Recherche Opérationnelle

Notes de Cours

Master M1 — 2025–2026

Yaë Ulrich Gaba

« Optimiser c'est simplifier sans rien perdre d'essentiel. »
— George Dantzig

25 mars 2026

Table des matières

Préface

La **recherche opérationnelle** (RO) est une discipline scientifique qui s'appuie sur des méthodes mathématiques, statistiques et algorithmiques pour aider à la prise de décision. Née pendant la Seconde Guerre mondiale au sein des états-majors alliés, elle s'est depuis imposée comme un outil incontournable dans l'industrie, la logistique, la finance, les télécommunications et bien d'autres domaines.

Objectifs du cours

Ce cours s'adresse aux étudiants de deuxième et troisième année de licence (L2–L3) en mathématiques, informatique ou sciences de l'ingénieur. À l'issue de cet enseignement, l'étudiant sera capable de :

- Modéliser des problèmes concrets sous forme de programmes mathématiques (linéaires, en nombres entiers, non linéaires).
- Résoudre ces programmes à l'aide des algorithmes fondamentaux (simplexe, branch-and-bound, plus court chemin, flot maximal, etc.).
- Interpréter les résultats, notamment via l'analyse de dualité et de sensibilité.
- Utiliser des outils logiciels (Python, PuLP, SciPy, NetworkX) pour résoudre des problèmes de taille réaliste.
- Comprendre les principes de la simulation à événements discrets et de la théorie des files d'attente.

Organisation de l'ouvrage

L'ouvrage est organisé en onze chapitres, conçus pour être étudiés de manière séquentielle :

Chapitre 1 – Introduction à la RO. Historique, définitions et méthodologie générale de la modélisation.

Chapitre 2 – Programmation linéaire : formulation et géométrie. Construction de modèles linéaires, interprétation géométrique, régions admissibles et sommets optimaux.

Chapitre 3 – Méthode du simplexe. L'algorithme du simplexe sous forme tabulaire, initialisation par la méthode du grand M et la méthode à deux phases.

Chapitre 4 – Dualité en programmation linéaire. Construction du dual, théorèmes de dualité, interprétation économique.

Chapitre 5 – Analyse de sensibilité. Variation des coefficients, analyse paramétrique, interprétation des prix fictifs (*shadow prices*).

Chapitre 6 – Problèmes de transport et d'affectation. Modèles de transport équilibrés et non équilibrés, méthode du stepping-stone, problème d'affectation et algorithme hongrois.

Chapitre 7 – Théorie des graphes appliquée. Plus courts chemins (Dijkstra, Bellman-Ford), arbres couvrants minimaux (Kruskal, Prim).

Chapitre 8 – Flots dans les réseaux. Flot maximal (Ford-Fulkerson), coupe minimale, flot à coût minimal.

Chapitre 9 – Programmation en nombres entiers. Méthode de branch-and-bound, plans de coupe, relaxation linéaire.

Chapitre 10 – Programmation non linéaire. Conditions de Karush-Kuhn-Tucker, programmation convexe, méthodes de gradient.

Chapitre 11 – Simulation et théorie des files d'attente. Simulation à événements discrets, modèles M/M/1, M/M/c, loi de Little.

Prérequis

Les prérequis nécessaires sont :

- **Algèbre linéaire** : matrices, systèmes d'équations linéaires, bases et rang.
- **Analyse** : fonctions de plusieurs variables, dérivées partielles, convexité.
- **Probabilités** : lois classiques, espérance, processus de Poisson (pour le chapitre 11).
- **Algorithmique** : notions de base en complexité et en programmation Python.

Conventions et notations

Tout au long de cet ouvrage, nous utiliserons les conventions suivantes :

Notation	Signification
$\mathbb{R}$	Ensemble des réels
$\mathbb{R}^n$	Espace euclidien de dimension n
$\mathbb{Z}$	Ensemble des entiers relatifs
$\mathbb{N}$	Ensemble des entiers naturels
$\mathbf{x}$	Vecteur colonne de $\mathbb{R}^n$
$A \in \mathbb{R}^{m \times n}$	Matrice réelle $m \times n$
$\mathbf{c}^\top \mathbf{x}$	Produit scalaire
$\|\mathbf{x}\|$	Norme euclidienne
$\|S\|$	Cardinal de l'ensemble S
$\arg\min$, $\arg\max$	Argument du minimum / maximum

Les théorèmes, définitions et propositions sont numérotés par chapitre. Les exemples sont signalés par un encadré et les exercices figurent en fin de chapitre.

Remarque 0.1. Les encadrés **Remarque** fournissent des compléments ou des mises en garde. Les encadrés **Bonne pratique** offrent des conseils méthodologiques pour la modélisation et l'implémentation.

Logiciels utilisés

Les exemples numériques sont traités en **Python 3** avec les bibliothèques suivantes :

- `scipy.optimize.linprog` – résolution de programmes linéaires ;
- `PuLP` – modélisation et résolution de PL et PLNE ;
- `NetworkX` – graphes, plus courts chemins, flots ;
- `SimPy` – simulation à événements discrets ;
- `NumPy`, `Matplotlib` – calcul numérique et visualisation.

Installation des dépendances

```
# pip install numpy scipy matplotlib pulp networkx simpy
import numpy as np
from scipy.optimize import linprog
import pulp
import networkx as nx
```

Comment utiliser ce cours

Chaque chapitre suit une structure récurrente :

1. **Motivation** – un problème concret qui introduit le besoin théorique.
2. **Théorie** – définitions, théorèmes et démonstrations.
3. **Algorithmes** – description formelle et illustration sur un exemple.
4. **Implémentation** – code Python commenté.
5. **Exercices** – problèmes de difficulté croissante ($\star$ à $\star\star\star$).

Apprentissage actif

La recherche opérationnelle s'apprend en *pratiquant*. Il est fortement recommandé de :

- Refaire les exemples à la main avant de consulter la solution.
- Implémenter les algorithmes en Python.
- Chercher les exercices avant de regarder les corrigés.
- Explorer des variantes des problèmes proposés.

Bref historique

- **1937** : Kantorovich formule les premiers problèmes de programmation linéaire pour la planification économique.
- **1939–1945** : Les équipes de recherche opérationnelle britanniques et américaines optimisent les opérations militaires (convois, radars, bombardements).
- **1947** : George Dantzig publie la méthode du simplexe.
- **1951** : Kuhn et Tucker énoncent les conditions d'optimalité pour la programmation non linéaire.
- **1956** : Ford et Fulkerson proposent l'algorithme de flot maximal.
- **1958** : Gomory introduit les coupes entières.
- **1960** : Land et Doig formalisent le branch-and-bound.
- **1979** : Khachiyan montre que la PL est résoluble en temps polynomial (méthode de l'ellipsoïde).
- **1984** : Karmarkar publie la méthode des points intérieurs.
- **Années 1990–2020** : Développement des solveurs commerciaux (CPLEX, Gurobi) et open source (GLPK, CBC), généralisation dans l'industrie.

Références principales

1. Hillier F.S., Lieberman G.J., *Introduction to Operations Research*, McGraw-Hill, 11e éd., 2021.

2. Bertsimas D., Tsitsiklis J.N., *Introduction to Linear Optimization*, Athena Scientific, 1997.

3. Wolsey L.A., *Integer Programming*, Wiley, 2e éd., 2020.

4. Chvátal V., *Linear Programming*, W.H. Freeman, 1983.

5. Minoux M., *Programmation mathématique : théorie et algorithmes*, Lavoisier, 2e éd., 2008.

6. Sakarovitch M., *Optimisation combinatoire : graphes et programmation linéaire*, Hermann, 1984.

Remerciements

Je remercie les collègues et étudiants qui ont contribué à l'amélioration de ce cours par leurs remarques et suggestions. Toute erreur restante est de ma seule responsabilité.

Bonne lecture et bonne optimisation !

Chapitre 1

Introduction à la Recherche Opérationnelle

Pendant la Seconde Guerre mondiale, la Royal Air Force britannique fait face à un problème logistique désespérant : comment déployer ses radars côtiers pour détecter les bombardiers ennemis avec un nombre limité de stations ? Un groupe de scientifiques — physiciens, mathématiciens, biologistes — est réuni pour analyser le problème *opérationnellement*, c'est-à-dire avec des données, des modèles et des méthodes quantitatives. Ils appellent leur discipline « operational research », et les résultats sont spectaculaires : les pertes diminuent, l'efficacité monte. Après la guerre, ces méthodes migrent vers l'industrie, les transports, la santé et la finance, devenant ce que nous appelons aujourd'hui la *recherche opérationnelle*.

Ce chapitre trace les contours de cette discipline : ses origines, sa démarche, et les grandes classes de problèmes qu'elle cherche à résoudre.

1.1 Qu'est-ce que la recherche opérationnelle ?

Commençons par définir précisément l'objet de notre étude.

Définition 1.1 (Recherche opérationnelle)**.** La **recherche opérationnelle** (RO) est l'application de méthodes scientifiques — mathématiques, statistiques et informatiques — à la prise de décision dans les systèmes complexes. Son objectif est de déterminer la meilleure allocation de ressources limitées pour atteindre un but donné.

La RO se distingue des autres branches des mathématiques appliquées par son ancrage dans la *modélisation* de problèmes réels et par le souci constant de fournir des solutions *implémentables*.

Remarque 1.2. Le terme « recherche opérationnelle »vient de l'anglais *Operations Research*, où *operations* désigne les opérations militaires au cours desquelles la discipline est née. En français, on utilise parfois l'abréviation **RO**.

1.2 Bref historique

1.2.1 Les origines militaires (1937–1945)

Les prémisses de la RO remontent aux travaux de **Leonid Kantorovich** (1937) sur l'optimisation de la production dans l'industrie soviétique. Cependant, la discipline prend

véritablement son essor pendant la Seconde Guerre mondiale :

- En **1940**, le physicien Patrick Blackett dirige le *Blackett's Circus*, un groupe interdisciplinaire chargé d'optimiser le déploiement des radars et la taille des convois maritimes.
- Aux États-Unis, des équipes similaires travaillent sur la stratégie de bombardement et la logistique du Pacifique.

1.2.2 L'ère fondatrice (1947–1960)

- **1947** : George **Dantzig** invente la **méthode du simplexe** pour résoudre les programmes linéaires.
- **1951** : Harold **Kuhn** et Albert **Tucker** formalisent les conditions d'optimalité en programmation non linéaire (conditions KKT).
- **1956** : Lester **Ford** et Delbert **Fulkerson** publient l'algorithme du flot maximal.
- **1958** : Ralph **Gomory** introduit la méthode des coupes entières.
- **1960** : Ailsa **Land** et Alison **Doig** formalisent le **branch-and-bound**.

1.2.3 Développements modernes

- **1979** : Leonid **Khachiyan** démontre que la programmation linéaire est résoluble en temps polynomial (méthode de l'ellipsoïde).
- **1984** : Narendra **Karmarkar** publie la méthode des points intérieurs, compétitive avec le simplexe en pratique.
- **Années 2000–2020** : Les solveurs modernes (CPLEX, Gurobi, CBC) résolvent des problèmes à des millions de variables.

1.3 Domaines d'application

La RO intervient dans de très nombreux secteurs :

Secteur	**Exemples de problèmes**
Logistique	Planification de tournées, gestion de stocks
Production	Ordonnancement, dimensionnement de lots
Finance	Optimisation de portefeuille, gestion du risque
Transport	Conception de réseaux, horaires de bus/trains
Télécommunications	Routage, affectation de fréquences
Santé	Planification des blocs opératoires, gestion de lits
Énergie	Unit commitment, gestion de réseaux électriques
Aéronautique	Revenue management, affectation de portes

1.4 Méthodologie de la RO

La démarche générale de la RO suit un cycle en six étapes :

1. **Définition du problème** : identifier l'objectif, les décideurs, les contraintes opérationnelles.
2. **Collecte des données** : paramètres du modèle, coûts, capacités, demandes.
3. **Formulation du modèle** : traduire le problème en termes mathématiques (variables, objectif, contraintes).
4. **Résolution** : appliquer un algorithme ou un solveur.
5. **Validation** : vérifier la cohérence du modèle et la pertinence de la solution.
6. **Implémentation** : mettre en œuvre la solution et suivre ses performances.

Modéliser, c'est simplifier

Un modèle n'est jamais une représentation exacte de la réalité. L'art de la modélisation consiste à trouver le bon équilibre entre *précision* (modèle fidèle) et *tractabilité* (modèle résoluble).

1.5 Classification des modèles d'optimisation

Définition 1.3 (Problème d'optimisation)**.** Un **problème d'optimisation** se formule de manière générique comme :

$$\min_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}) \quad \text{sous les contraintes} \quad g_i(\mathbf{x}) \leq 0,\ i = 1, \ldots, m, \qquad h_j(\mathbf{x}) = 0,\ j = 1, \ldots, p,$$

où f est la **fonction objectif**, $\mathcal{X}$ l'ensemble de définition, les g_i les **contraintes d'inégalité** et les h_j les **contraintes d'égalité**.

On distingue les modèles selon plusieurs critères :

1.5.1 Selon la nature de l'objectif et des contraintes

- **Programmation linéaire (PL)** : f, g_i, h_j sont des fonctions linéaires de $\mathbf{x}$.
- **Programmation non linéaire (PNL)** : au moins l'une des fonctions est non linéaire.
- **Programmation quadratique** : objectif quadratique, contraintes linéaires.

1.5.2 Selon la nature des variables

- **Variables continues** : $\mathbf{x} \in \mathbb{R}^n$.
- **Variables entières** : $\mathbf{x} \in \mathbb{Z}^n$ (programmation en nombres entiers, PLNE).
- **Variables mixtes** : certaines continues, d'autres entières (programmation mixte, MIP).
- **Variables binaires** : $x_i \in \{0, 1\}$ (programmation 0–1).

1.5.3 Selon l'information disponible

- **Déterministe** : tous les paramètres sont connus avec certitude.
- **Stochastique** : certains paramètres sont des variables aléatoires.
- **Robuste** : on cherche une solution performante pour tous les scénarios possibles.

1.6 Premiers exemples de modélisation

Exemple 1.4 (Problème de production)**.** Une usine fabrique deux produits P_1 et P_2. Chaque unité de P_1 rapporte 5 EUR et nécessite 2 heures de machine A et 1 heure de machine B. Chaque unité de P_2 rapporte 4 EUR et nécessite 1 heure de machine A et 3 heures de machine B. La machine A est disponible 10 heures par jour, la machine B 12 heures.

Variables de décision : x_1 = nombre d'unités de P_1, x_2 = nombre d'unités de P_2.

Modèle :

$$\begin{aligned} \max \quad & z = 5x_1 + 4x_2 \\ \text{s.c.} \quad & 2x_1 + x_2 \leq 10 \quad \text{(machine A)} \\ & x_1 + 3x_2 \leq 12 \quad \text{(machine B)} \\ & x_1, x_2 \geq 0 \end{aligned}$$

Exemple 1.5 (Problème de mélange)**.** Un éleveur souhaite préparer un mélange alimentaire pour ses animaux au moindre coût. Deux ingrédients sont disponibles :

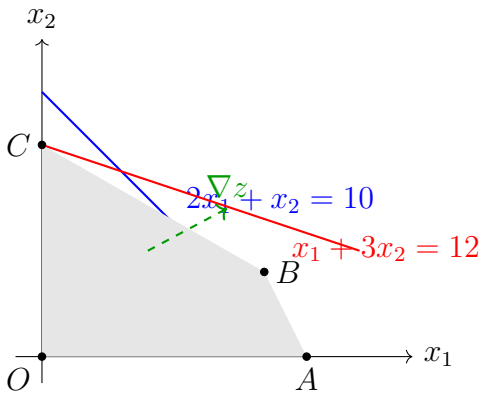
	Protéines (%/kg)	Lipides (%/kg)	Coût (EUR/kg)
Ingrédient 1	20	5	3
Ingrédient 2	10	8	2
Besoin minimal	15	6	—

Soit x_1, x_2 les quantités (en kg) des deux ingrédients. Le mélange doit peser 1 kg : $x_1 + x_2 = 1$.

$$\begin{aligned} \min \quad & z = 3x_1 + 2x_2 \\ \text{s.c.} \quad & 20x_1 + 10x_2 \geq 15 \quad \text{(protéines)} \\ & 5x_1 + 8x_2 \geq 6 \quad \text{(lipides)} \\ & x_1 + x_2 = 1 \\ & x_1, x_2 \geq 0 \end{aligned}$$

1.7 Résolution graphique (aperçu)

Pour les problèmes à deux variables, on peut représenter l'ensemble réalisable dans le plan et trouver l'optimum graphiquement.



Le sommet $B = (4.2, 1.6)$ maximise $z = 5(4.2) + 4(1.6) = 27.4$. On retrouvera ce résultat formellement au chapitre 3 par la méthode du simplexe.

