L'algorithme du Simplexe expliqué simplement

Introduction

En cours de **programmation linéaire**, nous avons étudié un algorithme fascinant : le **Simplexe**. Nous avons appris à le **résoudre à la main**, à **modéliser des problèmes** avec cet algorithme, puis les resoudre avec Julia.

Dans cet article, je vais partager ce que j'ai appris de façon simple et accessible, pour tous ceux qui découvrent l'optimisation.

1. C'est quoi l'algorithme du Simplexe ?

L'algorithme du Simplexe est une méthode pour **résoudre un problème de programmation linéaire**, c'est-à-dire **maximiser (ou minimiser)** une fonction sous des contraintes linéaires.

Il explore les **sommets (ou coins)** du polygone formé par les contraintes, à la recherche de la meilleure solution.

2. Étude de cas : résolution d'un problème de programmation linéaire

Pour bien comprendre comment fonctionne l'algorithme du Simplexe, commençons par un **exemple concret** que nous avons vu en cours avec Mme FIGUEIREDO Rosa (rosa.figueiredo@univ-avignon.fr).

Voici le **problème de programmation linéaire (PL)** que nous allons résoudre :

Maximiser

```
z = x_c + x_o
```

Sous contraintes :

- `4x_c + 2x_o ≤ 160`
- `3x_c + 6x_o ≤ 150`
- `x_c + x_o ≤ 20 x_c`
- `≥ 0, x_o ≥ 0`

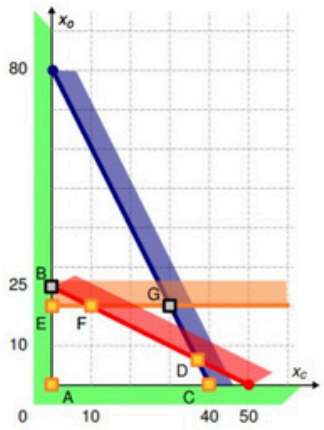
Ce problème est typique d'un **problème de programmation linéaire en deux variables**. Il est parfait pour être résolu à la main ou graphiquement.
Avant d'appliquer l'algorithme du Simplexe, il faut **réécrire notre problème sous forme standard**. Cela signifie :

- Toutes les contraintes sont transformées en **égalités** (en ajoutant des variables dites de **slack** ou **artificielles**).
- Toutes les variables sont **positives ou nulles** (≥ 0).
- L'objectif est de **maximiser une fonction linéaire**.

Source : cours de Mme Figueiredo Roza (module Programmation linéaire, 2024).

Solution associée à la base $\{s_1, s_2, s_3\}$

	x_c	x_o	s_1	s_2	s_3	z	RHS
(C_1)	4	2	1				160
(C_2)	3	6		1			150
(C_3)		1			1		20
(Obj)	−1	−1				1	



On introduit trois variables supplémentaires :

- s_1, s_2, s_3 : ce sont les **variables de slack**, qui transforment les inégalités en égalités.

Notre problème devient :

Maximiser `z = x_c + x_o`

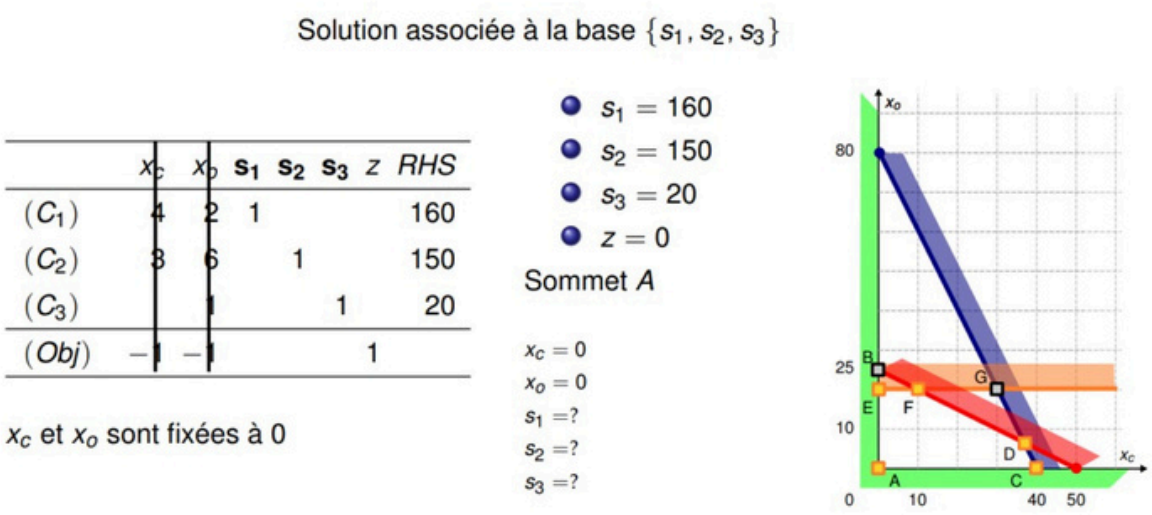
Sous contraintes :

