python a été grandement amélioré, notamment à sa version 3, et est devenu un langage de choix pour l'intelligence artificielle. Il reste cependant moins puissant que le C
- Le niveau : le Python est un langage de très haut niveau, il est presque la traduction en anglais de l'algorithmique. Il est donc très facilement utilisable, et compréhensible, même pour un débutant. De plus, c'est un langage à paradigme hybride : procédural à la base comme le C, il peut cependant utiliser le paradigme objet comme Java. Et tout comme le Java, il ne demande aucun traitement mémoire.

Donc parmi ces trois langages, lequel est le meilleur pour l'algorithme du simplexe ? Et bien le Python semble être un langage de choix : très portable, suffisamment puissant, et très haut niveau, donc facile à mettre en œuvre.

Partie 2 : Choix de la bibliothèque et organisation

Le Python permet de faire beaucoup de calculs, cependant, comme le C et le Java, il ne sait pas faire grand-chose sans bibliothèque. Il ne sait pas par exemple faire de calculs matriciel. Il nous faut donc une bibliothèque qui puisse nous aider à programmer la méthode, ou du moins vérifier si ça existe. En ayant fait quelques recherche, la bibliothèque Numpy semble parfaite pour notre cas : elle gère les calculs matriciels, les tableaux multidimentionnels, peut trouver l'indice du nombre le plus grand etc. Il est donc temps de l'installer.
Pour l'installer, le site https://scipy.org/install.html indique la marche à suivre pour installer Numpy. Comme je suis sous Linux Debian, j'utilise la méthode sudo apt-get install python-numpy. Pour de plus amples informations, allez consulter le site (disponible également en annexe).

Nous allons voir l'organisation de la programmation du code, et comment nous allons le programmer.

Nous allons d'abord utiliser l'exemple 1 du cours dans la partie « première espèce », cela nous donnera une référence au résultat, mais on jettera un coup d'œil sur l'exemple 2, il contient une spécificité qu'il n'y a pas dans le 1.

Le programme doit être facilement réutilisable, c'est-à-dire que l'algorithme mis en place dans le programme ne doit pas être spécifique à l'exemple pris en référence. On doit pouvoir changer le tableau de départ sans changer l'algorithme.

Maintenant que les bases sont posées, explorons le programme.

Partie 3 : Le programme

Nous allons nous attarder sur l'algorithme en lui-même, les initialisation de tableaux et tout le reste sont commentées dans le code source, disponible en annexe.

Cependant, petit point sur l'initialisation : la ligne L0 contenant les coefficients de la fonction économique est un tableau unidimensionnel à part, ça facilite les calculs de lignes.

On a donc deux tableaux : la ligne L0, contenant les coefficients de la fonction économique ainsi que la valeur maximale avec le dernier item Z-0 (noté directement 0 dans le programme, il n'aime pas mélanger les chaînes de caractères et les nombres), et le tableau multidimentionnel contenant les lignes L1 à Ln.

On va donc récupérer la taille d'une ligne, qui reste la même pour tout le tableau, et la taille d'une colonne du deuxième tableau, cela nous servira à l'explorer la deuxième étape de l'algorithme.

On créé également une variable qui va permettre de sortir de la boucle lorsque tous les coefficients de L0 sont inférieurs ou égaux à 0 (lorsque l'algorithme se termine).

On entre dans la boucle et la première chose que l'on fait, c'est de vérifier si le coefficient maximal de la ligne L0 est inférieur ou égal à 0. Si le coefficient maximal est inférieur ou égal à 0, ça veut dire qu'aucun nombre ne dépasse un nombre négatif ou nul, et donc l'algorithme s'arrête. On met donc notre variable à 0 pour sortir de la boucle, et on indique que l'algorithme est terminé. S'il y a encore un coefficient positif strictement, alors on peut commencer/continuer l'algorithme.