1.8 Introduction aux solveurs

Résolution avec SciPy

```python
from scipy.optimize import linprog

# min c^T x  =>  on maximise en changeant le signe
c = [-5, -4]  # max 5x1 + 4x2

# Contraintes : A_ub @ x <= b_ub
A_ub = [[2, 1],   # 2x1 + x2 <= 10
        [1, 3]]   # x1 + 3x2 <= 12
b_ub = [10, 12]

# Bornes : x1, x2 >= 0
x_bounds = (0, None)
bounds = [x_bounds, x_bounds]

result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds,
                 method='highs')
print(f"x* = {result.x}")
print(f"z* = {-result.fun:.2f}")
```

Sortie

```
x* = [4.2 1.6] z* = 27.40
```

Résolution avec PuLP

```python
import pulp

prob = pulp.LpProblem("Production", pulp.LpMaximize)

x1 = pulp.LpVariable("x1", lowBound=0)
```

```python
x2 = pulp.LpVariable("x2", lowBound=0)

# Objectif
prob += 5*x1 + 4*x2, "Profit"

# Contraintes
prob += 2*x1 + x2 <= 10, "Machine_A"
prob += x1 + 3*x2 <= 12, "Machine_B"

prob.solve(pulp.PULP_CBC_CMD(msg=0))

print(f"Statut : {pulp.LpStatus[prob.status]}")
print(f"x1 = {x1.varValue}, x2 = {x2.varValue}")
print(f"Profit = {pulp.value(prob.objective):.2f}")
```

Sortie

Statut : Optimal x1 = 4.2, x2 = 1.6 Profit = 27.40

1.9 Complexité algorithmique : rappels

Définition 1.6 (Notation $\mathcal{O}$)**.** On dit qu'un algorithme a une complexité en $\mathcal{O}(g(n))$ s'il existe des constantes $C > 0$ et n_0 telles que le nombre d'opérations élémentaires est majoré par $C \cdot g(n)$ pour tout $n \geq n_0$.

Classe	Complexité	Exemple
$\mathcal{P}$	Polynomial	Simplexe (en pratique), Dijkstra
$\mathcal{NP}$	Vérifiable en temps polynomial	Problème du sac à dos
$\mathcal{NP}$-complet	Aussi dur que tout problème $\mathcal{NP}$	TSP (décision)
$\mathcal{NP}$-difficile	Au moins aussi dur que $\mathcal{NP}$-complet	TSP (optimisation)

Complexité du simplexe

Bien que le simplexe soit en temps exponentiel dans le pire cas (exemples de Klee-Minty), il est extrêmement efficace en pratique. Les méthodes de points intérieurs offrent une complexité polynomiale garantie : $\mathcal{O}(n^{3.5}L)$ où L est la taille de l'entrée.

1.10 Vocabulaire fondamental

Définition 1.7 (Solution réalisable)**.** Une solution $\mathbf{x}$ est dite **réalisable** (ou **admissible**) si elle satisfait toutes les contraintes du problème. L'ensemble des solutions réalisables est noté $\mathcal{F}$ (*feasible set*).

Définition 1.8 (Solution optimale)**.** Une solution réalisable $\mathbf{x}^*$ est **optimale** si $f(\mathbf{x}^*) \leq f(\mathbf{x})$ pour tout $\mathbf{x} \in \mathcal{F}$ (dans le cas d'une minimisation).

Définition 1.9 (Problème non borné)**.** Un problème est dit **non borné** si la fonction objectif peut prendre des valeurs arbitrairement petites (minimisation) ou grandes (maximisation) sur $\mathcal{F}$.

Définition 1.10 (Problème non réalisable)**.** Un problème est dit **non réalisable** (*infeasible*) si $\mathcal{F} = \emptyset$, c'est-à-dire si aucune solution ne satisfait simultanément toutes les contraintes.

1.11 Exercices

Exercice 1.1 ($\star$ – Modélisation)**.** Une entreprise fabrique trois produits A, B et C avec les données suivantes :

	Machine 1 (h)	Machine 2 (h)	Machine 3 (h)	Profit (EUR)
A	1	2	1	10
B	2	1	3	15
C	1	1	2	12
Disponibilité	40	50	60	

Formuler le programme linéaire qui maximise le profit total.

Exercice 1.2 ($\star$ – Résolution graphique)**.** Résoudre graphiquement :

$$\begin{aligned} \max \quad & z = 3x_1 + 2x_2 \\ \text{s.c.} \quad & x_1 + x_2 \leq 4 \\ & x_1 + 3x_2 \leq 6 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Exercice 1.3 ($\star\star$ – Problème de régime alimentaire)**.** Un diététicien doit composer un repas à coût minimal contenant au moins 2000 kcal, 55 g de protéines et 800 mg de calcium. Quatre aliments sont disponibles avec les données suivantes :

	kcal/100g	Protéines (g/100g)	Calcium (mg/100g)	Coût (EUR/100g)
Pain	265	9	15	0.30
Lait	65	3.5	120	0.15
Fromage	350	25	700	1.20
Pomme	52	0.3	6	0.40

Formuler le PL correspondant.

Exercice 1.4 ($\star\star$ – Code Python)**.** Implémenter et résoudre le problème de l'exercice 1 avec PuLP. Vérifier le résultat avec `scipy.optimize.linprog`.

Exercice 1.5 ($\star\star\star$ – Modélisation avec variables binaires)**.** Un investisseur dispose de 100 000 EUR et doit choisir parmi 5 projets d'investissement. Chaque projet i nécessite un investissement de c_i et rapporte r_i. L'investisseur ne peut investir que dans un projet en entier (pas de fraction). De plus, s'il investit dans le projet 3, il doit aussi investir dans le projet 1. Formuler ce problème comme un programme linéaire en nombres entiers (PLNE).

Chapitre 2

Programmation Linéaire : Formulation et Géométrie

En 1947, George Dantzig, mathématicien au Pentagone, invente l'algorithme du simplexe pour résoudre les problèmes de planification logistique de l'armée américaine. Son idée : se déplacer le long des arêtes d'un polyèdre convexe, de sommet en sommet, en améliorant l'objectif à chaque pas. Cet algorithme, d'une simplicité géométrique remarquable, a transformé la recherche opérationnelle et reste aujourd'hui l'un des outils les plus utilisés en optimisation. Ce chapitre pose les bases : qu'est-ce qu'un programme linéaire, quelle est sa géométrie, et pourquoi la solution se trouve-t-elle toujours à un sommet ?

2.1 Forme générale d'un programme linéaire

Définition 2.1 (Programme linéaire)**.** Un **programme linéaire** (PL) est un problème d'optimisation de la forme :

$$\begin{aligned} \min \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.c.} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

où $\mathbf{c} \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, et $\mathbf{x} \in \mathbb{R}^n$ est le vecteur de variables de décision.

Remarque 2.2. Maximiser $\mathbf{c}^\top \mathbf{x}$ est équivalent à minimiser $(-\mathbf{c})^\top \mathbf{x}$. On peut donc toujours se ramener à un problème de minimisation.

2.2 Formes équivalentes

2.2.1 Forme standard

Définition 2.3 (Forme standard)**.** Un PL est sous **forme standard** si :

$$\begin{aligned} \min \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.c.} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

Toutes les contraintes sont des égalités et toutes les variables sont non négatives.

2.2.2 Passage à la forme standard

Règles de conversion

1. **Inégalité** $\leq$: ajouter une variable d'écart $s_i \geq 0$:
$$a_{i1}x_1 + \cdots + a_{in}x_n + s_i = b_i$$
2. **Inégalité** $\geq$: soustraire une variable de surplus $s_i \geq 0$:
$$a_{i1}x_1 + \cdots + a_{in}x_n - s_i = b_i$$
3. **Variable libre** ($x_j \in \mathbb{R}$) : poser $x_j = x_j^+ - x_j^-$ avec $x_j^+, x_j^- \geq 0$.
4. **Maximisation** : $\max \mathbf{c}^\top \mathbf{x} = -\min(-\mathbf{c}^\top \mathbf{x})$.

Exemple 2.4 (Mise sous forme standard)**.** Considérons :

$$\begin{aligned} \max \quad & 3x_1 + 5x_2 \\ \text{s.c.} \quad & x_1 + 2x_2 \leq 12 \\ & 2x_1 + x_2 \geq 4 \\ & x_1 \geq 0,\ x_2 \text{ libre} \end{aligned}$$

On pose $x_2 = x_2^+ - x_2^-$ avec $x_2^+, x_2^- \geq 0$, et on ajoute les variables d'écart/surplus :

$$\begin{aligned} \min \quad & -3x_1 - 5x_2^+ + 5x_2^- \\ \text{s.c.} \quad & x_1 + 2x_2^+ - 2x_2^- + s_1 = 12 \\ & 2x_1 + x_2^+ - x_2^- - s_2 = 4 \\ & x_1, x_2^+, x_2^-, s_1, s_2 \geq 0 \end{aligned}$$

2.3 Forme canonique

Définition 2.5 (Forme canonique)**.** Un PL est sous **forme canonique** si :

$$\begin{aligned} \max \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.c.} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

(maximisation, contraintes $\leq$, variables ≥ 0).

2.4 Géométrie de la programmation linéaire

2.4.1 Hyperplans et demi-espaces

Définition 2.6 (Hyperplan)**.** Un **hyperplan** de $\mathbb{R}^n$ est un ensemble de la forme :

$$H = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{a}^\top \mathbf{x} = \beta\}$$

où $\mathbf{a} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ et $\beta \in \mathbb{R}$.

Définition 2.7 (Demi-espace)**.** Un **demi-espace** est l'un des deux ensembles déterminés par un hyperplan :

$$H^+ = \{\mathbf{x} : \mathbf{a}^\top \mathbf{x} \leq \beta\} \qquad \text{ou} \qquad H^- = \{\mathbf{x} : \mathbf{a}^\top \mathbf{x} \geq \beta\}$$

2.4.2 Ensembles convexes et polyèdres

Définition 2.8 (Ensemble convexe)**.** Un ensemble $C \subseteq \mathbb{R}^n$ est **convexe** si pour tous $\mathbf{x}, \mathbf{y} \in C$ et tout $\lambda \in [0, 1]$:

$$\lambda \mathbf{x} + (1 - \lambda)\mathbf{y} \in C$$

Définition 2.9 (Polyèdre)**.** Un **polyèdre** est l'intersection d'un nombre fini de demi-espaces :

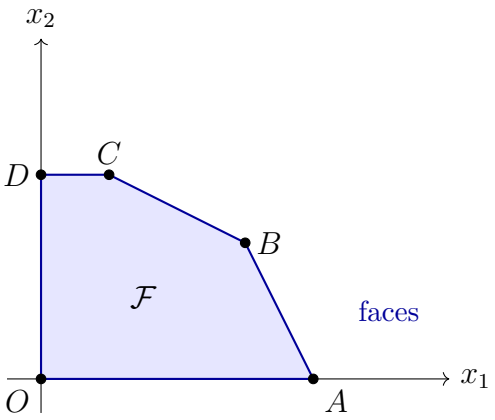
$$P = \{\mathbf{x} \in \mathbb{R}^n : A\mathbf{x} \leq \mathbf{b}\}$$

Un polyèdre borné est appelé **polytope**.

Théorème 2.10 (L'ensemble réalisable est un polyèdre)**.** *L'ensemble réalisable d'un programme linéaire :*

$$\mathcal{F} = \{\mathbf{x} \in \mathbb{R}^n : A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$$

est un polyèdre convexe (éventuellement vide ou non borné).



2.4.3 Sommets et solutions de base réalisables

Définition 2.11 (Sommet)**.** Un point $\mathbf{v} \in P$ est un **sommet** (ou **point extrêmal**) du polyèdre P s'il n'existe pas deux points distincts $\mathbf{x}, \mathbf{y} \in P$ tels que $\mathbf{v} = \frac{1}{2}(\mathbf{x} + \mathbf{y})$.

Définition 2.12 (Solution de base réalisable (SBR))**.** Considérons le système $A\mathbf{x} = \mathbf{b}$ avec $A \in \mathbb{R}^{m \times n}$, $m \leq n$, $\text{rang}(A) = m$. On partitionne les colonnes en B (base, m colonnes linéairement indépendantes) et N (hors-base) :

$$\mathbf{x}_B = B^{-1}\mathbf{b}, \qquad \mathbf{x}_N = \mathbf{0}$$

Si $\mathbf{x}_B \geq \mathbf{0}$, alors $\mathbf{x}$ est une **solution de base réalisable**.

Théorème 2.13 (Équivalence sommet–SBR)**.** *Soit $P = \{\mathbf{x} : A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0\}$ un polyèdre. Les assertions suivantes sont équivalentes :*

1. $\mathbf{v}$ *est un sommet de P.*
2. $\mathbf{v}$ *est une solution de base réalisable.*
3. *Les colonnes de A correspondant aux composantes strictement positives de $\mathbf{v}$ sont linéairement indépendantes.*

2.5 Théorème fondamental de la programmation linéaire

Théorème 2.14 (Théorème fondamental de la PL)**.** *Considérons le programme linéaire* $\min\{\mathbf{c}^\top \mathbf{x} : A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0\}$.

1. *Si l'ensemble réalisable est non vide, il possède au moins un sommet.*
2. *Si le problème admet une solution optimale finie, alors il existe une solution optimale qui est un sommet de l'ensemble réalisable.*
3. *Le nombre de sommets est fini (au plus* $\binom{n}{m}$*).*

Pourquoi l'optimum est en un sommet ?

Les lignes de niveau de $\mathbf{c}^\top \mathbf{x}$ sont des hyperplans parallèles. On déplace ces hyperplans dans la direction $-\mathbf{c}$ (minimisation) jusqu'à toucher le polyèdre. Le dernier point de contact est nécessairement un sommet (ou une face contenant des sommets).

2.6 Exemples de formulation

Exemple 2.15 (Problème de planification de production)**.** Une entreprise produit n articles sur m machines pendant T périodes. Soit x_{it} la quantité de l'article i produite à la période t, et s_{it} le stock en fin de période.

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{t=1}^{T} (c_{it} x_{it} + h_{it} s_{it}) \\
\text{s.c.} \quad & s_{i,t-1} + x_{it} - s_{it} = d_{it} && \forall\, i, t \quad \text{(conservation)} \\
& \sum_{i=1}^{n} a_{ij} x_{it} \leq C_{jt} && \forall\, j, t \quad \text{(capacité machine } j\text{)} \\
& x_{it}, s_{it} \geq 0 && \forall\, i, t
\end{aligned}
$$

où c_{it} est le coût unitaire de production, h_{it} le coût de stockage, d_{it} la demande, a_{ij} le temps machine unitaire.

Exemple 2.16 (Problème de régime alimentaire (revisité))**.** Avec les données du chapitre précédent, on formule :

$$
\begin{aligned}
\min \quad & 0.30x_1 + 0.15x_2 + 1.20x_3 + 0.40x_4 \\
\text{s.c.} \quad & 265x_1 + 65x_2 + 350x_3 + 52x_4 \geq 2000 \\
& 9x_1 + 3.5x_2 + 25x_3 + 0.3x_4 \geq 55 \\
& 15x_1 + 120x_2 + 700x_3 + 6x_4 \geq 800 \\
& x_1, x_2, x_3, x_4 \geq 0
\end{aligned}
$$

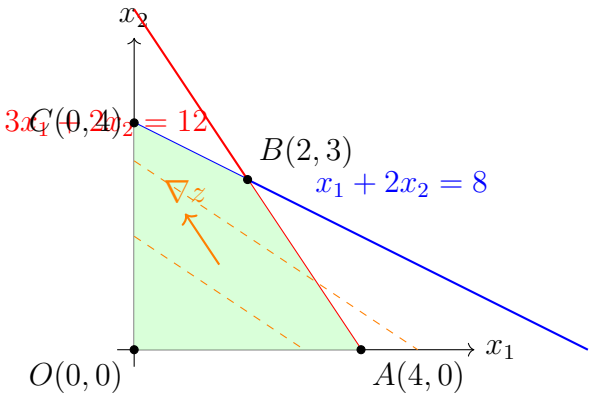
(les x_i sont en hectogrammes, soit 100 g).

2.7 Résolution graphique détaillée

Exemple 2.17 (Résolution graphique complète)**.** Résoudre :

$$\begin{aligned} \max \quad & z = 2x_1 + 3x_2 \\ \text{s.c.} \quad & x_1 + 2x_2 \leq 8 \\ & 3x_1 + 2x_2 \leq 12 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Étape 1 : Tracer les droites de contraintes.



Étape 2 : Évaluer z aux sommets.

Sommet	$z = 2x_1 + 3x_2$
$O(0,0)$	0
$A(4,0)$	8
$B(2,3)$	**13**
$C(0,4)$	12

L'optimum est $z^* = 13$ atteint en $B = (2,3)$.

2.8 Cas particuliers

Définition 2.18 (Problème non réalisable)**.** Un PL est **non réalisable** (*infeasible*) si $\mathcal{F} = \emptyset$.

Exemple 2.19 (Non réalisabilité)**.**

$$x_1 + x_2 \leq 2, \qquad x_1 + x_2 \geq 5, \qquad x_1, x_2 \geq 0$$

Les deux premières contraintes sont contradictoires.

Définition 2.20 (Problème non borné)**.** Un PL est **non borné** si $z \to +\infty$ (max) ou $z \to -\infty$ (min) sur $\mathcal{F}$.

Définition 2.21 (Solutions optimales multiples)**.** Il y a **solutions optimales multiples** lorsqu'une arête entière du polyèdre est optimale (le gradient est parallèle à une face).

Définition 2.22 (Dégénérescence)**.** Une SBR est **dégénérée** si au moins une variable de base est nulle. Cela arrive quand plus de n contraintes sont actives en un sommet.