- `4x_c+2x_o + s₁ = 160`
- `3x_c+6x_o + s₂ = 150`
- `x_c+x_o + s₃ = 20`

- $x\_c,\ x\_o,\ s_1,\ s_2,\ s_3 \geq 0$

Puis on fait la construction du premier tableau du Simplexe, c'est le tableau en haut à gauche.
Dans notre tableau initial, nous utilisons comme **base de départ** les variables **{s_1, s_2, s_3}**. Ces variables sont dites **artificielles**, car elles n'apparaissent pas dans la fonction objectif, mais elles permettent de démarrer le processus.

À ce stade, x_c et x_o sont fixées à 0. On lit alors la valeur des variables de base directement dans la colonne RHS (second membre). C'est ce qu'on appelle **la solution associée à la base initiale**.



Solution associée à la base {s_1, s_2, s_3} – tiré du cours de Mme Figueiredo Roza

- On lit directement les valeurs des variables **de base**s_1, s_2, s_3 dans la colonne **RHS** du tableau :
  - $s_1 = 160$
  - $s_2 = 150$
  - $s_3 = 20$
- La valeur de la fonction objectif est $z = 0$.

Cela correspond au **sommet A** dans la représentation graphique à droite, où toutes les variables de décision (x_c, x_o) sont nulles. C'est donc notre **point de départ** dans l'espace des solutions.

Après avoir appris à **lire la solution associée à une base**, nous allons maintenant explorer ce qu'on appelle la **forme canonique** et comprendre **pourquoi elle est essentielle** dans l'algorithme du Simplexe.

> Un problème de programmation linéaire est sous **forme canonique** lorsqu'il est écrit de manière à ce que les **variables de base**, ainsi que la fonction objectif, forment une **matrice identité** dans le tableau.

Cela signifie que dans le tableau :

- Chaque variable de base a **un coefficient égal à 1** sur **sa ligne**,
- Et **0 dans toutes les autres lignes**.

Mettre le tableau sous forme canonique pour la base $\{x_o, s_2, s_3\}$

	x_c	x_o	s_1	s_2	s_3	z	RHS
(C_1)	4	2	1				160
(C_2)	3	6		1			150
(C_3)		1			1		20
(Obj)	-1	-1				1	

	x_c	x_o	s_1	s_2	s_3	z	RHS
$(C_1)/2$	2	1	$1/2$	0	0	0	80
$(C_2)-3(C_1)$	-9	0	-3	1	0	0	-330
$(C_3)-\frac{1}{2}(C_1)$	-2	0	$-\frac{1}{2}$	0	1	0	-60
$(Obj)+\frac{1}{2}(C_1)$	1	0	$\frac{1}{2}$	0	0	1	80

Exemple de forme canonique colorée – tiré du cours de Mme Figueiredo Roza Mettre un

tableau sous forme canonique permet de **lire immédiatement la solution associée à une base** : les valeurs des variables de base se trouvent directement dans la colonne RHS (second membre).
Cela simplifie les calculs et structure l'algorithme pour effectuer des **pivots** correctement.
Parfait, on va maintenant passer à l'**exemple de pivotage**, qui est le cœur de l'algorithme du Simplexe. Je vais expliquer **clairement** ce qu'on fait, pourquoi on le fait, et comment le lire à partir des **transformations de lignes** dans le tableau.

Le **pivotage** permet de **passer d'une base à une base voisine** en améliorant la valeur de la fonction objectif z.
C'est un processus itératif qui consiste à :

- **faire entrer** une variable dans la base,
- et **faire sortir** une autre.