On va donc chercher le premier critère de Danzig. Sauf que ce qui nous intéresse, ce n'est pas le nombre en lui-même, mais l'indice de ce nombre dans le tableau. Et c'est là où la puissance de la bibliothèque Numpy fait son œuvre : plutôt que de faire une boucle pour chercher à quel indice se situe le plus grand nombre, une simple ligne de code suffit : `danzig1 = np.argmax(tabEq)`

La fonction argmax va renvoyer l'indice du premier plus grand nombre d'un tableau. Il va de gauche à droite, ce qui n'est pas très grave. En quelques sortes : si le plus grand nombre (par exemple 3) se retrouve en 2 exemplaires à l'indice 0 et 1, l'indice renvoyé est l'indice 0.

On a donc désormais le premier critère de Danzig, qui va nous donner la colonne sur laquelle on va travailler.

On va donc pouvoir se charger du 2ème critère de Danzig. Là aussi on veut le rang, et non la valeur en elle même. Cependant ce rang n'est pas associé à une valeur maximale ou minimale, mais à la ligne où la division entre le nombre de la dernière colonne et le nombre de la colonne où le 1er critère de Danzig s'applique est la plus petite. C'est là que les problèmes ont commencé à arriver.

Comme on commence à la ligne d'index 0, et que la première ligne n'est pas exclue du 2ème critère de Danzig, on initialise la variable danzig2 à 0. On commence ensuite une boucle qui va explorer lignes par lignes le tableau des contraintes grâce à la taille récupérée en début de programme.

Le premier test que l'on va effectuer n'est pas tout de suite la division, mais on va vérifier si la valeur qui se trouve à la ligne courante et à la colonne du critère de Danzig 1 est strictement supérieure à 0. En effet, on ne vérifie que les divisions où le dénominateur est positif, car s'il est nul, cela donnera une division par 0. On vérifie si on est sur la première ligne. Si oui, on initialise une variable div qui va contenir le premier résultat de la division que l'on va comparer. Sinon, ça veut dire qu'on est au moins à la deuxième ligne, et la comparaison commence. Voyons comment ça se comporte :

On met la division à la ligne actuelle (celle qui suit forcément la première), et on va comparer si cette division à un résultat plus petit que le précédent (si test <div). Si oui, la valeur de la division précédente est remplacée par celle testée, afin de pouvoir préparer le test suivant, et la valeur de danzig2 est mise à la valeur de la ligne testée. Sinon, on passe juste à la ligne suivante (on ne fait rien).

Ce programme va boucler, et à la fin, on aura un rang danzig2 qui va nous indiquer la ligne. Et grâce à la combinaison danzig2 (ligne) et danzig1 (colonne), on peut trouver la valeur exacte.

Une fois cette boucle terminée, on a donc tout ce qu'il nous faut pour les soustractions de lignes.On va d'abord faire la soustraction de la première ligne L0, car elle dépend du coefficient qu'on peut trouver à la ligne danzig2 colonne danzig1. On calcule le coefficient (peut être égal à 1) et ensuite, on remplace la ligne L0 par L0 – coef*LD2, LD2 étant la ligne à l'index trouvé danzig2 (exemple : L0 – 3*L3 si L3 est la ligne où le critère de Danzig 2 s'applique).

Une fois ce premier remplacement fait, on peut donc s'occuper des autres lignes (L1 à Ln) :