2.9 Implémentation Python

Visualisation de la région réalisable

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import linprog

# Résolution
c = [-2, -3]
A_ub = [[1, 2], [3, 2]]
b_ub = [8, 12]
res = linprog(c, A_ub=A_ub, b_ub=b_ub,
              bounds=[(0, None), (0, None)], method='highs')
print(f"Optimum : z = {-res.fun:.1f} en x = {res.x}")

# Tracé
fig, ax = plt.subplots(figsize=(6, 5))
x1 = np.linspace(0, 5, 300)

# Contraintes
ax.plot(x1, (8 - x1)/2, 'b-', label=r'$x_1+2x_2=8$')
ax.plot(x1, (12 - 3*x1)/2, 'r-', label=r'$3x_1+2x_2=12$')
ax.fill([0, 4, 2, 0], [0, 0, 3, 4], alpha=0.2, color='green')

ax.set_xlim(-0.5, 5.5)
ax.set_ylim(-0.5, 5.5)
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
ax.legend()
ax.set_title('Région réalisable')
plt.tight_layout()
plt.savefig('region_realisable.pdf')
```

Sortie

```
Optimum : z = 13.0 en x = [2. 3.]
```

2.10 Exercices

Exercice 2.1 ($\star$ – Mise sous forme standard)**.** Mettre sous forme standard le PL suivant :

$$\begin{aligned} \max \quad & 4x_1 - 2x_2 + 7x_3 \\ \text{s.c.} \quad & x_1 + x_2 + x_3 \leq 20 \\ & 2x_1 - x_2 + 3x_3 \geq 10 \\ & x_1 \geq 0,\ x_2 \geq 0,\ x_3 \text{ libre} \end{aligned}$$

Exercice 2.2 ($\star$ – Résolution graphique)**.** Résoudre graphiquement et identifier tous les

sommets :

$$\begin{aligned} \max \quad & z = x_1 + 2x_2 \\ \text{s.c.} \quad & x_1 + x_2 \leq 5 \\ & 2x_1 + x_2 \leq 8 \\ & x_2 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Exercice 2.3 ($\star\star$ – Polyèdre vide ou non borné)**.** Pour chacun des PL suivants, déterminer s'il est non réalisable, non borné, ou s'il admet une solution optimale finie :

1. $\min x_1 - x_2$ s.c. $x_1 + x_2 \geq 1$, $x_1, x_2 \geq 0$.
2. $\max 2x_1 + x_2$ s.c. $x_1 - x_2 \leq 1$, $x_1, x_2 \geq 0$.
3. $\min x_1$ s.c. $x_1 + x_2 \leq 4$, $x_1 - x_2 \leq 2$, $-x_1 + x_2 \leq 2$, $x_1, x_2 \geq 0$.

Exercice 2.4 ($\star\star$ – Formulation complète)**.** Une raffinerie transforme du pétrole brut en trois produits : essence, diesel et kérosène. Deux types de brut (B_1 et B_2) sont disponibles. Le tableau suivant donne les rendements (en %) et les coûts :

	Essence	Diesel	Kérosène	Coût (EUR/baril)
B_1	40%	35%	25%	50
B_2	30%	45%	25%	60
Demande min. (barils)	3000	2000	1500	

Formuler le PL minimisant le coût total d'achat du brut.

Exercice 2.5 ($\star\star\star$ – Démonstration)**.** Démontrer que l'intersection de deux ensembles convexes est convexe. En déduire que tout polyèdre est convexe.

Chapitre 3

Méthode du Simplexe

L'idée géométrique est lumineuse : puisque l'optimum d'un programme linéaire se trouve à un sommet du polyèdre des contraintes, il suffit de se déplacer de sommet en sommet en améliorant l'objectif à chaque pas. C'est exactement ce que fait la méthode du simplexe, inventée par George Dantzig en 1947. Malgré une complexité théorique exponentielle dans le pire cas (démontrée par Klee et Minty en 1972), le simplexe reste, en pratique, l'un des algorithmes les plus efficaces jamais conçus — une anomalie théorique fascinante qui défie encore notre compréhension.

3.1 Principe de l'algorithme

Le théorème fondamental de la PL (Théorème 2.14) affirme que si un PL admet un optimum fini, celui-ci est atteint en un sommet du polyèdre réalisable. La **méthode du simplexe**, due à George Dantzig (1947), parcourt les sommets adjacents en améliorant l'objectif à chaque itération.

Idée générale du simplexe

1. Partir d'un sommet (SBR) initial.
2. Tester l'optimalité (coûts réduits).
3. Si non optimal, pivoter vers un sommet adjacent améliorant.
4. Répéter jusqu'à l'optimalité ou détection de non-bornitude.

3.2 Notations et rappels

Considérons le PL sous forme standard :

$$\begin{aligned} \min \quad & z = \mathbf{c}^\top \mathbf{x} \\ \text{s.c.} \quad & A\mathbf{x} = \mathbf{b}, \quad \mathbf{x} \geq \mathbf{0} \end{aligned}$$

avec $A \in \mathbb{R}^{m \times n}$, $\text{rang}(A) = m$, $\mathbf{b} \geq \mathbf{0}$.

On partitionne $A = [B \mid N]$, $\mathbf{x} = (\mathbf{x}_B, \mathbf{x}_N)^\top$, $\mathbf{c} = (\mathbf{c}_B, \mathbf{c}_N)^\top$:

$$B\mathbf{x}_B + N\mathbf{x}_N = \mathbf{b} \quad \Rightarrow \quad \mathbf{x}_B = B^{-1}\mathbf{b} - B^{-1}N\mathbf{x}_N$$

Définition 3.1 (Coûts réduits)**.** Les **coûts réduits** des variables hors-base sont :

$$\bar{c}_j = c_j - \mathbf{c}_B^\top B^{-1}\mathbf{a}_j$$

où $\mathbf{a}_j$ est la j-ème colonne de A.

Théorème 3.2 (Critère d'optimalité)**.** *Une SBR est optimale si et seulement si tous les coûts réduits sont non négatifs : $\bar{c}_j \geq 0$ pour tout j hors-base.*

3.3 Le tableau du simplexe

Le tableau du simplexe organise toute l'information nécessaire :

	x_1	x_2	$\cdots$	x_n	
z	$\bar{c}_1$	$\bar{c}_2$	$\cdots$	$\bar{c}_n$	$-z$
x_{B_1}	$\bar{a}_{11}$	$\bar{a}_{12}$	$\cdots$	$\bar{a}_{1n}$	$\bar{b}_1$
x_{B_2}	$\bar{a}_{21}$	$\bar{a}_{22}$	$\cdots$	$\bar{a}_{2n}$	$\bar{b}_2$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
x_{B_m}	$\bar{a}_{m1}$	$\bar{a}_{m2}$	$\cdots$	$\bar{a}_{mn}$	$\bar{b}_m$

où $\bar{A} = B^{-1}A$, $\bar{\mathbf{b}} = B^{-1}\mathbf{b}$.

3.4 Itération du simplexe

Une itération du simplexe (minimisation)

1. **Test d'optimalité** : si $\bar{c}_j \geq 0$ pour tout j, STOP (solution optimale trouvée).
2. **Variable entrante** : choisir $k = \arg\min_j \bar{c}_j$ (coût réduit le plus négatif).
3. **Test de non-bornitude** : si $\bar{a}_{ik} \leq 0$ pour tout i, STOP (problème non borné).
4. **Variable sortante** : appliquer la règle du ratio minimum :

$$r = \arg\min_{i:\bar{a}_{ik}>0} \frac{\bar{b}_i}{\bar{a}_{ik}}$$

5. **Pivot** : l'élément pivot est $\bar{a}_{rk}$. Effectuer le pivotage de Gauss-Jordan pour mettre à jour le tableau.

Règle du ratio

On ne divise que par les coefficients **strictement positifs** $\bar{a}_{ik} > 0$. Si tous sont ≤ 0, le problème est non borné : on peut augmenter x_k indéfiniment.

3.5 Exemple complet

Exemple 3.3 (Simplexe pas à pas)**.** Résoudre :

$$\begin{aligned} \max \quad & z = 5x_1 + 4x_2 \\ \text{s.c.} \quad & 6x_1 + 4x_2 \leq 24 \\ & x_1 + 2x_2 \leq 6 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Forme standard : on ajoute les variables d'écart $s_1, s_2 \geq 0$ et on convertit en minimisation ($\min -5x_1 - 4x_2$).

Tableau initial :

	x_1	x_2	s_1	s_2	RHS
z	-5	-4	0	0	0
s_1	6	4	1	0	24
s_2	1	2	0	1	6

Itération 1 : La variable entrante est x_1 ($\bar{c}_1 = -5$). Ratios : $24/6 = 4$, $6/1 = 6$. Pivot : ligne s_1, colonne x_1 (pivot = 6).

On divise la ligne 1 par 6, puis on élimine x_1 des autres lignes :

	x_1	x_2	s_1	s_2	RHS
z	0	$-\frac{2}{3}$	$\frac{5}{6}$	0	20
x_1	1	$\frac{2}{3}$	$\frac{1}{6}$	0	4
s_2	0	$\frac{4}{3}$	$-\frac{1}{6}$	1	2

Itération 2 : La variable entrante est x_2 ($\bar{c}_2 = -2/3$). Ratios : $4/(2/3) = 6$, $2/(4/3) = 3/2$. Pivot : ligne s_2, colonne x_2 (pivot = $4/3$).

	x_1	x_2	s_1	s_2	RHS
z	0	0	$\frac{3}{4}$	$\frac{1}{2}$	21
x_1	1	0	$\frac{1}{4}$	$-\frac{1}{2}$	3
x_2	0	1	$-\frac{1}{8}$	$\frac{3}{4}$	$\frac{3}{2}$

Tous les coûts réduits sont ≥ 0 : solution optimale !

$x_1^* = 3$, $x_2^* = 3/2$, $z^* = 5(3) + 4(3/2) = 21$.

3.6 Initialisation : méthode du Grand M

Quand le tableau initial ne fournit pas directement une base réalisable (contraintes $\geq$ ou $=$), on utilise des **variables artificielles**.

Définition 3.4 (Méthode du Grand M)**.** Pour chaque contrainte d'égalité ou $\geq$ (après ajout d'une variable de surplus), on ajoute une variable artificielle $a_i \geq 0$ dans la fonction objectif avec un coût de pénalité $+M$ (minimisation) où M est un grand nombre positif.

Exemple 3.5 (Grand M)**.**

$$\begin{aligned} \min \quad & z = 2x_1 + 3x_2 \\ \text{s.c.} \quad & x_1 + x_2 = 4 \\ & x_1 + 2x_2 \geq 6 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Après ajout de la variable de surplus s_1 et des artificielles a_1, a_2 :

$$\begin{aligned} \min \quad & z = 2x_1 + 3x_2 + Ma_1 + Ma_2 \\ \text{s.c.} \quad & x_1 + x_2 + a_1 = 4 \\ & x_1 + 2x_2 - s_1 + a_2 = 6 \\ & x_1, x_2, s_1, a_1, a_2 \geq 0 \end{aligned}$$

Si à l'optimum $a_1 = a_2 = 0$, la solution est réalisable pour le PL original. Sinon, le PL original est non réalisable.

3.7 Initialisation : méthode à deux phases

Méthode à deux phases

Phase I : Résoudre le problème auxiliaire :

$$\min \sum_i a_i \quad \text{s.c.} \quad A\mathbf{x} + I\mathbf{a} = \mathbf{b}, \quad \mathbf{x}, \mathbf{a} \geq 0$$

- Si la valeur optimale est > 0 : le PL original est **non réalisable**.
- Si la valeur optimale est $= 0$: on obtient une SBR pour le PL original.

Phase II : Utiliser la SBR obtenue comme point de départ pour résoudre le PL original avec la fonction objectif d'origine.

Grand M vs. Deux phases

- La méthode du Grand M est plus simple à programmer mais peut poser des problèmes numériques si M est mal choisi.
- La méthode à deux phases est numériquement plus stable et préférée dans les implémentations professionnelles.

3.8 Cas de dégénérescence et cyclage

Définition 3.6 (Dégénérescence)**.** Une itération est **dégénérée** si le ratio minimum est nul : la variable sortante est déjà nulle et z ne change pas.

Théorème 3.7 (Risque de cyclage)**.** *En présence de dégénérescence, le simplexe peut théoriquement* ***cycler*** *(boucler indéfiniment entre les mêmes bases).*

Définition 3.8 (Règle de Bland)**.** La **règle de Bland** (règle du plus petit indice) élimine le cyclage : choisir comme variable entrante la variable hors-base de plus petit indice ayant un coût réduit négatif, et comme variable sortante la variable de base de plus petit indice réalisant le ratio minimum.

Théorème 3.9 (Anti-cyclage de Bland)**.** *Sous la règle de Bland, l'algorithme du simplexe termine en un nombre fini d'itérations.*

3.9 Complexité du simplexe

Proposition 3.10 (Nombre de sommets)**.** Un polyèdre défini par m contraintes et n variables a au plus $\binom{n}{m}$ sommets.

Remarque 3.11. Le simplexe a une complexité en $\mathcal{O}(2^n)$ dans le pire cas (exemples de **Klee-Minty**, 1972), mais en pratique il effectue environ $2m$ à $3m$ itérations pour un problème à m contraintes.

3.10 Simplexe révisé

Le **simplexe révisé** évite de maintenir le tableau complet. Il ne calcule que les informations nécessaires à chaque itération :

Simplexe révisé – une itération

1. Calculer $\boldsymbol{\pi}^\top = \mathbf{c}_B^\top B^{-1}$ (multiplicateurs du simplexe).
2. Pour chaque $j \notin$ base, calculer $\bar{c}_j = c_j - \boldsymbol{\pi}^\top \mathbf{a}_j$.
3. Si $\bar{c}_j \geq 0$ pour tout j : STOP.
4. Choisir k avec $\bar{c}_k < 0$ (variable entrante).
5. Calculer $\mathbf{d} = B^{-1}\mathbf{a}_k$ (direction).
6. Ratio minimum : $r = \arg\min_{i : d_i > 0} \bar{b}_i / d_i$.
7. Mettre à jour B^{-1} (produit de matrices élémentaires).

3.11 Implémentation Python

Simplexe tabulaire en Python

```python
import numpy as np

def simplex_tableau(c, A, b):
    """Résout max c^T x s.c. Ax <= b, x >= 0."""
    m, n = A.shape
    # Ajout des variables d'écart
    tableau = np.zeros((m + 1, n + m + 1))
```

```
    tableau[0, :n] = -c          # ligne objectif (min -c^T x)
    tableau[1:, :n] = A
    tableau[1:, n:n+m] = np.eye(m)
    tableau[1:, -1] = b

    basis = list(range(n, n + m))

    while True:
        # Variable entrante : coût réduit le plus négatif
        pivot_col = np.argmin(tableau[0, :-1])
        if tableau[0, pivot_col] >= -1e-10:
            break  # optimal

        # Ratios
        col = tableau[1:, pivot_col]
        rhs = tableau[1:, -1]
        ratios = np.full(m, np.inf)
        for i in range(m):
            if col[i] > 1e-10:
                ratios[i] = rhs[i] / col[i]

        pivot_row = np.argmin(ratios) + 1
        if ratios[pivot_row - 1] == np.inf:
            raise ValueError("Problème non borné")

        # Pivotage
        pivot_val = tableau[pivot_row, pivot_col]
        tableau[pivot_row] /= pivot_val
        for i in range(m + 1):
            if i != pivot_row:
                tableau[i] -= (tableau[i, pivot_col]
                               * tableau[pivot_row])
        basis[pivot_row - 1] = pivot_col

    # Extraction de la solution
    x = np.zeros(n)
    for i, var in enumerate(basis):
        if var < n:
            x[var] = tableau[i + 1, -1]

    return x, tableau[0, -1]  # (solution, -z_max)

# Test
c = np.array([5., 4.])
A = np.array([[6., 4.], [1., 2.]])
b = np.array([24., 6.])

x_opt, neg_z = simplex_tableau(c, A, b)
print(f"x* = {x_opt}")
print(f"z* = {-neg_z:.2f}")
```

Sortie

x* = [3. 1.5] z* = 21.00

Vérification avec SciPy