Passage à la forme canonique pour la base {x_o, s_2, s_3}

Mettre le tableau sous forme canonique pour la base $\{x_o, s_2, s_3\}$

	x_c	x_o	s_1	s_2	s_3	z	RHS
(C_1)	4	2	1				160
(C_2)	3	6		1			150
(C_3)		1			1		20
(Obj)	−1	−1				1	

	x_c	x_o	s_1	s_2	s_3	z	RHS
$(C_1)/2$	2	1	1/2	0	0	0	80
$(C_2)-3(C_1)$	−9	0	−3	1	0	0	−330
$(C_3)-\frac{1}{2}(C_1)$	−2	0	$-\frac{1}{2}$	0	1	0	−60
$(Obj)+\frac{1}{2}(C_1)$	1	0	$\frac{1}{2}$	0	0	1	80

Mise en forme canonique par pivot – extrait du cours de Mme Figueiredo Roza

Objectif : faire entrer x_o dans la base

Étape 1 : identifier la colonne pivot

- x_o a un **coût réduitnégatif** (−1), on peut donc l'ajouter à la base.

Étape 2 : ratio test On cherche dans quelle ligne elle va entrer en divisant RHS / valeur de la colonne x_o :

- C_1 : 160 / 2 = 80
- C_2 : 150 / 6 = 25
- C_3 : 20 / 1 = 20 **(plus petit ratio)**

Le pivot se fera sur la cellule (C_3, x_o) = 1

Étape 3 : Réduction par pivot

On applique maintenant les **opérations élémentaires** pour transformer la colonne de x_o en un **vecteur unitaire** (1 en C_3, 0 ailleurs).

Résultat : x_o entre dans la base à la place de s_3

Le tableau final sera sous forme canonique pour la nouvelle base {x_o, s_2, s_1} (ou selon le pivot effectué).
Grâce à cette opération, on **progresse** vers une solution où la valeur de z est plus élevée.

Une fois le **premier pivot effectué**, on obtient un nouveau tableau avec une **base mise à jour**.
Ensuite, on continue le processus selon les mêmes étapes :

1. **Identifier** une colonne avec un **coût réduit négatif** dans la ligne objectif (ligne (Obj)).
2. **Faire un ratio test** pour choisir la variable qui **sortira de la base**.
3. **Effectuer un pivot** pour remplacer cette variable par la nouvelle.

On répète ces opérations **jusqu'à ce que tous les coûts réduits soient positifs ou nuls**, ce qui signifie qu'on a atteint une **solution optimale**.
À chaque étape, on progresse vers un sommet adjacent du polygone des solutions, et la valeur de z augmente ou reste stable.

3. Mise en pratique avec Julia

Dans le TP, nous avons utilisé **Julia** avec la librairie **JuMP.jl** pour modéliser et résoudre des problèmes de programmation linéaire.
Ce langage, conçu pour les performances scientifiques, offre une **syntaxe simple et intuitive**, ce qui rend la modélisation très accessible.

Comme vous pouvez le voir ci-dessous, la définition des **variables**, de la **fonction objectif**, et des **contraintes** se fait en quelques lignes seulement :
julia

```
using JuMP, GLPK

model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
@variable(model, y >= 0)
@objective(model, Max, 2x + 3y)
@constraint(model, x + y <= 4)
@constraint(model, 2x + 5y <= 12)
@constraint(model, x + 2y <= 5)
```

En quelques secondes, Julia résout le problème en appelant un solveur comme **GLPK**, qui applique en arrière-plan l'algorithme du **Simplexe**.

Ce type d’outil nous permet de **généraliser facilement** des modèles plus complexes et d’obtenir des résultats très rapidement.

4. Conclusion:

Le Simplexe repose sur une **logique géométrique** : on explore les **sommets** du polygone de contraintes, pas toutes les combinaisons possibles.

Il existe des cas particuliers à prendre en compte : **dégénérescence**, **problèmes non bornés**, ou **cycliques**.
Cet algorithme est toujours utilisé dans des domaines concrets comme :

- la **logistique**,
- la **finance**,
- l’**intelligence artificielle**,
- et même dans des outils comme Julia qui l’utilisent en arrière-plan pour résoudre automatiquement des modèles linéaires.