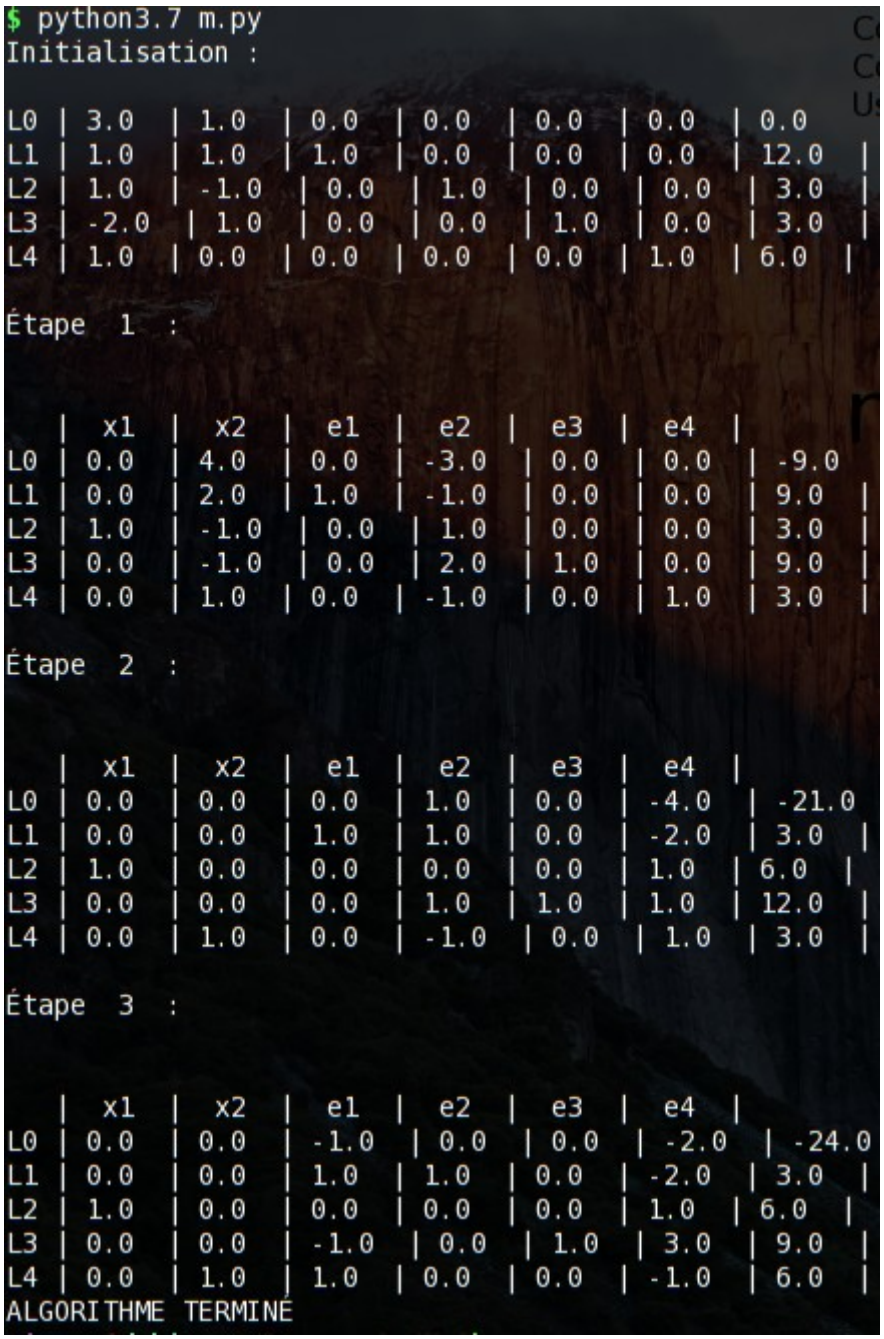
On explore le tableau ligne par ligne, et à chaque tour de boucle, on fait un test très simple : on vérifie s'il on se situe à la ligne LD2, car le comportement du changement de ligne ne sera pas le même. Si on n'y est pas, ça veut dire que l'on doit calculer le coefficient d'une manière similaire à la ligne L0, et on soustrait la ligne de la même manière que pour L0. Cependant, s'il on est à la ligne LD2, le coefficient sera le nombre à la ligne LD2 colonne COLD1 (colonne où le critère de Danzig n°1 s'applique). Et la ligne est remplacée par elle-même divisée par le coefficient. (LD2 = LD2/coef). Dans l'exemple 1 du cours, ce coefficient est toujours égal à 1, donc la ligne ne change pas, cependant dans l'exemple 2 qui le suit, il y a des étapes où la ligne LD2 est divisée par 3 par exemple. C'est pour ça qu'il fallait jeter un œil à l'exemple 2 du cours.

Une fois toutes ces lignes sont soustraites et calculées, on les affiche, et ainsi de suite jusqu'à l'arrêt de l'algorithme. Cela nous permet de voir les étapes de calculs.

Partie 4 : Exécution du code sur deux exemples

Pour montrer que le code fonctionne bien, il faut donc lui implémenter des exemples. L'exemple 1 du cours permettra de montrer son fonctionnement, mais comme lors du développement, l'exemple 2 a été testé, il fonctionne au moins sur les deux premiers exemples. D'ailleurs, cela m'a permis de remarquer deux petites erreurs dans le cours à l'exemple 2, mais rien de bien gros.