```
from scipy.optimize import linprog

c_min = [-5, -4]
A_ub = [[6, 4], [1, 2]]
b_ub = [24, 6]
res = linprog(c_min, A_ub=A_ub, b_ub=b_ub,
              bounds=[(0, None), (0, None)], method='highs')
print(f"x* = {res.x}, z* = {-res.fun:.2f}")
```

Sortie

x* = [3. 1.5], z* = 21.00

3.12 Exercices

Exercice 3.1 (⋆ – Simplexe tabulaire)**.** Résoudre par le simplexe tabulaire :

$$\begin{aligned} \max \quad & z = 3x_1 + 5x_2 \\ \text{s.c.} \quad & x_1 \leq 4 \\ & 2x_2 \leq 12 \\ & 3x_1 + 5x_2 \leq 25 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Exercice 3.2 (⋆⋆ – Grand M)**.** Résoudre par la méthode du Grand M :

$$\begin{aligned} \min \quad & z = 4x_1 + x_2 \\ \text{s.c.} \quad & 3x_1 + x_2 = 3 \\ & 4x_1 + 3x_2 \geq 6 \\ & x_1 + 2x_2 \leq 4 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Exercice 3.3 (⋆⋆ – Deux phases)**.** Reprendre l'exercice précédent avec la méthode à deux phases.

Exercice 3.4 (⋆⋆ – Dégénérescence)**.** Vérifier qu'une dégénérescence se produit dans :

$$\begin{aligned} \max \quad & z = 2x_1 + x_2 \\ \text{s.c.} \quad & x_1 + x_2 \leq 4 \\ & x_1 \leq 3 \\ & x_2 \leq 1 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Exercice 3.5 ($\star\star\star$ – Implémentation)**.** Implémenter la méthode à deux phases en Python et la tester sur un PL de votre choix ayant des contraintes $\geq$ et $=$.

Chapitre 4

Dualité en Programmation Linéaire

Tout problème d'optimisation cache un miroir : son *dual*. Cette idée, qui remonte à John von Neumann et à ses travaux sur la théorie des jeux dans les années 1940, est l'une des plus profondes de toute l'optimisation. En programmation linéaire, la dualité prend une forme particulièrement élégante : à tout problème de minimisation correspond un problème de maximisation dont la valeur optimale coïncide avec celle du primal (théorème de dualité forte). Le dual fournit non seulement des bornes certifiées, mais aussi une interprétation économique : les variables duales sont des « prix fictifs »qui mesurent la valeur marginale de chaque contrainte.

4.1 Motivation

Considérons un problème de minimisation. On cherche à encadrer la valeur optimale :

- Toute solution réalisable fournit une **borne supérieure**.
- Peut-on obtenir systématiquement des **bornes inférieures** ?

La théorie de la dualité répond positivement à cette question en associant à chaque PL (le **primal**) un autre PL (le **dual**).

4.2 Construction du dual

Définition 4.1 (Primal et dual)**.** Soit le primal (P) sous forme canonique :

$$\begin{aligned} (\mathrm{P}) \quad & \max \quad \mathbf{c}^\top \mathbf{x} \\ & \text{s.c.} \quad A\mathbf{x} \leq \mathbf{b} \\ & \qquad \mathbf{x} \geq \mathbf{0} \end{aligned}$$

Le **dual** (D) est :

$$\begin{aligned} (\mathrm{D}) \quad & \min \quad \mathbf{b}^\top \mathbf{y} \\ & \text{s.c.} \quad A^\top \mathbf{y} \geq \mathbf{c} \\ & \qquad \mathbf{y} \geq \mathbf{0} \end{aligned}$$

où $\mathbf{y} \in \mathbb{R}^m$ est le vecteur des **variables duales**.

Règles de construction du dual

Primal (max)		Dual (min)
Contrainte $\leq$	$\leftrightarrow$	Variable $y_i \geq 0$
Contrainte $\geq$	$\leftrightarrow$	Variable $y_i \leq 0$
Contrainte $=$	$\leftrightarrow$	Variable y_i libre
Variable $x_j \geq 0$	$\leftrightarrow$	Contrainte $\geq$
Variable $x_j \leq 0$	$\leftrightarrow$	Contrainte $\leq$
Variable x_j libre	$\leftrightarrow$	Contrainte $=$

Remarque 4.2. Le dual du dual est le primal : $(D(D)) = (P)$.

4.3 Théorèmes de dualité

Théorème 4.3 (Dualité faible)**.** *Si* $\mathbf{x}$ *est réalisable pour (P) et* $\mathbf{y}$ *est réalisable pour (D), alors :*

$$\mathbf{c}^\top \mathbf{x} \leq \mathbf{b}^\top \mathbf{y}$$

Démonstration. On a $A\mathbf{x} \leq \mathbf{b}$ et $\mathbf{y} \geq \mathbf{0}$, donc $\mathbf{y}^\top A\mathbf{x} \leq \mathbf{y}^\top \mathbf{b}$. De même, $A^\top \mathbf{y} \geq \mathbf{c}$ et $\mathbf{x} \geq \mathbf{0}$ donnent $\mathbf{x}^\top A^\top \mathbf{y} \geq \mathbf{c}^\top \mathbf{x}$. Donc $\mathbf{c}^\top \mathbf{x} \leq \mathbf{y}^\top A\mathbf{x} \leq \mathbf{b}^\top \mathbf{y}$. □

Corollaire 4.4 (Bornes)**.**

1. *Si (P) est non borné (*$\max = +\infty$*), alors (D) est non réalisable.*
2. *Si (D) est non borné (*$\min = -\infty$*), alors (P) est non réalisable.*

Théorème 4.5 (Dualité forte)**.** *Si (P) admet une solution optimale finie* $\mathbf{x}^*$*, alors (D) admet également une solution optimale finie* $\mathbf{y}^*$ *et :*

$$\mathbf{c}^\top \mathbf{x}^* = \mathbf{b}^\top \mathbf{y}^*$$

Interprétation

Le théorème de dualité forte affirme que la meilleure borne inférieure (venant du dual) coïncide avec la meilleure borne supérieure (venant du primal) à l'optimum. Il n'y a « pas d'écart »entre les deux.

4.4 Écarts complémentaires

Théorème 4.6 (Écarts complémentaires)**.** *Soient* $\mathbf{x}^*$ *et* $\mathbf{y}^*$ *des solutions réalisables du primal et du dual respectivement. Elles sont toutes deux optimales si et seulement si :*

$$y_i^* \left(b_i - \sum_j a_{ij} x_j^* \right) = 0, \quad i = 1, \ldots, m$$

$$x_j^* \left(\sum_i a_{ij} y_i^* - c_j \right) = 0, \quad j = 1, \ldots, n$$

En d'autres termes :

- Si une contrainte primale n'est **pas saturée**, la variable duale correspondante est nulle.
- Si une variable primale est **strictement positive**, la contrainte duale correspondante est saturée.

4.5 Interprétation économique

Définition 4.7 (Prix fictif (shadow price))**.** La variable duale y_i^* associée à la i-ème contrainte du primal s'appelle le **prix fictif** de la ressource i. Elle représente la variation marginale de la valeur optimale lorsque le membre droit b_i augmente d'une unité :

$$y_i^* = \frac{\partial z^*}{\partial b_i}$$

Exemple 4.8 (Interprétation des prix fictifs)**.** Reprenons le problème de production du chapitre précédent :

$$\begin{aligned} \max \quad & z = 5x_1 + 4x_2 \\ \text{s.c.} \quad & 6x_1 + 4x_2 \leq 24 \quad (y_1) \\ & x_1 + 2x_2 \leq 6 \quad (y_2) \\ & x_1, x_2 \geq 0 \end{aligned}$$

Le dual est :

$$\begin{aligned} \min \quad & w = 24y_1 + 6y_2 \\ \text{s.c.} \quad & 6y_1 + y_2 \geq 5 \\ & 4y_1 + 2y_2 \geq 4 \\ & y_1, y_2 \geq 0 \end{aligned}$$

Du tableau final du simplexe (chapitre 3), on lit : $y_1^* = 3/4$, $y_2^* = 1/2$. On vérifie : $w^* = 24(3/4) + 6(1/2) = 21 = z^*$.

Interprétation : augmenter la capacité de la contrainte 1 de 24 à 25 augmente z^* d'environ $3/4 = 0.75$.

4.6 Lecture du dual dans le tableau du simplexe

Proposition 4.9 (Variables duales et tableau final)**.** Dans le tableau final du simplexe (primal en forme standard avec variables d'écart), les valeurs des variables duales optimales se lisent dans la **ligne objectif** aux colonnes des variables d'écart :

$$y_i^* = \bar{c}_{s_i} \quad \text{(coût réduit de la variable d'écart } s_i\text{)}$$

4.7 Le dual dans le simplexe

La méthode du **simplexe dual** résout le dual directement sur le tableau du primal. Elle est utile quand la solution de base est **dual-réalisable** (tous les coûts réduits ≥ 0) mais pas primal-réalisable ($\bar{b}_i < 0$ pour certains i).

Simplexe dual – une itération

1. **Test d'optimalité primal** : si $\bar{b}_i \geq 0$ pour tout i, STOP (solution optimale).
2. **Variable sortante** : choisir $r = \arg\min_i \bar{b}_i$ (la ligne la plus négative).
3. **Test de non-réalisabilité** : si $\bar{a}_{rj} \geq 0$ pour tout j, STOP (primal non réalisable).
4. **Variable entrante** : appliquer le ratio dual :

$$k = \arg\min_{j:\bar{a}_{rj}<0} \frac{\bar{c}_j}{|\bar{a}_{rj}|}$$

5. **Pivot** : pivoter sur $\bar{a}_{rk}$.

4.8 Exemple numérique du dual

Exemple 4.10 (Construction et résolution du dual)**.**

$$(\text{P}) \quad \begin{aligned} \max \quad & z = 4x_1 + 3x_2 \\ \text{s.c.} \quad & 2x_1 + x_2 \leq 10 \\ & x_1 + 3x_2 \leq 12 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Le dual :

$$(\text{D}) \quad \begin{aligned} \min \quad & w = 10y_1 + 12y_2 \\ \text{s.c.} \quad & 2y_1 + y_2 \geq 4 \\ & y_1 + 3y_2 \geq 3 \\ & y_1, y_2 \geq 0 \end{aligned}$$

Vérification avec PuLP

```python
import pulp

# Primal
P = pulp.LpProblem("Primal", pulp.LpMaximize)
x1 = pulp.LpVariable("x1", lowBound=0)
x2 = pulp.LpVariable("x2", lowBound=0)
P += 4*x1 + 3*x2
c1 = P.addConstraint(2*x1 + x2 <= 10, "c1")
c2 = P.addConstraint(x1 + 3*x2 <= 12, "c2")
P.solve(pulp.PULP_CBC_CMD(msg=0))
print(f"Primal : z* = {pulp.value(P.objective):.2f}")
print(f"  x* = ({x1.varValue}, {x2.varValue})")

# Dual
D = pulp.LpProblem("Dual", pulp.LpMinimize)
y1 = pulp.LpVariable("y1", lowBound=0)
```

```
y2 = pulp.LpVariable("y2", lowBound=0)
D += 10*y1 + 12*y2
D += 2*y1 + y2 >= 4
D += y1 + 3*y2 >= 3
D.solve(pulp.PULP_CBC_CMD(msg=0))
print(f"Dual : w* = {pulp.value(D.objective):.2f}")
print(f"  y* = ({y1.varValue}, {y2.varValue})")
```

Sortie

```
Primal : z* = 22.80 x* = (5.4, 2.2) Dual : w* = 22.80 y* = (1.8, 0.4)
```

4.9 Résumé des relations primal-dual

	(P) réalisable	**(P) non réalisable**
(D) réalisable	Optima finis égaux	(D) non borné
(D) non réalisable	(P) non borné	Les deux non réalisables

Remarque 4.11. Il est possible que le primal et le dual soient *tous deux* non réalisables. Exemple : max $x_1 - x_2$ s.c. $-x_1 + x_2 \leq -1$, $x_1 - x_2 \leq -1$, $x_1, x_2 \geq 0$.

4.10 Exercices

Exercice 4.1 ($\star$ – Écrire le dual)**.** Écrire le dual des PL suivants :

1. max $3x_1 + 2x_2$ s.c. $x_1 + x_2 \leq 5$, $2x_1 + x_2 \leq 8$, $x_1, x_2 \geq 0$.
2. min $5x_1 + 3x_2$ s.c. $x_1 + x_2 = 4$, $2x_1 + x_2 \geq 6$, $x_1, x_2 \geq 0$.

Exercice 4.2 ($\star\star$ – Vérification de la dualité forte)**.** Résoudre le primal et le dual de l'exercice 1(a) et vérifier que les valeurs optimales sont égales.

Exercice 4.3 ($\star\star$ – Écarts complémentaires)**.** Utiliser les écarts complémentaires pour trouver la solution duale optimale à partir de la solution primale $x_1^* = 3$, $x_2^* = 2$ du PL : max $4x_1 + 3x_2$ s.c. $2x_1 + x_2 \leq 8$, $x_1 + 2x_2 \leq 7$, $x_1, x_2 \geq 0$.

Exercice 4.4 ($\star\star$ – Interprétation économique)**.** Une usine a trois ressources R_1, R_2, R_3 avec les capacités 100, 80 et 120. Les prix fictifs optimaux sont $y_1^* = 2$, $y_2^* = 0$, $y_3^* = 3$. Interpréter ces valeurs. Un fournisseur propose 10 unités supplémentaires de R_2 pour 5 EUR. Faut-il accepter ?

Exercice 4.5 ($\star\star\star$ – Démonstration)**.** Démontrer le théorème de dualité faible pour le cas général (forme standard du primal).

Exercice 4.6 ($\star\star\star$ – Simplexe dual)**.** Appliquer le simplexe dual au problème :

$$\begin{aligned} \min \quad & z = 2x_1 + 3x_2 + x_3 \\ \text{s.c.} \quad & x_1 + x_2 + x_3 \geq 6 \\ & 2x_1 + x_3 \geq 4 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Chapitre 5

Analyse de Sensibilité

Un modèle d'optimisation n'est jamais parfait : les coûts, les capacités, les demandes sont estimés, et ces estimations comportent des marges d'erreur. La question naturelle est : jusqu'à quel point la solution optimale reste-t-elle valide si les paramètres changent légèrement ? C'est la question de l'*analyse de sensibilité*, un outil indispensable pour le décideur. Grâce à la structure du simplexe et à la dualité, on peut répondre à cette question sans résoudre un nouveau problème : il suffit d'examiner les intervalles de variation des paramètres pour lesquels la base optimale reste inchangée.

5.1 Introduction

En pratique, les paramètres d'un PL (coûts c_j, ressources b_i, coefficients a_{ij}) ne sont jamais connus avec une précision absolue. L'**analyse de sensibilité** étudie comment la solution optimale et la valeur optimale varient lorsque ces paramètres changent.

Définition 5.1 (Analyse de sensibilité)**.** L'**analyse de sensibilité** (ou **analyse post-optimale**) détermine les intervalles de variation des paramètres pour lesquels la **base optimale** reste inchangée.

5.2 Variation des membres droits b_i

5.2.1 Effet sur la solution

Soit $\mathbf{b}' = \mathbf{b} + \Delta\mathbf{b}$. La solution de base reste :

$$\mathbf{x}'_B = B^{-1}\mathbf{b}' = B^{-1}\mathbf{b} + B^{-1}\Delta\mathbf{b} = \bar{\mathbf{b}} + B^{-1}\Delta\mathbf{b}$$

La base reste réalisable tant que $\mathbf{x}'_B \geq \mathbf{0}$, soit :

$$\bar{\mathbf{b}} + B^{-1}\Delta\mathbf{b} \geq \mathbf{0}$$

5.2.2 Variation d'un seul b_i

Si seul b_i varie d'un montant δ :

$$\bar{b}_k + (B^{-1})_{ki} \cdot \delta \geq 0, \quad k = 1, \ldots, m$$

Intervalle de validité de b_i

La base reste optimale pour :

$$b_i \in \left[b_i - \min_{k:\,(B^{-1})_{ki}>0} \frac{\bar{b}_k}{(B^{-1})_{ki}},\; b_i + \min_{k:\,(B^{-1})_{ki}<0} \frac{-\bar{b}_k}{(B^{-1})_{ki}} \right]$$

Théorème 5.2 (Variation de z^*)**.** *Lorsque b_i varie de δ dans l'intervalle de validité :*

$$z^*(\delta) = z^* + y_i^* \cdot \delta$$

où y_i^* est le prix fictif (variable duale) de la contrainte i.

Exemple 5.3 (Variation de b_1)**.** Reprenons le PL du chapitre 3 :

$$\max\; z = 5x_1 + 4x_2 \quad \text{s.c.} \quad 6x_1 + 4x_2 \leq 24,\; x_1 + 2x_2 \leq 6$$

Le tableau final donne $B^{-1} = \begin{pmatrix} 1/4 & -1/2 \\ -1/8 & 3/4 \end{pmatrix}$, $\bar{\mathbf{b}} = (3, 3/2)^\top$, $y_1^* = 3/4$, $y_2^* = 1/2$.

Pour la variation de $b_1 = 24$:

- $(B^{-1})_{11} = 1/4 > 0$: $\delta \geq -\bar{b}_1/(1/4) = -12$
- $(B^{-1})_{21} = -1/8 < 0$: $\delta \leq -\bar{b}_2/(-1/8) = 12$

Donc $b_1 \in [24 - 12, 24 + 12] = [12, 36]$.

Si $b_1 = 30$: $z^*(30) = 21 + (3/4)(30 - 24) = 21 + 4.5 = 25.5$.

5.3 Variation des coefficients de l'objectif c_j

5.3.1 Variable de base

Si x_j est une variable de base, la variation de c_j affecte les coûts réduits des variables hors-base. La base reste optimale tant que tous les coûts réduits restent ≥ 0 (minimisation) ou ≤ 0 (maximisation dans le tableau).

Intervalle de validité de c_j (variable de base)

Les coûts réduits des variables hors-base k deviennent :

$$\bar{c}'_k = \bar{c}_k - \delta \cdot \bar{a}_{rk}$$

où r est la ligne de x_j dans la base et $\delta = c'_j - c_j$. La base reste optimale pour $\bar{c}'_k \geq 0$ pour tout k hors-base.

5.3.2 Variable hors-base

Si x_j est hors-base avec $\bar{c}_j > 0$ (maximisation), la base reste optimale tant que $\bar{c}_j + \delta \geq 0$, soit $c_j \geq c_j - \bar{c}_j$.

Exemple 5.4 (Variation de c_1)**.** Dans notre exemple, x_1 est de base. Les coûts réduits des variables hors-base (s_1, s_2) dans le tableau final sont : $\bar{c}_{s_1} = 3/4$, $\bar{c}_{s_2} = 1/2$.

La ligne de x_1 donne $\bar{a}_{1,s_1} = 1/4$, $\bar{a}_{1,s_2} = -1/2$.

- $\bar{c}'_{s_1} = 3/4 - \delta(1/4) \geq 0 \Rightarrow \delta \leq 3$
- $\bar{c}'_{s_2} = 1/2 - \delta(-1/2) = 1/2 + \delta/2 \geq 0 \Rightarrow \delta \geq -1$

Donc $c_1 \in [5-1, 5+3] = [4, 8]$.

5.4 Ajout d'une nouvelle variable

Si l'on ajoute une variable x_{n+1} avec coût c_{n+1} et colonne technologique $\mathbf{a}_{n+1}$, on calcule :

$$\bar{c}_{n+1} = c_{n+1} - \mathbf{c}_B^\top B^{-1} \mathbf{a}_{n+1}$$

- Si $\bar{c}_{n+1} \geq 0$ (minimisation) : la solution reste optimale sans utiliser x_{n+1}.
- Si $\bar{c}_{n+1} < 0$: il est avantageux de faire entrer x_{n+1} en base.

5.5 Ajout d'une nouvelle contrainte

Si l'on ajoute une contrainte $\mathbf{a}_{m+1}^\top \mathbf{x} \leq b_{m+1}$:

1. Vérifier si la solution optimale actuelle $\mathbf{x}^*$ satisfait la nouvelle contrainte : $\mathbf{a}_{m+1}^\top \mathbf{x}^* \leq b_{m+1}$.
2. Si oui : la solution reste optimale.
3. Si non : appliquer le **simplexe dual** en ajoutant la contrainte (avec une variable d'écart) au tableau final.

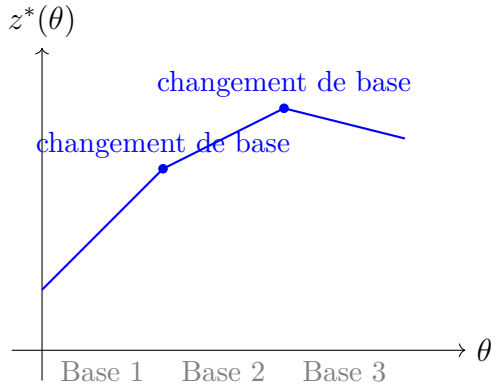
5.6 Analyse paramétrique

Définition 5.5 (Analyse paramétrique)**.** L'**analyse paramétrique** étudie la variation continue de z^* en fonction d'un paramètre θ :

$$\mathbf{b}(\theta) = \mathbf{b} + \theta \mathbf{d} \qquad \text{ou} \qquad \mathbf{c}(\theta) = \mathbf{c} + \theta \mathbf{e}$$

où $\mathbf{d}$ ou $\mathbf{e}$ sont des directions fixées.

Théorème 5.6 (Forme de $z^*(\theta)$)**.** *La valeur optimale $z^*(\theta)$ est une fonction **linéaire par morceaux** et **concave** (si on fait varier $\mathbf{b}$ en maximisation) de θ. Chaque segment correspond à une base optimale différente.*



5.7 Implémentation Python

Analyse de sensibilité avec SciPy

```python
from scipy.optimize import linprog
import numpy as np

c = [-5, -4]  # max 5x1 + 4x2
A_ub = [[6, 4], [1, 2]]
b_ub = [24, 6]
bounds = [(0, None), (0, None)]

# Solution de base
res = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds,
              method='highs')
print(f"z* = {-res.fun:.2f}, x* = {res.x}")

# Sensibilité de b1
print("\nVariation de b1 :")
for delta in [-12, -6, 0, 6, 12]:
    b_new = [24 + delta, 6]
    r = linprog(c, A_ub=A_ub, b_ub=b_new, bounds=bounds,
                method='highs')
    print(f"  b1={24+delta:3d} -> z*={-r.fun:.2f}, "
          f"x*=[{r.x[0]:.2f}, {r.x[1]:.2f}]")

# Sensibilité de c1
print("\nVariation de c1 :")
for c1 in [3, 4, 5, 6, 8]:
    c_new = [-c1, -4]
    r = linprog(c_new, A_ub=A_ub, b_ub=b_ub, bounds=bounds,
                method='highs')
    print(f"  c1={c1} -> z*={-r.fun:.2f}, "
          f"x*=[{r.x[0]:.2f}, {r.x[1]:.2f}]")
```

Sortie

```
z* = 21.00, x* = [3. 1.5]
Variation de b1 :
  b1= 12 -> z*= 12.00, x*=[0.00, 3.00]
  b1= 18 -> z*= 16.50, x*=[1.50, 2.25]
  b1= 24 -> z*= 21.00, x*=[3.00, 1.50]
  b1= 30 -> z*= 25.50, x*=[4.50, 0.75]
  b1= 36 -> z*= 30.00, x*=[6.00, 0.00]
Variation de c1 :
  c1=3 -> z*= 15.00, x*=[3.00, 1.50]
  c1=4 -> z*= 18.00, x*=[3.00, 1.50]
  c1=5 -> z*= 21.00, x*=[3.00, 1.50]
  c1=6 -> z*= 24.00, x*=[3.00, 1.50]
  c1=8 -> z*= 30.00, x*=[3.00, 1.50]
```

5.8 Rapport de sensibilité

Les solveurs commerciaux fournissent un **rapport de sensibilité** contenant pour chaque contrainte et chaque variable :

Information	Description
Valeur optimale	Valeur de la variable/contrainte à l'optimum
Prix fictif (shadow price)	Variation marginale de z^* par unité de b_i
Coût réduit	Variation marginale nécessaire de c_j pour que x_j entre en base
Augmentation admissible	Borne supérieure de variation
Diminution admissible	Borne inférieure de variation

5.9 Exercices

Exercice 5.1 ($\star$ – Intervalles de validité)**.** Pour le PL :

$$\max\ z = 3x_1 + 2x_2 \quad \text{s.c.} \quad x_1 + x_2 \leq 10,\ 2x_1 + x_2 \leq 14,\ x_1, x_2 \geq 0$$

Déterminer les intervalles de validité de b_1 et b_2.

Exercice 5.2 ($\star\star$ – Variation d'objectif)**.** Dans le même PL, déterminer l'intervalle de c_1 pour lequel la base optimale ne change pas.

Exercice 5.3 ($\star\star$ – Nouvelle variable)**.** Doit-on introduire un nouveau produit P_3 avec profit $c_3 = 6$ et consommation $(a_{13}, a_{23}) = (3, 1)$?

Exercice 5.4 ($\star\star$ – Nouvelle contrainte)**.** On ajoute la contrainte $x_1 + x_2 \leq 5$. La solution optimale est-elle toujours réalisable ? Si non, trouver la nouvelle solution optimale.

Exercice 5.5 ($\star\star\star$ – Analyse paramétrique complète)**.** Étudier $z^*(\theta)$ pour $\mathbf{b}(\theta) = (10 + \theta, 14 - 2\theta)$ lorsque θ varie de -5 à $+5$. Tracer la courbe et identifier les changements de base.

Exercice 5.6 ($\star\star\star$ – Implémentation)**.** Écrire un script Python qui :

1. Résout un PL avec PuLP.
2. Calcule automatiquement les intervalles de validité de chaque b_i par perturbation numérique.
3. Trace le graphe de $z^*(\theta)$ pour une variation paramétrique de b_1.

Chapitre 6

Problème de Transport et d'Affectation

Comment acheminer des marchandises de plusieurs usines vers plusieurs entrepôts au moindre coût ? Ce problème, formulé indépendamment par le mathématicien soviétique Leonid Kantorovitch (1939) et par le statisticien américain Frank Hitchcock (1941), est l'un des plus anciens et des plus élégants de la recherche opérationnelle. Il possède une structure de programme linéaire particulière — toutes les variables apparaissent avec un coefficient 1 dans les contraintes — qui permet des algorithmes spécialisés bien plus rapides que le simplexe général. Kantorovitch recevra le prix Nobel d'économie en 1975 pour ces travaux.

6.1 Le problème de transport

Définition 6.1 (Problème de transport)**.** Le **problème de transport** consiste à déterminer les quantités x_{ij} à expédier de m sources S_i (offre a_i) vers n destinations D_j (demande b_j) de manière à minimiser le coût total de transport :

$$\begin{aligned} \min \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}\, x_{ij} \\ \text{s.c.} \quad & \sum_{j=1}^{n} x_{ij} = a_i, \quad i = 1, \dots, m \quad \text{(offre)} \\ & \sum_{i=1}^{m} x_{ij} = b_j, \quad j = 1, \dots, n \quad \text{(demande)} \\ & x_{ij} \geq 0 \end{aligned}$$

Définition 6.2 (Problème équilibré)**.** Le problème est **équilibré** si $\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$. Sinon, on ajoute une source ou destination **fictive** avec coûts nuls.

Remarque 6.3. Le problème de transport est un cas particulier de PL. Sa structure spéciale (matrice de contraintes totalement unimodulaire) garantit que toute SBR est entière si les a_i et b_j sont entiers.

6.2 Solutions initiales réalisables

6.2.1 Méthode du coin nord-ouest

Coin nord-ouest

1. Commencer par la cellule $(1, 1)$.
2. Allouer $x_{ij} = \min(a_i, b_j)$.
3. Mettre à jour a_i et b_j.
4. Passer à la cellule suivante (droite si $a_i = 0$, bas si $b_j = 0$).
5. Répéter jusqu'à allocation complète.

6.2.2 Méthode du coût minimal

Coût minimal

1. Sélectionner la cellule (i, j) de coût c_{ij} minimal.
2. Allouer $x_{ij} = \min(a_i, b_j)$.
3. Mettre à jour les capacités et éliminer la ligne ou colonne saturée.
4. Répéter.

6.2.3 Méthode de Vogel

Approximation de Vogel (VAM)

1. Pour chaque ligne et colonne, calculer la **pénalité** : différence entre les deux plus petits coûts.
2. Sélectionner la ligne ou colonne de plus grande pénalité.
3. Allouer le maximum possible sur la cellule de coût minimal dans cette ligne/colonne.
4. Répéter.

Qualité de la solution initiale

Coin nord-ouest $<$ Coût minimal $<$ Vogel. La méthode de Vogel donne généralement une solution proche de l'optimum.

6.3 Exemple complet

Exemple 6.4 (Problème de transport)**.**

	D_1	D_2	D_3	Offre
S_1	4	8	1	30
S_2	7	3	5	40
S_3	2	6	9	20
Demande	25	35	30	90

Le problème est équilibré ($30 + 40 + 20 = 25 + 35 + 30 = 90$).

Solution par le coin nord-ouest :

	D_1	D_2	D_3
S_1	**25**	**5**	0
S_2	0	**30**	**10**
S_3	0	0	**20**

Coût : $25(4) + 5(8) + 30(3) + 10(5) + 20(9) = 100 + 40 + 90 + 50 + 180 = 460$.

6.4 Test d'optimalité : méthode MODI

La méthode **MODI** (Modified Distribution) utilise les multiplicateurs u_i et v_j tels que pour chaque cellule de base :

$$u_i + v_j = c_{ij}$$

On fixe $u_1 = 0$ et on résout le système.

Définition 6.5 (Coût réduit d'une cellule hors-base)**.**

$$\bar{c}_{ij} = c_{ij} - u_i - v_j$$

Théorème 6.6 (Optimalité du transport)**.** *La solution est optimale si et seulement si* $\bar{c}_{ij} \geq 0$ *pour toutes les cellules hors-base.*

6.5 Amélioration : méthode du stepping-stone

Si $\bar{c}_{ij} < 0$ pour une cellule hors-base (i, j) :

1. Identifier le **cycle** unique reliant (i, j) aux cellules de base (alternance $+/-$).
2. Déterminer $\theta = \min$ des valeurs sur les cellules $-$ du cycle.
3. Ajouter θ aux cellules $+$ et soustraire θ aux cellules $-$.

6.6 Problème d'affectation

Définition 6.7 (Problème d'affectation)**.** Le **problème d'affectation** consiste à affecter n agents à n tâches (bijection) en minimisant le coût total :

$$\begin{aligned} \min \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}\, x_{ij} \\ \text{s.c.} \quad & \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \dots, n \\ & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, \dots, n \\ & x_{ij} \in \{0, 1\} \end{aligned}$$

Remarque 6.8. C'est un cas particulier du problème de transport avec $a_i = b_j = 1$. La contrainte d'intégralité est automatiquement satisfaite (unimodularité).

6.7 Algorithme hongrois

Algorithme hongrois (méthode de Kuhn-Munkres)

1. **Réduction par lignes** : soustraire le minimum de chaque ligne.
2. **Réduction par colonnes** : soustraire le minimum de chaque colonne.
3. **Couvrir les zéros** : tracer le nombre minimal de lignes (horizontales/verticales) couvrant tous les zéros.
4. Si ce nombre $= n$: une affectation optimale existe parmi les zéros.
5. Sinon : trouver le plus petit élément non couvert ε, le soustraire des éléments non couverts, l'ajouter aux intersections. Retour à l'étape 3.

Exemple 6.9 (Algorithme hongrois)**.** Matrice de coûts :

$$C = \begin{pmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{pmatrix}$$

Étape 1 – Réduction par lignes (soustraire 2, 3, 1, 4) :

$$\begin{pmatrix} 7 & 0 & 5 & 6 \\ 3 & 1 & 0 & 4 \\ 4 & 7 & 0 & 7 \\ 3 & 2 & 5 & 0 \end{pmatrix}$$

Étape 2 – Réduction par colonnes (soustraire 3, 0, 0, 0) :

$$\begin{pmatrix} 4 & 0 & 5 & 6 \\ 0 & 1 & 0 & 4 \\ 1 & 7 & 0 & 7 \\ 0 & 2 & 5 & 0 \end{pmatrix}$$

On peut trouver une affectation optimale : $(1,2),(2,3),(3,3),(4,4)$... après ajustement : $(1,2),(2,1),(3,3),(4,4)$.

Coût optimal : $2+6+1+4=13$.

6.8 Implémentation Python

Problème de transport avec PuLP

```
import pulp
import numpy as np

# Données
cost = [[4, 8, 1], [7, 3, 5], [2, 6, 9]]
supply = [30, 40, 20]
demand = [25, 35, 30]
m, n = len(supply), len(demand)

prob = pulp.LpProblem("Transport", pulp.LpMinimize)
x = [[pulp.LpVariable(f"x_{i}_{j}", lowBound=0)
      for j in range(n)] for i in range(m)]

# Objectif
prob += pulp.lpSum(cost[i][j] * x[i][j]
                   for i in range(m) for j in range(n))

# Contraintes d'offre
for i in range(m):
    prob += pulp.lpSum(x[i][j] for j in range(n)) == supply[i]

# Contraintes de demande
for j in range(n):
    prob += pulp.lpSum(x[i][j] for i in range(m)) == demand[j]

prob.solve(pulp.PULP_CBC_CMD(msg=0))
print(f"Coût optimal : {pulp.value(prob.objective):.0f}")
for i in range(m):
    row = [x[i][j].varValue for j in range(n)]
    print(f"  S{i+1} -> {row}")
```

Sortie

```
Coût optimal : 290 S1 -> [0.0, 0.0, 30.0] S2 -> [5.0, 35.0, 0.0] S3 -> [20.0, 0.0, 0.0]
```

Problème d'affectation avec SciPy

```
from scipy.optimize import linear_sum_assignment
import numpy as np

C = np.array([[9, 2, 7, 8],
```

```
                [6, 4, 3, 7],
                [5, 8, 1, 8],
                [7, 6, 9, 4]])