Exemple 1 :



À la ligne L0, la dernière valeur correspond bien à la valeur Z – 0 dans le cours.

Comparaisons :

Initialisation : c’est forcément la même que dans le cours, puisqu’elle est entrée manuellement dans le code.

Étape 1 :

x_1	x_2	e_1	e_2	e_3	e_4		
0	4	0	-3	0	0	Z-9	L_0
0	2	1	-1	0	0	9	L_1
1	-1	0	1	0	0	3	L_2
0	-1	0	2	1	0	9	L_3
0	1	0	-1	0	1	3	L_4

Étape 2 :

x_1	x_2	e_1	e_2	e_3	e_4		
0	0	0	1	0	-4	Z-21	L_0
0	0	1	1	0	-2	3	L_1
1	0	0	0	0	1	6	L_2
0	0	0	1	1	1	12	L_3
0	1	0	-1	0	1	3	L_4

Étape 3 (étape finale) :

x_1	x_2	e_1	e_2	e_3	e_4	
0	0	-1	0	0	-2	Z-24
0	0	1	1	0	-2	3
1	0	0	0	0	1	6
0	0	-1	0	1	3	9
0	1	1	0	0	-1	6

L’affichage n’est pas très aligné à chaque fois mais on peut bien suivre l’évolution de chaque étape.

Partie 5 : Problèmes rencontrés et améliorations possibles

Un problème de Typage...

La programmation de la méthode du simplexe ne s'est pas faite aussi facilement que prévue. Bien qu'il n'y ait eu que peu de soucis, le plus gros problème n'est pas dû à l'algorithme en lui-même, mais au langage de programmation.

Lors de tests sur l'exemple 2 du cours, j'ai été confronté à un comportement étrange : les premières étapes étaient correctes, mais dès que les étapes nécessitaient un coefficient réel, à virgule infinie (ici, il s'agissait de 1/3, donnant 0,33333….), les valeurs n'étaient plus correctes. J'ai donc dû analysées les valeurs des coefficients en les affichant. Après quelques tests, je me suis rendu compte du problème, pourtant très bête : le type de variable.
Python est un langage dit à typage dynamique, c'est-à-dire qu'il type les variables à la volée. Cependant, le tableau des contraintes était déclaré à la base en entier. Et Python ne peut pas retyper les variables d'un tableau multidimensionnel. Et donc ils étaient restés en entier, et donc pour des valeurs comme des tiers, il arrondi. Il fallait donc déclarer les éléments des tableaux en float. Pour ce faire, on ajoute juste ce qui suit la virgule (par exemple, remplacer le coefficient 3 par 3.0). Et une fois fait, plus de problème de typage.

Limitations et améliorations

Le programme ne traite que des problèmes de première espèce. Il faudrait donc l'améliorer pour qu'il intègre les possibilités de cas de deuxième espèce, tout à fait possible, mais difficilement réalisable avec la configuration actuelle du programme.
Le programme ne traite pas des cas particuliers, comme le cyclage et la dégénérescence. S'il y a un cyclage, le programme va juste continuer indéfiniment. C'est donc quelque chose à prendre en compte.
Le programme n'a pas d'interface en ligne de commande. Pour changer la fonction économique et les contraintes, il faut changer les tableaux correspondants dans le code source. Bien que ce ne soit pas compliqué à faire, il serait préférable qu'on puisse entrer la fonction économique et les contraintes directement depuis le terminal, et laisser le programme faire le travail. Des tentatives ont été menées, mais sans succès, la difficulté étant de trouver une bonne interface et de créer une fonction de remplissage de tableau à taille variable, notamment multidimensionnel pour les contraintes.

Le programme ne rempli donc pas l'entièreté du cahier des charges du sujet, cependant, l'algorithme du simplexe a été mis en place, ce qui est la plus grosse base.