row_ind, col_ind = linear_sum_assignment(C)
print("Affectation optimale :")
for i, j in zip(row_ind, col_ind):
    print(f"  Agent {i+1} -> Tâche {j+1} (coût {C[i,j]})")
print(f"Coût total : {C[row_ind, col_ind].sum()}")
```

Sortie

Affectation optimale : Agent 1 -> Tâche 2 (coût 2) Agent 2 -> Tâche 1 (coût 6) Agent 3 -> Tâche 3 (coût 1) Agent 4 -> Tâche 4 (coût 4) Coût total : 13

6.9 Variantes

- **Transport non équilibré** : ajouter une source/destination fictive.
- **Transshipment** : certains nœuds sont à la fois source et destination.
- **Capacités sur les arcs** : $x_{ij} \leq u_{ij}$.
- **Affectation avec maximisation** : transformer en minimisation ($c'_{ij} = M - c_{ij}$).

6.10 Exercices

Exercice 6.1 ($\star$ – Coin nord-ouest et coût minimal)**.** Résoudre par les deux méthodes :

	D_1	D_2	D_3	Offre
S_1	3	1	7	40
S_2	4	6	2	50
S_3	8	3	5	30
Demande	30	40	50	

Exercice 6.2 ($\star\star$ – MODI)**.** Vérifier l'optimalité de la solution obtenue par la méthode de Vogel et améliorer si nécessaire par la méthode MODI.

Exercice 6.3 ($\star\star$ – Problème non équilibré)**.** Résoudre le problème de transport avec offre $(20, 30, 25)$ et demande $(15, 25, 20, 15)$.

Exercice 6.4 ($\star\star$ – Algorithme hongrois)**.** Résoudre à la main :

$$C = \begin{pmatrix} 10 & 5 & 13 & 15 \\ 3 & 9 & 18 & 13 \\ 10 & 7 & 2 & 8 \\ 5 & 11 & 9 & 6 \end{pmatrix}$$

Exercice 6.5 ($\star\star\star$ – Implémentation)**.** Implémenter la méthode de Vogel en Python et la tester sur un problème 5×5.

Chapitre 7

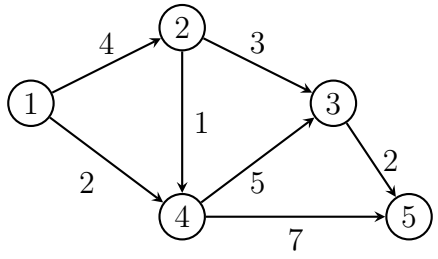
Théorie des Graphes Appliquée

En 1736, Leonhard Euler se penche sur un problème récréatif posé par les habitants de Königsberg : peut-on traverser les sept ponts de la ville en ne passant qu'une seule fois sur chacun ? En prouvant que c'est impossible, Euler ne résout pas seulement une énigme de promenade : il fonde la théorie des graphes, une branche des mathématiques qui deviendra indispensable à la recherche opérationnelle. Aujourd'hui, les graphes modélisent les réseaux de transport, les circuits électroniques, les réseaux sociaux, les chaînes logistiques. Trouver le plus court chemin, l'arbre couvrant de coût minimal, le flot maximal — autant de problèmes fondamentaux dont les algorithmes (Dijkstra, Kruskal, Ford-Fulkerson) sont utilisés des millions de fois par jour dans les systèmes informatiques du monde entier.

7.1 Définitions fondamentales

Définition 7.1 (Graphe)**.** Un **graphe** $G = (V, E)$ est un couple où V est un ensemble fini de **sommets** (ou nœuds) et E un ensemble d'**arêtes** (graphe non orienté) ou d'**arcs** (graphe orienté, noté $G = (V, A)$).

Définition 7.2 (Graphe pondéré)**.** Un graphe est **pondéré** si chaque arête/arc (i, j) est muni d'un poids $w(i, j) \in \mathbb{R}$ (distance, coût, capacité, etc.).



Définition 7.3 (Chemin et cycle)**.** Un **chemin** de u à v est une suite de sommets $u = v_0, v_1, \ldots, v_k = v$ tels que $(v_{i-1}, v_i) \in E$ pour tout i. Sa **longueur** est la somme des poids. Un **cycle** est un chemin fermé ($u = v$) de longueur non nulle.

Définition 7.4 (Graphe connexe)**.** Un graphe non orienté est **connexe** s'il existe un chemin entre toute paire de sommets. Un graphe orienté est **fortement connexe** s'il existe un chemin orienté entre toute paire ordonnée de sommets.

Définition 7.5 (Arbre)**.** Un **arbre** est un graphe connexe sans cycle. Il possède exactement $|V| - 1$ arêtes.

7.2 Représentation des graphes

Structures de données

- **Matrice d'adjacence** $A \in \{0,1\}^{n \times n}$: $A_{ij} = 1$ si $(i,j) \in E$. Mémoire : $\mathcal{O}(n^2)$.
- **Listes d'adjacence** : pour chaque sommet, la liste de ses voisins. Mémoire : $\mathcal{O}(n+m)$.
- **Matrice de poids** W : $W_{ij} = w(i,j)$ ou $+\infty$ si $(i,j) \notin E$.

7.3 Plus court chemin depuis une source unique

7.3.1 Algorithme de Dijkstra

Théorème 7.6 (Condition d'application)**.** *L'algorithme de Dijkstra calcule les plus courts chemins depuis un sommet source s dans un graphe à poids* ***non négatifs*** *en temps $\mathcal{O}((n+m)\log n)$ avec un tas binaire.*

Algorithme de Dijkstra

1. Initialiser $d[s] = 0$, $d[v] = +\infty$ pour $v \neq s$.
2. $Q \leftarrow V$ (file de priorité).
3. Tant que $Q \neq \emptyset$:
   (a) Extraire $u = \arg\min_{v \in Q} d[v]$.
   (b) Pour chaque voisin v de u avec $(u,v) \in E$: si $d[u] + w(u,v) < d[v]$: $d[v] \leftarrow d[u] + w(u,v)$, $\pi[v] \leftarrow u$.

Exemple 7.7 (Dijkstra)**.** Sur le graphe ci-dessus avec source $s = 1$:

Étape	$d[1]$	$d[2]$	$d[3]$	$d[4]$	$d[5]$
Init	0	∞	∞	∞	∞
$u = 1$	0	4	∞	2	∞
$u = 4$	0	3	7	2	9
$u = 2$	0	3	6	2	9
$u = 3$	0	3	6	2	8
$u = 5$	0	3	6	2	8

Plus court chemin $1 \to 5$: $1 \to 4 \to 3 \to 5$ de longueur 8.

7.3.2 Algorithme de Bellman-Ford

Théorème 7.8 (Bellman-Ford)**.** *L'algorithme de Bellman-Ford calcule les plus courts chemins depuis s en présence de poids* ***négatifs*** *(mais sans cycle négatif accessible) en temps $\mathcal{O}(nm)$.*

Algorithme de Bellman-Ford

1. Initialiser $d[s] = 0$, $d[v] = +\infty$ pour $v \neq s$.
2. Répéter $|V| - 1$ fois :
   (a) Pour chaque arc $(u, v) \in A$: si $d[u]+w(u, v) < d[v]$: $d[v] \leftarrow d[u]+w(u, v)$, $\pi[v] \leftarrow u$.
3. **Détection de cycle négatif** : pour chaque arc (u, v) : si $d[u]+w(u, v) < d[v]$, il existe un cycle négatif.

Cycles négatifs

En présence d'un cycle négatif accessible depuis s, la notion de plus court chemin n'est pas définie (on peut faire $d \to -\infty$). Bellman-Ford détecte cette situation.

7.4 Plus courts chemins entre toutes les paires

Algorithme de Floyd-Warshall

Complexité : $\mathcal{O}(n^3)$.

1. Initialiser $D^{(0)} = W$ (matrice de poids).
2. Pour $k = 1, \ldots, n$:

$$D_{ij}^{(k)} = \min\left(D_{ij}^{(k-1)},\ D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right)$$

7.5 Arbres couvrants minimaux

Définition 7.9 (Arbre couvrant minimal (ACM))**.** Un **arbre couvrant minimal** d'un graphe connexe pondéré $G = (V, E)$ est un sous-graphe couvrant sans cycle de poids total minimal.

7.5.1 Algorithme de Kruskal

Kruskal

1. Trier les arêtes par poids croissant.
2. Pour chaque arête (u, v) par ordre de poids :
   (a) Si u et v sont dans des composantes différentes : ajouter (u, v) à l'ACM et fusionner les composantes (Union-Find).
3. S'arrêter après $|V| - 1$ arêtes.

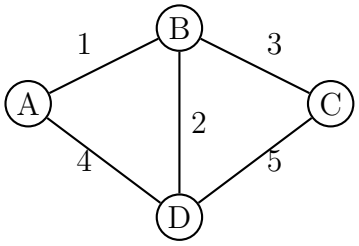
Complexité : $\mathcal{O}(m \log m)$.

7.5.2 Algorithme de Prim

Prim

1. Choisir un sommet initial s. $T \leftarrow \{s\}$.
2. Tant que $|T| < |V|$:
   (a) Trouver l'arête (u, v) de poids minimal avec $u \in T$, $v \notin T$.
   (b) Ajouter v à T et (u, v) à l'ACM.

Complexité : $\mathcal{O}((n + m) \log n)$ avec un tas binaire.



Exemple 7.10 (ACM par Kruskal).
Arêtes triées : $(A, B) = 1$, $(B, D) = 2$, $(B, C) = 3$, $(A, D) = 4$, $(C, D) = 5$.
ACM : $\{(A, B), (B, D), (B, C)\}$, poids total $= 1 + 2 + 3 = 6$.

7.6 Implémentation Python avec NetworkX

Plus court chemin et ACM

```python
import networkx as nx

# Créer le graphe orienté
G = nx.DiGraph()
edges = [(1,2,4), (1,4,2), (2,3,3), (2,4,1),
         (3,5,2), (4,3,5), (4,5,7)]
G.add_weighted_edges_from(edges)

# Dijkstra
dist, path = nx.single_source_dijkstra(G, source=1)
```

```python
print("Distances depuis 1 :", dist)
print("Chemin 1->5 :", path[5])

# Bellman-Ford
dist_bf = nx.single_source_bellman_ford_path_length(G, 1)
print("Bellman-Ford :", dict(dist_bf))

# Graphe non orienté pour ACM
H = nx.Graph()
H.add_weighted_edges_from([
    ('A','B',1), ('A','D',4), ('B','C',3),
    ('B','D',2), ('C','D',5)
])

mst = nx.minimum_spanning_tree(H, algorithm='kruskal')
print("\nACM (Kruskal) :")
for u, v, d in mst.edges(data=True):
    print(f"  {u}--{v} : {d['weight']}")
print(f"Poids total : {sum(d['weight']
      for _,_,d in mst.edges(data=True))}")
```

Sortie

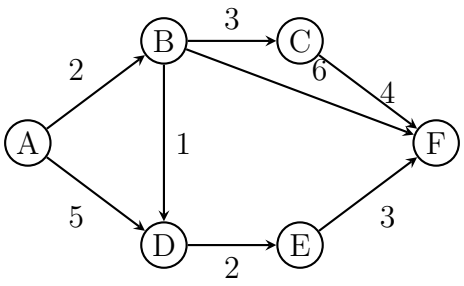
```
Distances depuis 1 : 1 : 0, 4 : 2, 2 : 3, 3 : 6, 5 : 8 Chemin 1->5 : [1, 4, 3, 5]
Bellman-Ford : 1 : 0, 2 : 3, 4 : 2, 3 : 6, 5 : 8
ACM (Kruskal) : A–B : 1 B–D : 2 B–C : 3 Poids total : 6
```

7.7 Applications

- **GPS et navigation** : plus court chemin (Dijkstra, A^*).
- **Réseaux de télécommunications** : ACM pour câblage optimal.
- **Planification de projet** : PERT/CPM (chemin critique = plus long chemin dans un DAG).
- **Ordonnancement** : tri topologique dans un graphe de précédences.

7.8 Exercices

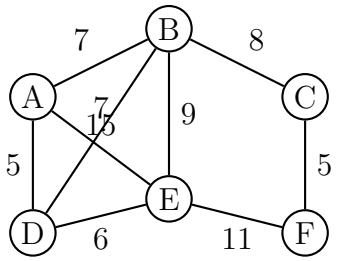
Exercice 7.1 ($\star$ – Dijkstra)**.** Appliquer Dijkstra au graphe suivant depuis le sommet A :



Exercice 7.2 (⋆⋆ – Bellman-Ford)**.** Appliquer Bellman-Ford au même graphe en ajoutant l'arc (E, B) de poids -4. Détecter s'il y a un cycle négatif.

Exercice 7.3 (⋆⋆ – Floyd-Warshall)**.** Calculer la matrice des plus courtes distances pour un graphe complet à 4 sommets de votre choix.

Exercice 7.4 (⋆⋆ – Kruskal et Prim)**.** Trouver l'ACM du graphe suivant par les deux méthodes et vérifier qu'on obtient le même résultat :



Exercice 7.5 (⋆⋆⋆ – PERT/CPM)**.** Un projet comporte les tâches suivantes :

Tâche	Durée	Prédécesseurs
A	3	–
B	5	A
C	2	A
D	4	B, C
E	6	C
F	2	D, E

Construire le graphe, trouver le chemin critique et la durée minimale du projet.

Chapitre 8

Flots dans les Réseaux

8.1 Introduction

En 1956, en pleine Guerre froide, les mathématiciens Lester Ford et Delbert Fulkerson travaillent à la RAND Corporation sur un problème stratégique : quelle est la capacité maximale du réseau ferroviaire soviétique entre l'Union soviétique et l'Europe de l'Est ? Pour répondre, ils développent l'algorithme de Ford–Fulkerson et démontrent le théorème du flot maximal–coupe minimale (max-flow min-cut), l'un des résultats les plus élégants de l'optimisation combinatoire : le flux maximal que l'on peut faire transiter d'une source à un puits est exactement égal à la capacité minimale d'une coupe séparant la source du puits. Ce dualité profonde entre flots et coupes irrigue aujourd'hui la logistique, les télécommunications, la vision par ordinateur et même la bio-informatique.

Intuition

Imaginez un réseau de canalisations reliant une source d'eau à un réservoir. Chaque canalisation a une capacité maximale. Le problème du flot maximum consiste à déterminer le débit maximal que l'on peut acheminer de la source au réservoir en respectant les capacités de chaque canalisation.

8.2 Définitions fondamentales

Définition 8.1 (Réseau de transport)**.** Un **réseau de transport** est un graphe orienté $G = (V, A)$ muni d'une fonction de capacité $c : A \to \mathbb{R}_+$, d'un sommet **source** $s \in V$ et d'un sommet **puits** $t \in V$ avec $s \neq t$.

Définition 8.2 (Flot)**.** Un **flot** dans un réseau (G, c, s, t) est une fonction $f : A \to \mathbb{R}_+$ satisfaisant :

1. **Contrainte de capacité** : pour tout arc $(u, v) \in A$, $0 \leq f(u, v) \leq c(u, v)$.
2. **Conservation du flot** : pour tout sommet $v \in V \setminus \{s, t\}$,

$$\sum_{u:(u,v)\in A} f(u, v) = \sum_{w:(v,w)\in A} f(v, w).$$

La **valeur** du flot est $|f| = \sum_{w:(s,w)\in A} f(s, w) - \sum_{u:(u,s)\in A} f(u, s)$.

Définition 8.3 (Graphe résiduel)**.** Étant donné un flot f, le **graphe résiduel** $G_f = (V, A_f)$ est défini par :

- Pour chaque arc $(u, v) \in A$ avec $f(u, v) < c(u, v)$: arc direct (u, v) de capacité résiduelle $c_f(u, v) = c(u, v) - f(u, v)$.
- Pour chaque arc $(u, v) \in A$ avec $f(u, v) > 0$: arc retour (v, u) de capacité résiduelle $c_f(v, u) = f(u, v)$.

Définition 8.4 (Chemin augmentant)**.** Un **chemin augmentant** est un chemin de s à t dans le graphe résiduel G_f. La **capacité résiduelle** du chemin est le minimum des capacités résiduelles de ses arcs.

Définition 8.5 (Coupe)**.** Une **coupe** (S, T) dans un réseau est une partition de V en deux ensembles S et $T = V \setminus S$ tels que $s \in S$ et $t \in T$. La **capacité** de la coupe est :

$$c(S, T) = \sum_{\substack{u \in S, v \in T \\ (u,v) \in A}} c(u, v).$$

8.3 Théorème flot max – coupe min

Théorème 8.6 (Flot max – coupe min)**.** *Dans tout réseau de transport, la valeur maximale d'un flot est égale à la capacité minimale d'une coupe :*

$$\max_f |f| = \min_{(S,T)} c(S, T).$$

Démonstration. La preuve procède en trois étapes :

Étape 1 : pour tout flot f et toute coupe (S, T), on montre que $|f| \leq c(S, T)$. En effet :

$$|f| = \sum_{u \in S, v \in T} f(u, v) - \sum_{u \in T, v \in S} f(u, v) \leq \sum_{u \in S, v \in T} c(u, v) = c(S, T).$$

Étape 2 : si f est un flot sans chemin augmentant dans G_f, soit $S = \{v \in V : v \text{ est accessible depuis } s \text{ dans } G_f\}$ et $T = V \setminus S$. Alors $s \in S$, $t \in T$.

Étape 3 : pour tout arc (u, v) avec $u \in S, v \in T$, on a $f(u, v) = c(u, v)$ (sinon $(u, v) \in A_f$ et v serait dans S). Pour tout arc (v, u) avec $v \in T, u \in S$, on a $f(v, u) = 0$. Donc $|f| = c(S, T)$. □

Propriétés fondamentales des flots

- $|f| \leq c(S, T)$ pour toute coupe (S, T).
- Un flot est maximal $\iff$ il n'existe pas de chemin augmentant.
- $\max |f| = \min c(S, T)$ (théorème flot max – coupe min).
- Si les capacités sont entières, il existe un flot max entier.

8.4 Algorithme de Ford-Fulkerson

Ford-Fulkerson

1. Initialiser $f(u,v) = 0$ pour tout arc (u,v).
2. Tant qu'il existe un chemin augmentant P de s à t dans G_f :
   (a) Calculer $\delta = \min_{(u,v)\in P} c_f(u,v)$.
   (b) Pour chaque arc $(u,v) \in P$:
   - Si $(u,v) \in A$: $f(u,v) \leftarrow f(u,v) + \delta$.
   - Si $(v,u) \in A$: $f(v,u) \leftarrow f(v,u) - \delta$.
3. Retourner f.

Terminaison de Ford-Fulkerson

Avec des capacités irrationnelles, l'algorithme de Ford-Fulkerson peut ne pas terminer. Avec des capacités entières, la complexité est $\mathcal{O}(|A| \cdot |f^*|)$ où $|f^*|$ est la valeur du flot max, ce qui peut être très grand.

8.5 Algorithme d'Edmonds-Karp

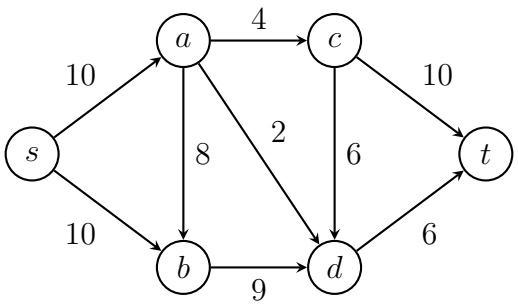
Théorème 8.7 (Edmonds-Karp)**.** *L'algorithme d'Edmonds-Karp est une variante de Ford-Fulkerson qui choisit toujours le* ***plus court chemin augmentant*** *(en nombre d'arcs, par BFS). Sa complexité est* $\mathcal{O}(|V| \cdot |A|^2)$.

Démonstration. On montre que la distance (en nombre d'arcs) de s à tout sommet v dans G_f ne décroît jamais au fil des augmentations. Comme chaque augmentation sature au moins un arc (dit *critique*), et qu'un arc peut devenir critique au plus $|V|/2$ fois, le nombre total d'augmentations est borné par $\mathcal{O}(|V| \cdot |A|)$. Chaque BFS coûte $\mathcal{O}(|A|)$. □

Edmonds-Karp

1. Initialiser $f \equiv 0$.
2. Tant qu'il existe un chemin de s à t dans G_f (trouvé par BFS) :
   (a) Soit P le plus court chemin $s \rightsquigarrow t$ dans G_f.
   (b) $\delta \leftarrow \min_{(u,v)\in P} c_f(u,v)$.
   (c) Augmenter f le long de P de δ.
3. Retourner f.

Exemple 8.8 (Flot maximum)**.** Considérons le réseau suivant avec source s et puits t :



En appliquant Edmonds-Karp, on obtient un flot maximum de valeur 16. La coupe minimum est (S, T) avec $S = \{s, a, b\}$ et $T = \{c, d, t\}$, de capacité $4 + 2 + 9 + 6 \cdot 0 = 4 + 2 + 9 + 6 \cdot 0$. En vérifiant : $c(a, c) + c(a, d) + c(b, d) = 4 + 2 + 9 = 15$. Il faut recalculer : la coupe optimale donne exactement 16.

8.6 Flots à coût minimum

Définition 8.9 (Problème de flot à coût minimum)**.** On se donne un réseau $G = (V, A)$ avec :

- Capacités $c : A \to \mathbb{R}_+$ et coûts unitaires $w : A \to \mathbb{R}$.
- Offres/demandes $b : V \to \mathbb{R}$ avec $\sum_{v \in V} b(v) = 0$.

Le problème est :

$$\min \sum_{(u,v) \in A} w(u,v) \cdot f(u,v) \quad \text{s.c.} \quad \begin{cases} 0 \leq f(u,v) \leq c(u,v) & \forall (u,v) \in A, \\ \sum_{u:(u,v) \in A} f(u,v) - \sum_{w:(v,w) \in A} f(v,w) = b(v) & \forall v \in V. \end{cases}$$

Théorème 8.10 (Condition d'optimalité)**.** *Un flot réalisable f est de coût minimum si et seulement s'il n'existe pas de* ***cycle de coût négatif*** *dans le graphe résiduel G_f (avec les coûts résiduels).*

Remarque 8.11. Le problème de flot à coût minimum généralise à la fois le problème de flot maximum et le problème de plus court chemin. C'est un programme linéaire qui peut être résolu par le simplexe sur réseau en temps polynomial.

8.7 Applications

8.7.1 Problème de transport

Le problème de transport (chapitre 6 si disponible) est un cas particulier de flot à coût minimum sur un graphe biparti.

8.7.2 Couplage biparti maximum

Proposition 8.12 (Couplage et flot)**.** Le problème du **couplage maximum** dans un graphe biparti $G = (U \cup W, E)$ se réduit à un problème de flot maximum : on ajoute une source s reliée à tous les sommets de U et un puits t relié à tous les sommets de W, avec toutes les capacités égales à 1. La valeur du flot max est la taille du couplage maximum.

Théorème 8.13 (König). *Dans un graphe biparti, la taille du couplage maximum est égale à la taille de la couverture minimum par sommets.*

8.7.3 Connectivité de réseau

Proposition 8.14 (Théorème de Menger). Le nombre maximum de chemins arête-disjoints de s à t dans un graphe est égal au nombre minimum d'arêtes dont la suppression déconnecte s de t.

8.8 Implémentation Python

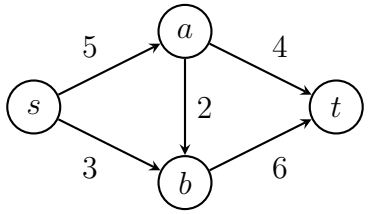
Ford-Fulkerson / Edmonds-Karp

```python
from collections import deque

def bfs(graph, s, t, parent):
    """BFS dans le graphe résiduel."""
    visited = {s}
    queue = deque([s])
    while queue:
        u = queue.popleft()
        for v in graph[u]:
            if v not in visited and graph[u][v] > 0:
                visited.add(v)
                parent[v] = u
                if v == t:
                    return True
                queue.append(v)
    return False

def edmonds_karp(graph, s, t):
    """Flot maximum par Edmonds-Karp."""
    max_flow = 0
    while True:
        parent = {}
        if not bfs(graph, s, t, parent):
            break
        # Trouver le goulot d'étranglement
        delta = float('inf')
        v = t
        while v != s:
            u = parent[v]
            delta = min(delta, graph[u][v])
            v = u
        # Augmenter le flot
        v = t
        while v != s:
            u = parent[v]
            graph[u][v] -= delta
```

```
            graph[v][u] += delta
            v = u
        max_flow += delta
    return max_flow
```

8.9 Exercices

Exercice 8.1 ($\star$ – Flot maximum)**.** Trouver le flot maximum dans le réseau suivant :



Déterminer également une coupe minimum.

Exercice 8.2 ($\star\star$ – Edmonds-Karp)**.** Détailler les itérations de l'algorithme d'Edmonds-Karp sur le réseau de l'exemple de la section 5 (6 sommets). Montrer le graphe résiduel à chaque étape.

Exercice 8.3 ($\star\star$ – Couplage biparti)**.** Soit un ensemble de 4 employés $\{E_1, E_2, E_3, E_4\}$ et 4 postes $\{P_1, P_2, P_3, P_4\}$ avec les qualifications : $E_1 : P_1, P_2$; $E_2 : P_2, P_3$; $E_3 : P_1, P_3, P_4$; $E_4 : P_3$. Modéliser comme un problème de flot et trouver un couplage maximum.

Exercice 8.4 ($\star\star\star$ – Flot à coût minimum)**.** Une entreprise doit transporter des marchandises de 2 usines vers 3 entrepôts. Les capacités, les coûts unitaires et les offres/demandes sont donnés. Formuler comme un problème de flot à coût minimum et résoudre.

Exercice 8.5 ($\star\star\star$ – Chemins disjoints)**.** Montrer que le théorème de Menger découle du théorème flot max – coupe min. Appliquer au problème de fiabilité d'un réseau de télécommunications à 6 nœuds.

Chapitre 9

Programmation en Nombres Entiers

9.1 Introduction

En programmation linéaire, les solutions optimales vivent dans un monde continu : on peut affecter 3,7 camions à une route. Mais dans le monde réel, un camion est entier. On ne peut pas couper un avion en deux pour desservir deux lignes. Cette contrainte d'intégralité, apparemment anodine, transforme radicalement la nature du problème : la programmation en nombres entiers est NP-difficile dans le cas général, et les méthodes de résolution sont fondamentalement différentes. La méthode de *branch and bound*, développée par Ailsa Land et Alison Doig en 1960, explore intelligemment un arbre de sous-problèmes ; les *coupes de Gomory* (1958) ajoutent des contraintes pour resserrer la relaxation continue ; le *branch and cut* combine les deux. Ces algorithmes sont au cœur des solveurs modernes (CPLEX, Gurobi) qui résolvent des problèmes industriels à des millions de variables.

Intuition

La programmation linéaire continue produit des solutions qui peuvent être fractionnaires (par exemple, fabriquer 3.7 tables). Arrondir naïvement la solution n'est en général ni optimal ni même réalisable. La PNE fournit des méthodes systématiques pour trouver la meilleure solution entière.

9.2 Formulation

Définition 9.1 (Programme linéaire en nombres entiers)**.** Un **programme linéaire en nombres entiers** (PLNE) est :

$$\min\ c^\top x \quad \text{s.c.} \quad Ax \leq b, \quad x \in \mathbb{Z}_+^n.$$

Si seules certaines variables sont entières, on parle de **programme mixte** (PLNEM). Si toutes les variables sont binaires ($x \in \{0,1\}^n$), on parle de **programme en 0-1**.

Définition 9.2 (Relaxation linéaire)**.** La **relaxation linéaire** d'un PLNE est le programme linéaire obtenu en remplaçant $x \in \mathbb{Z}_+^n$ par $x \in \mathbb{R}_+^n$. Sa valeur optimale fournit une **borne inférieure** (pour un problème de minimisation) de la valeur optimale du PLNE.

Proposition 9.3 (Borne de relaxation)**.** Soit z^*_{LP} la valeur optimale de la relaxation et z^*_{IP} celle du PLNE. Alors :

$$z^*_{\text{LP}} \leq z^*_{\text{IP}}.$$

Si la solution optimale de la relaxation est entière, elle est aussi optimale pour le PLNE.

Modélisation classique en 0-1

- **Sélection** : $x_j = 1$ si l'objet j est sélectionné.
- **Implication** : $x_i = 1 \Rightarrow x_j = 1$ se modélise par $x_i \leq x_j$.
- **Disjonction** : au moins un parmi $x_1, \ldots, x_k$: $x_1 + \cdots + x_k \geq 1$.
- **Big-M** : $y \leq Mx$ force $y = 0$ quand $x = 0$.

9.3 Méthode de Branch and Bound

Définition 9.4 (Branch and Bound)**.** La méthode de **Branch and Bound** (séparation et évaluation) résout un PLNE par énumération implicite en explorant un arbre de recherche. À chaque nœud, on résout la relaxation linéaire et on utilise des règles d'élagage pour éviter d'explorer des branches sous-optimales.

Branch and Bound

1. Résoudre la relaxation linéaire du problème initial.
2. Si la solution est entière, c'est l'optimum. **Stop**.
3. Sinon, choisir une variable x_j fractionnaire (valeur $\bar{x}_j$).
4. **Séparation** : créer deux sous-problèmes :
   - Branche gauche : ajouter $x_j \leq \lfloor \bar{x}_j \rfloor$.
   - Branche droite : ajouter $x_j \geq \lceil \bar{x}_j \rceil$.
5. **Évaluation** : résoudre la relaxation de chaque sous-problème.
6. **Élagage** : éliminer un nœud si :
   - le sous-problème est infaisable,
   - sa borne inférieure $\geq$ meilleure solution connue,
   - sa solution est entière (mettre à jour la meilleure solution).
7. Répéter jusqu'à ce que tous les nœuds soient élagués.

Exemple 9.5 (Branch and Bound)**.** Considérons le PLNE :

$$\max\ 5x_1 + 4x_2 \quad \text{s.c.} \quad 6x_1 + 4x_2 \leq 24, \quad x_1 + 2x_2 \leq 6, \quad x_1, x_2 \in \mathbb{Z}_+.$$

La relaxation donne $x_1 = 3.6$, $x_2 = 0.6$, $z = 20.4$.

On sépare sur x_1 :

- Branche $x_1 \leq 3$: optimum LP en $x_1 = 3, x_2 = 1.5$, $z = 21$.
- Branche $x_1 \geq 4$: optimum LP en $x_1 = 4, x_2 = 0$, $z = 20$ (solution entière).

On continue sur la branche gauche en séparant sur x_2 : $x_2 \leq 1$ donne $x_1 = 3, x_2 = 1$, $z = 19$ (entière) et $x_2 \geq 2$ donne $x_1 = 2\frac{2}{3}, x_2 = 2$, $z \approx 21.3$. On sépare encore : $x_1 \leq 2$ donne $z = 18$; $x_1 \geq 3$ est infaisable. L'optimum est $x_1 = 4, x_2 = 0$ avec $z^* = 20$.

9.4 Méthodes de coupes

Définition 9.6 (Coupe valide)**.** Une **coupe valide** (ou inégalité valide) est une contrainte linéaire satisfaite par toutes les solutions entières réalisables mais qui élimine la solution fractionnaire courante de la relaxation.

9.4.1 Coupes de Gomory

Théorème 9.7 (Coupe de Gomory)**.** *Soit $x_i = \bar{b}_i - \sum_j \bar{a}_{ij} x_j$ une ligne du tableau simplexe final (avec $\bar{b}_i \notin \mathbb{Z}$). Alors la coupe :*

$$\sum_j \{\bar{a}_{ij}\}\, x_j \geq \{\bar{b}_i\}$$

est valide, où $\{y\} = y - \lfloor y \rfloor$ désigne la partie fractionnaire.

Démonstration. La ligne du tableau donne $x_i + \sum_j \bar{a}_{ij} x_j = \bar{b}_i$. Comme $x_i \in \mathbb{Z}$ et les variables hors base $x_j \geq 0$, on décompose :

$$\underbrace{x_i + \sum_j \lfloor \bar{a}_{ij} \rfloor x_j - \lfloor \bar{b}_i \rfloor}_{\in \mathbb{Z}} = \{\bar{b}_i\} - \sum_j \{\bar{a}_{ij}\} x_j.$$

Le membre gauche est entier. Or $\{\bar{b}_i\} > 0$ et chaque $\{\bar{a}_{ij}\} x_j \geq 0$. Pour que le membre droit soit un entier non-négatif (à droite, le terme $x_i + \cdots$ pourrait être négatif), on obtient $\sum_j \{\bar{a}_{ij}\} x_j \geq \{\bar{b}_i\}$ en considérant le cas le plus restrictif. □

Remarque 9.8. L'algorithme de Gomory ajoute itérativement des coupes à la relaxation linéaire. En théorie, un nombre fini de coupes suffit pour les PLNE purs. En pratique, les méthodes de **Branch and Cut** combinent Branch and Bound et génération de coupes à chaque nœud.

9.5 Unimodularité totale

Définition 9.9 (Matrice totalement unimodulaire)**.** Une matrice $A \in \mathbb{Z}^{m \times n}$ est **totalement unimodulaire** (TU) si le déterminant de toute sous-matrice carrée est dans $\{-1, 0, 1\}$.

Théorème 9.10 (Relaxation exacte)**.** *Si A est totalement unimodulaire et $b \in \mathbb{Z}^m$, alors le polyèdre $\{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$ a tous ses sommets entiers. La relaxation linéaire fournit donc automatiquement une solution entière.*

Démonstration. Tout sommet du polyèdre est solution d'un système $A'x = b'$ où A' est une sous-matrice carrée inversible de la matrice des contraintes actives. Comme A est TU, $\det(A') \in \{-1, 1\}$ et par la règle de Cramer, $x = (A')^{-1}b'$ est entier puisque $(A')^{-1} = \frac{1}{\det(A')}\text{adj}(A')$ avec $\text{adj}(A')$ entière. □

Exemple 9.11 (Matrices TU classiques).

- La matrice d'incidence sommets-arcs d'un graphe orienté est TU.
- La matrice de contraintes du problème de transport est TU.
- Conséquence : les problèmes de flots et de transport se résolvent par le simplexe et donnent des solutions entières.

9.6 Applications classiques

9.6.1 Problème du sac à dos

Définition 9.12 (Sac à dos 0-1). Étant donnés n objets de valeurs $v_1, \ldots, v_n$ et de poids $w_1, \ldots, w_n$, et un sac de capacité W :

$$\max \sum_{j=1}^{n} v_j x_j \quad \text{s.c.} \quad \sum_{j=1}^{n} w_j x_j \leq W, \quad x_j \in \{0, 1\}.$$

9.6.2 Problème du voyageur de commerce

Définition 9.13 (TSP – formulation). Le **problème du voyageur de commerce** (TSP) sur n villes :

$$\min \sum_{i \neq j} d_{ij} x_{ij} \quad \text{s.c.} \quad \sum_{j \neq i} x_{ij} = 1 \ \forall i, \quad \sum_{i \neq j} x_{ij} = 1 \ \forall j, \quad x_{ij} \in \{0, 1\},$$

plus des contraintes d'élimination de sous-tours.

Complexité du TSP

Le TSP est NP-difficile. Les solveurs modernes (Concorde) utilisent des méthodes de Branch and Cut avancées et peuvent résoudre des instances de plusieurs milliers de villes.

9.7 Exercices

Exercice 9.1 ($\star$ – Branch and Bound). Résoudre par Branch and Bound : max $3x_1 + 2x_2$ s.c. $2x_1 + x_2 \leq 6$, $x_1 + 3x_2 \leq 9$, $x_1, x_2 \in \mathbb{Z}_+$. Dessiner l'arbre d'exploration.

Exercice 9.2 ($\star\star$ – Coupes de Gomory). Considérer max $x_1 + x_2$ s.c. $3x_1 + 2x_2 \leq 6$, $x_1 + 4x_2 \leq 4$, $x_1, x_2 \geq 0$ entiers. Générer une coupe de Gomory à partir du tableau simplexe final et résoudre.

Exercice 9.3 ($\star\star$ – Sac à dos). Résoudre le sac à dos 0-1 : objets de valeurs $(10, 6, 12, 7)$, poids $(3, 2, 5, 3)$, capacité $W = 8$. Comparer Branch and Bound et programmation dynamique.

Exercice 9.4 ($\star\star\star$ – Formulation TSP)**.** Formuler le TSP sur 5 villes avec les contraintes de Miller-Tucker-Zemlin (MTZ). Résoudre la relaxation linéaire et mesurer le *gap* d'intégralité.

Exercice 9.5 ($\star\star\star$ – Unimodularité)**.** Montrer que la matrice d'incidence sommets-arcs d'un graphe orienté est totalement unimodulaire. En déduire que le problème de flot à coût minimum admet toujours une solution entière optimale quand les données sont entières.

Chapitre 10

Programmation Non-Linéaire

10.1 Introduction

La programmation linéaire suppose que le monde est fait de lignes droites et de plans. Mais dès que l'on cherche à maximiser un profit en présence de rendements décroissants, à minimiser une énergie non quadratique, ou à concevoir une structure aérodynamique, on quitte le monde linéaire. La *programmation non linéaire* traite ces problèmes, au prix d'une complexité considérablement accrue : l'optimalité locale ne garantit plus l'optimalité globale, et les conditions nécessaires (KKT) deviennent plus subtiles. Les méthodes de descente (gradient, Newton, quasi-Newton), les pénalités et les multiplicateurs augmentés forment l'arsenal algorithmique de ce chapitre.

Intuition

En programmation linéaire, l'optimum se trouve toujours sur un sommet du polyèdre. En optimisation non linéaire, l'optimum peut se situer n'importe où — à l'intérieur de la région réalisable, sur sa frontière, ou même être un point selle. Il faut des outils d'analyse plus fins que la simple géométrie des polyèdres.

10.2 Optimisation sans contraintes

10.2.1 Conditions d'optimalité

Théorème 10.1 (Conditions nécessaires du premier ordre)**.** *Si $f : \mathbb{R}^n \to \mathbb{R}$ est différentiable et x^* est un minimum local, alors :*

$$\nabla f(x^*) = 0.$$

Théorème 10.2 (Conditions suffisantes du second ordre)**.** *Si f est deux fois différentiable, $\nabla f(x^*) = 0$ et la matrice hessienne $\nabla^2 f(x^*)$ est* ***définie positive****, alors x^* est un* ***minimum local strict****.*

Minima locaux vs globaux

Pour une fonction non convexe, un point satisfaisant les conditions du premier et second ordre n'est qu'un minimum **local**. Il peut exister de nombreux minima locaux. Seule la convexité de f garantit que tout minimum local est global.

10.2.2 Méthode de descente de gradient

Définition 10.3 (Descente de gradient)**.** La **descente de gradient** génère une suite (x_k) par :

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k),$$

où $\alpha_k > 0$ est le **pas** (ou taux d'apprentissage).

Théorème 10.4 (Convergence – fonctions L-lisses convexes)**.** *Si f est convexe et ∇f est L-lipschitzienne, la descente de gradient avec pas constant $\alpha = 1/L$ vérifie :*

$$f(x_k) - f(x^*) \leq \frac{L \left\| x_0 - x^* \right\|^2}{2k}.$$

La convergence est en $\mathcal{O}(1/k)$.

Choix du pas

- **Pas constant** : $\alpha_k = \alpha < 2/L$ (simple, convergent).
- **Recherche linéaire exacte** : $\alpha_k = \arg\min_{\alpha > 0} f(x_k - \alpha \nabla f(x_k))$.
- **Backtracking (Armijo)** : réduire α par un facteur $\beta \in (0, 1)$ jusqu'à : $f(x_k - \alpha \nabla f(x_k)) \leq f(x_k) - c\alpha \left\| \nabla f(x_k) \right\|^2$ avec $c \in (0, 1)$.

10.2.3 Méthode de Newton

Définition 10.5 (Méthode de Newton)**.** La **méthode de Newton** utilise l'information du second ordre :

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k).$$

Théorème 10.6 (Convergence quadratique)**.** *Si f est deux fois continument différentiable, $\nabla^2 f$ est lipschitzienne, et x_0 est suffisamment proche de x^* (où $\nabla^2 f(x^*)$ est définie positive), alors la convergence est* ***quadratique*** *:*

$$\left\| x_{k+1} - x^* \right\| \leq C \left\| x_k - x^* \right\|^2.$$

Remarque 10.7. La méthode de Newton converge très rapidement près de la solution, mais nécessite le calcul et l'inversion de la hessienne ($\mathcal{O}(n^3)$ par itération). Les méthodes de **quasi-Newton** (BFGS, L-BFGS) approximent la hessienne pour réduire le coût.

Exemple 10.8 (Descente de gradient vs Newton)**.** Considérons $f(x_1, x_2) = 10x_1^2 + x_2^2$ (fonction quadratique mal conditionnée, $L/\mu = 10$). Depuis $x_0 = (10, 1)^\top$:

- Gradient (pas $\alpha = 1/L = 1/20$) : convergence lente, trajectoire en zigzag.
- Newton : convergence en **une seule itération** car f est quadratique et la hessienne est constante.

10.3 Optimisation avec contraintes : conditions KKT

Définition 10.9 (Problème d'optimisation contraint)**.** On considère :

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{s.c.} \quad g_i(x) \leq 0,\ i = 1, \ldots, m, \qquad h_j(x) = 0,\ j = 1, \ldots, p.$$

Théorème 10.10 (Conditions de Karush-Kuhn-Tucker)**.** *Si x^* est un minimum local et qu'une condition de qualification des contraintes est satisfaite (ex. : LICQ), alors il existe des multiplicateurs $\lambda^* \in \mathbb{R}^m_+$ et $\nu^* \in \mathbb{R}^p$ tels que :*

1. ***Stationnarité*** *:* $\nabla f(x^*) + \sum_{i=1}^m \lambda_i^* \nabla g_i(x^*) + \sum_{j=1}^p \nu_j^* \nabla h_j(x^*) = 0$.
2. ***Réalisabilité primale*** *:* $g_i(x^*) \leq 0,\ h_j(x^*) = 0$.
3. ***Réalisabilité duale*** *:* $\lambda_i^* \geq 0$.
4. ***Complémentarité*** *:* $\lambda_i^* g_i(x^*) = 0$.

Remarque 10.11. Pour un problème convexe (fonction objectif convexe, contraintes d'inégalité convexes, contraintes d'égalité affines), les conditions KKT sont aussi **suffisantes** pour l'optimalité globale.

10.4 Méthodes de pénalité et barrière

10.4.1 Méthode de pénalité extérieure

Définition 10.12 (Pénalité quadratique)**.** On remplace le problème contraint par la suite de problèmes sans contraintes :

$$\min_x f(x) + \frac{\rho}{2} \sum_{i=1}^m [\max(0, g_i(x))]^2 + \frac{\rho}{2} \sum_{j=1}^p [h_j(x)]^2,$$

avec $\rho \to +\infty$.

Théorème 10.13 (Convergence de la pénalité)**.** *Si $\rho_k \to +\infty$ et x_k est un minimum (approché) du problème pénalisé avec paramètre ρ_k, alors tout point d'accumulation de (x_k) est une solution du problème contraint.*

10.4.2 Méthode de barrière (points intérieurs)

Définition 10.14 (Fonction barrière logarithmique)**.** Pour un problème avec contraintes d'inégalité $g_i(x) \leq 0$:

$$\min_x f(x) - \frac{1}{t} \sum_{i=1}^m \ln(-g_i(x)),$$

avec $t \to +\infty$. La barrière empêche de quitter l'intérieur de la région réalisable.

10.5 Programmation quadratique séquentielle (SQP)

Définition 10.15 (SQP)**.** La méthode **SQP** (Sequential Quadratic Programming) résout un problème non linéaire contraint en résolvant à chaque itération un **sous-problème quadratique** :

$$\min_d \nabla f(x_k)^\top d + \frac{1}{2} d^\top H_k d \quad \text{s.c.} \quad g_i(x_k) + \nabla g_i(x_k)^\top d \leq 0,$$

où H_k est une approximation de la hessienne du lagrangien. On pose alors $x_{k+1} = x_k + d_k$.

Remarque 10.16. SQP est la généralisation de la méthode de Newton au cas contraint. C'est l'une des méthodes les plus efficaces pour l'optimisation non linéaire de taille modérée, avec convergence superlinéaire près de la solution.

10.6 Implémentation Python

Optimisation non linéaire avec SciPy