Annexe

Sources

Site officiel du langage Python : https://www.python.org/

Télécharger et installer Python : https://www.python.org/downloads/

Installer la bibliothèque Numpy : https://www.scipy.org/install.html

Cours sur la bibliothèque Numpy (en Français) : https://courspython.com/apprendre-numpy.html

Code source (non commenté, voir fichier .py):

```
import numpy as np
tabEq = np.array([3.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0])
tabCons = np.array([[1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 12.0],
                    [1.0, -1.0, 0.0, 1.0, 0.0, 0.0, 3.0],
                    [-2.0, 1.0, 0.0, 0.0, 1.0, 0.0, 3.0],
                    [1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 6.0]])
print("Initialisation : \n")
print("L0 |", tabEq[0], " |", tabEq[1], " |", tabEq[2], " |", tabEq[3], " |", tabEq[4], " |", tabEq[5], " |", tabEq[6])
print("L1 |", tabCons[0, 0], " |", tabCons[0, 1], " |", tabCons[0, 2], " |", tabCons[0, 3], " |", tabCons[0, 4], " |", tabCons[0, 5], " |", tabCons[0, 6], " |")
print("L2 |", tabCons[1, 0], " |", tabCons[1, 1], " |", tabCons[1, 2], " |", tabCons[1, 3], " |", tabCons[1, 4], " |", tabCons[1, 5], " |", tabCons[1, 6], " |")
print("L3 |", tabCons[2, 0], " |", tabCons[2, 1], " |", tabCons[2, 2], " |", tabCons[2, 3], " |", tabCons[2, 4], " |", tabCons[2, 5], " |", tabCons[2, 6], " |")
print("L4 |", tabCons[3, 0], " |", tabCons[3, 1], " |", tabCons[3, 2], " |", tabCons[3, 3], " |", tabCons[3, 4], " |", tabCons[3, 5], " |", tabCons[3, 6], " |")
tailleCol = len(tabCons[:, 0])
tailleLin = len(tabEq)
run = 1
noPhase = 0
while run == 1:
  if max(tabEq) <= 0:
    run = 0
    print("ALGORITHME TERMINÉ")
  else:
    noPhase = noPhase + 1
    print("\nÉtape ", noPhase, " :")
    danzig1 = np.argmax(tabEq)
    danzig2 = 0
    for i in range(tailleCol):
      if tabCons[i, danzig1] > 0:
        if i == 0:
          div = tabCons[i, len(tabCons[0, :]) -1] / tabCons[i, danzig1]
          danzig2 = i
          else:
            test = tabCons[i, len(tabCons[0, :]) -1] / tabCons[i, danzig1]
            if test < div:
              danzig2 = i
              div = test
              coef = tabEq[danzig1]/tabCons[danzig2, danzig1]
              tabEq = tabEq - coef*tabCons[danzig2, :]
              for i in range(tailleCol):
                if i != danzig2:
                  coef = tabCons[i, danzig1] / tabCons[danzig2, danzig1]
                  tabCons[i] = tabCons[i] - coef * tabCons[danzig2]
                  elif i == danzig2:
                    coef = tabCons[danzig2, danzig1]
                    tabCons[i] = tabCons[i] / coef
              print("\n")
              print("   | x1 | x2 | e1 | e2 | e3 | e4 |")
              print("L0 |", tabEq[0], " |", tabEq[1], " |", tabEq[2], " |", tabEq[3], " |", tabEq[4], " |", tabEq[5], " |", tabEq[6])
              print("L1 |", tabCons[0, 0], " |", tabCons[0, 1], " |", tabCons[0, 2], " |", tabCons[0, 3], " |", tabCons[0, 4], " |", tabCons[0, 5], " |", tabCons[0, 6], " |")
              print("L2 |", tabCons[1, 0], " |", tabCons[1, 1], " |", tabCons[1, 2], " |", tabCons[1, 3], " |", tabCons[1, 4], " |", tabCons[1, 5], " |", tabCons[1, 6], " |")
              print("L3 |", tabCons[2, 0], " |", tabCons[2, 1], " |", tabCons[2, 2], " |", tabCons[2, 3], " |", tabCons[2, 4], " |", tabCons[2, 5], " |", tabCons[2, 6], " |")
              print("L4 |", tabCons[3, 0], " |", tabCons[3, 1], " |", tabCons[3, 2], " |", tabCons[3, 3], " |", tabCons[3, 4], " |", tabCons[3, 5], " |", tabCons[3, 6], " |")
```