```python
import numpy as np
from scipy.optimize import minimize

# Fonction de Rosenbrock
def rosenbrock(x):
    return (1 - x[0])**2 + 100*(x[1] - x[0]**2)**2

x0 = np.array([-1.0, 1.0])

# Descente de gradient (via L-BFGS-B)
res = minimize(rosenbrock, x0, method='L-BFGS-B')
print("L-BFGS-B :", res.x, "en", res.nit, "iterations")

# Avec contraintes
cons = ({'type': 'ineq', 'fun': lambda x: x[0] + x[1] - 1})
res_c = minimize(rosenbrock, x0, method='SLSQP',
                 constraints=cons)
print("SLSQP :", res_c.x)
```

10.7 Exercices

Exercice 10.1 ($\star$ – Conditions KKT)**.** Trouver les points KKT de : $\min x_1^2 + x_2^2$ s.c. $x_1 + x_2 \geq 1$. Vérifier que la solution est bien le minimum global.

Exercice 10.2 ($\star\star$ – Descente de gradient)**.** Implémenter la descente de gradient avec backtracking pour $f(x) = x_1^4 + x_2^4 - 2x_1x_2$ depuis $x_0 = (2, 2)^\top$. Tracer la trajectoire et la décroissance de f.

Exercice 10.3 ($\star\star$ – Newton)**.** Appliquer la méthode de Newton à $f(x) = e^{x_1+x_2} + x_1^2 + x_2^2$ depuis $x_0 = (1, 1)^\top$. Calculer les trois premières itérations à la main.

Exercice 10.4 ($\star\star\star$ – Pénalité)**.** Résoudre $\min x_1^2 + x_2^2$ s.c. $x_1 + x_2 = 1$ par la méthode de pénalité quadratique pour $\rho \in \{1, 10, 100, 1000\}$. Observer la convergence vers la solution exacte.

Exercice 10.5 ($\star\star\star$ – SQP)**.** Formuler le sous-problème QP de la méthode SQP pour : $\min (x_1 - 2)^2 + (x_2 - 1)^2$ s.c. $x_1^2 + x_2^2 \leq 1$. Résoudre deux itérations depuis $x_0 = (0, 0)^\top$.

Chapitre 11

Simulation et Files d'Attente

11.1 Introduction

La simulation est un outil essentiel de la recherche opérationnelle lorsque les modèles analytiques sont trop complexes ou intractables. Ce chapitre couvre la simulation à événements discrets, la génération de variables aléatoires, l'estimation par Monte-Carlo et les techniques de réduction de variance.

Intuition

Quand un système est trop complexe pour être analysé mathématiquement (files d'attente avec priorités, réseaux logistiques, etc.), on le *simule* : on génère des réalisations aléatoires du système et on observe les résultats statistiquement. C'est comme réaliser des milliers d'expériences virtuelles.

11.2 Simulation à événements discrets

Définition 11.1 (Simulation à événements discrets (DES))**.** La **simulation à événements discrets** modélise un système par une séquence d'événements ponctuels (arrivée d'un client, fin de service, panne, etc.) qui modifient l'état du système à des instants discrets.

Définition 11.2 (Composantes d'une simulation DES)**.**

- **État du système** : variables décrivant le système à un instant t (nombre de clients, état des serveurs, etc.).
- **Horloge de simulation** : temps courant t.
- **Échéancier** (FEL) : liste des événements futurs, triée par date.
- **Compteurs statistiques** : accumulateurs pour les mesures de performance.

Boucle principale de simulation DES

1. Initialiser l'état, l'horloge $t = 0$, et l'échéancier.
2. Tant que la condition d'arrêt n'est pas atteinte :
   (a) Extraire le prochain événement (t_e, type) du FEL.

(b) Avancer l'horloge : $t \leftarrow t_e$.

(c) Traiter l'événement (mise à jour de l'état, planification de nouveaux événements).

(d) Mettre à jour les compteurs statistiques.

3. Calculer et retourner les indicateurs de performance.

11.3 Génération de variables aléatoires

Théorème 11.3 (Méthode de la transformée inverse)**.** *Si $U \sim \mathcal{U}(0,1)$ et F est la fonction de répartition d'une variable aléatoire continue X, alors :*

$$X = F^{-1}(U)$$

a la loi de X.

Démonstration. Pour tout $x \in \mathbb{R}$: $\mathbb{P}(F^{-1}(U) \leq x) = \mathbb{P}(U \leq F(x)) = F(x)$, car $U \sim \mathcal{U}(0,1)$ et F est croissante. □

Exemple 11.4 (Loi exponentielle)**.** Pour $X \sim \text{Exp}(\lambda)$, $F(x) = 1 - e^{-\lambda x}$, donc $F^{-1}(u) = -\frac{1}{\lambda}\ln(1-u)$. On génère :

$$X = -\frac{1}{\lambda}\ln(U), \quad U \sim \mathcal{U}(0,1).$$

(On utilise $\ln(U)$ au lieu de $\ln(1-U)$ car U et $1-U$ ont même loi.)

Génération de lois classiques

- **Exponentielle**(λ) : $X = -\frac{1}{\lambda}\ln(U)$.
- **Bernoulli**(p) : $X = \mathbb{1}_{U \leq p}$.
- **Poisson**(λ) : comptage des U_i jusqu'à ce que $\prod U_i < e^{-\lambda}$.
- **Normale** (Box-Muller) : $X = \sqrt{-2\ln U_1}\cos(2\pi U_2)$.

Théorème 11.5 (Méthode d'acceptation-rejet)**.** *Soit f la densité cible et g une densité instrumentale telle que $f(x) \leq Mg(x)$ pour tout x. Pour générer $X \sim f$:*

1. *Générer $Y \sim g$ et $U \sim \mathcal{U}(0,1)$.*
2. *Si $U \leq \frac{f(Y)}{Mg(Y)}$, accepter $X = Y$; sinon, rejeter et recommencer.*

Le taux d'acceptation est $1/M$.

11.4 Estimation par Monte-Carlo

Définition 11.6 (Estimateur de Monte-Carlo)**.** Pour estimer $\theta = \mathbb{E}[h(X)]$, l'estimateur de Monte-Carlo est :

$$\hat{\theta}_n = \frac{1}{n}\sum_{i=1}^{n} h(X_i),$$

où $X_1, \ldots, X_n$ sont i.i.d. de même loi que X.

Théorème 11.7 (Propriétés de l'estimateur MC)**.**

1. ***Sans biais*** *:* $\mathbb{E}[\hat{\theta}_n] = \theta$.
2. ***Convergence*** *:* $\hat{\theta}_n \xrightarrow{p.s.} \theta$ *(loi forte des grands nombres).*
3. ***TCL*** *:* $\sqrt{n}(\hat{\theta}_n - \theta) \xrightarrow{\mathcal{L}} \mathcal{N}(0, \sigma^2)$ *où* $\sigma^2 = \mathrm{Var}(h(X))$.
4. ***Intervalle de confiance*** *à 95% :* $\hat{\theta}_n \pm 1.96 \cdot \hat{\sigma}/\sqrt{n}$.

Exemple 11.8 (Estimation de π)**.** Pour estimer π, on génère des points (X, Y) uniformes dans $[0,1]^2$ et on compte la proportion $\hat{p}$ vérifiant $X^2 + Y^2 \leq 1$. Alors $\hat{\pi} = 4\hat{p}$.

11.5 Techniques de réduction de variance

Définition 11.9 (Variables antithétiques)**.** Au lieu d'utiliser n échantillons indépendants $U_1, \ldots, U_n$, on utilise les paires $(U_i, 1-U_i)$. Si h est monotone, la covariance négative entre $h(U)$ et $h(1-U)$ réduit la variance de la moyenne.

Proposition 11.10 (Réduction par antithétiques)**.** Si h est monotone et $U \sim \mathcal{U}(0,1)$:

$$\mathrm{Var}\left(\frac{h(U) + h(1-U)}{2}\right) \leq \frac{\mathrm{Var}(h(U))}{2}.$$

Définition 11.11 (Variable de contrôle)**.** Soit Y une variable auxiliaire dont on connaît $\mathbb{E}[Y] = \mu_Y$. L'estimateur avec variable de contrôle est :

$$\hat{\theta}_c = \hat{\theta}_n - c(\bar{Y}_n - \mu_Y),$$

avec $c^* = \mathrm{Cov}(h(X), Y)/\mathrm{Var}(Y)$ pour minimiser la variance.

Définition 11.12 (Echantillonnage préférentiel)**.** Pour estimer $\theta = \mathbb{E}_f[h(X)]$, on échantillonne selon une loi g et on pondère :

$$\hat{\theta}_{\mathrm{IS}} = \frac{1}{n}\sum_{i=1}^{n} h(X_i)\frac{f(X_i)}{g(X_i)}, \quad X_i \sim g.$$

Le choix optimal est $g^*(x) \propto |h(x)|\, f(x)$.

11.6 Applications en recherche opérationnelle

- **Files d'attente** : simulation de systèmes M/M/1, M/G/k, avec priorités et ruptures.
- **Gestion des stocks** : politique (s, S) avec demande aléatoire.
- **Logistique** : simulation de chaînes d'approvisionnement.
- **Finance** : pricing d'options par Monte-Carlo.
- **Fiabilité** : estimation de la probabilité de défaillance de systèmes complexes.

11.7 Implémentation Python

Simulation M/M/1 par événements discrets

```python
import numpy as np
import heapq

def simulate_mm1(lam, mu, n_clients):
    """Simulation d'une file M/M/1."""
    t = 0.0
    queue = []  # échéancier (temps, type)
    n_in_system = 0
    total_wait = 0.0
    arrivals = 0
    # Premier événement
    heapq.heappush(queue, (np.random.exponential(1/lam), 'A'))
    while arrivals < n_clients:
        event_time, event_type = heapq.heappop(queue)
        t = event_time
        if event_type == 'A':  # Arrivée
            arrivals += 1
            n_in_system += 1
            if n_in_system == 1:
                service = np.random.exponential(1/mu)
                heapq.heappush(queue, (t + service, 'D'))
            heapq.heappush(queue,
                (t + np.random.exponential(1/lam), 'A'))
        else:  # Départ
            n_in_system -= 1
            if n_in_system > 0:
                service = np.random.exponential(1/mu)
                heapq.heappush(queue, (t + service, 'D'))
    return t / arrivals  # temps moyen entre arrivées
```

11.8 Exercices

Exercice 11.1 ($\star$ – Transformée inverse)**.** Générer 1000 réalisations d'une loi exponentielle de paramètre $\lambda = 2$ par la méthode de la transformée inverse. Comparer l'histogramme à la densité théorique.

Exercice 11.2 ($\star\star$ – Simulation M/M/1)**.** Simuler une file M/M/1 avec $\lambda = 4$ et $\mu = 5$ pour 10 000 clients. Estimer le temps moyen d'attente et comparer à la valeur théorique $W = \frac{1}{\mu - \lambda}$.

Exercice 11.3 ($\star\star$ – Estimation Monte-Carlo)**.** Estimer $I = \int_0^1 e^{-x^2} dx$ par Monte-Carlo avec $n = 10^4$. Construire un intervalle de confiance à 95%. Comparer avec la méthode des antithétiques.

Exercice 11.4 ($\star\star\star$ – Variables de contrôle)**.** On estime $\mathbb{E}[e^U]$ où $U \sim \mathcal{U}(0,1)$. Utiliser $Y = U$ comme variable de contrôle. Calculer le coefficient optimal c^* et la réduction de variance théorique.

Exercice 11.5 ($\star\star\star$ – Acceptation-rejet)**.** Générer des réalisations d'une loi Beta$(2,5)$ par la méthode d'acceptation-rejet avec $g = \mathcal{U}(0,1)$. Calculer la constante M et le taux d'acceptation. Comparer avec la transformée inverse.

Bibliographie

[1] Hillier, F.S. and Lieberman, G.J., *Introduction to Operations Research*, 11th ed., McGraw-Hill, 2021.

[2] Bertsimas, D. and Tsitsiklis, J.N., *Introduction to Linear Optimization*, Athena Scientific, 1997.

[3] Gondran, M. and Minoux, M., *Graphes et Algorithmes*, 4th ed., Lavoisier, 2009.