Reconstruction de matrices Origine-Destination à partir de flux de voyageurs

Compte rendu de stage de L3

Garance Malnoë
encadrée par Sylvain Faure et Bertrand Maury
Institut de Mathématiques d'Orsay

Mai - Juin 2024

Table des matières

1 Introduction

La compréhension des différents flux de voyageurs est un enjeu important pour optimiser les réseaux routiers et de transports en commun, ainsi que pour le développement de nouvelles infrastructures adaptées aux besoins des usagers.

Les mathématiques, et notamment la modélisation par des graphes, offrent des outils utiles pour aborder cette problématique. Ce rapport se concentre sur la reconstruction des trajets des voyageurs à partir des données de montées et descentes de lignes linéaires.

Nous commençons par définir les notions de données eulériennes et lagrangiennes et la problématique du passage de l'un à l'autre. Nous nous intéressons par la suite à la description de l'ensemble des scénarios lagrangiens correspondants à des données eulériennes pour un réseau linéaire, notamment par le biais de deux algorithmes. Ensuite, nous regardons deux méthodes d'optimisation de la fonction entropie permettant de sélectionner un scénario lagrangien que nous mettons alors en œuvre en Python. Enfin, nous nous intéressons à deux méthodes probabilistes de sélection de scénario issues d'une problématique similaire.

2 Passage de données eulériennes à lagrangiennes

Cette partie s'appuie sur le poly du cours de modélisation de Bertrand Maury [Maury(2024)].

Les termes "eulérien" et "lagrangien" désignent deux approches différentes de l'analyse des phénomènes de transport et de mouvement. Bien qu'ils soient couramment utilisés dans le domaine de la mécanique des fluides, ils peuvent plus généralement s'appliquer à la description du mouvement de toute entité évoluant dans un espace (physique ou plus abstrait). Ils permettent de décrire la manière dont sont recueillies les données, mais aussi l'approche utilisée pour élaborer les modèles décrivant ces phénomènes de mouvement.

2.1 Données lagrangiennes

La description lagrangienne consiste à suivre les entités à partir de leur position d'origine puis tout au long de leur mouvement.

Si l'on considère, par exemple, un ensemble de N particules évoluant dans l'espace physique $\mathbb{R}^d$, la collection de leurs positions au cours du temps $t \mapsto x(t) = (x_i(t))_{i\in[\![1;N]\!]} \in \mathbb{R}^d$ est lagrangienne : on suit chaque particule de manière individuelle.

Par la suite, nous nous intéresserons aux déplacements d'individus sur un réseau de transports en commun. Pour étudier ces déplacements, on peut récupérer les données GPS des voyageurs à partir de leur téléphone. Bien-sûr, la loi ne nous permet pas de stocker l'identité de la personne suivie, mais un identifiant d'anonymisation nous per-

met d'associer plusieurs positions successives à un seul individu : ce sont des données lagrangiennes.

2.2 Données eulériennes

La description eulérienne ne se base pas sur l'observation de chaque entité, mais sur l'observation de zones de l'espace dans lesquelles le mouvement s'effectue. On regarde "ce qu'il se passe" dans chaque zone sans tenir compte de l'identité des entités qui se trouvent dans la zone.

Dans le cadre d'une étude des déplacements de voyageurs sur un territoire, la mesure du nombre de passages dans les gares au cours du temps ou du nombre de voitures sur un tronçon de route au cours du temps sont des données de nature eulérienne.

2.3 Passage de l'un à l'autre

Lorsque l'on possède des données de nature lagrangienne ou eulérienne, il peut être pertinent de se demander s'il est possible d'en changer la nature.

Il est souvent possible de passer de la vision lagrangienne à la vision eulérienne. En effet, reprenons l'exemple de la modélisation des trajectoires de N particules dans $\mathbb{R}^d$ avec les positions au cours du temps $t \mapsto x(t) = (x_i(t))_{i\in[\![1;N]\!]}$ comme données lagrangiennes. On introduit la notion de masse ponctuelle : à une particule située en un point x de l'espace, on associe la mesure δ_x qui est une application de l'ensemble des parties de l'espace dans $\mathbb{R}_+$ (ou plus simplement $\mathbb{N}$ si on se limite à compter des entités). Pour $A \subset \mathbb{R}^d$, on dit que $\delta_x(A)$ vaut 1 si $x \in A$, 0 sinon. Si deux particules se trouvent au même endroit, on applique une règle de sommation : $\delta_x + \delta_x = 2\delta_x$. On peut alors faire la somme des masses ponctuelles pour chacune des N particules pour obtenir une mesure qui décrit l'ensemble des positions des particules au temps t :

$$\mu_t = \sum_{i=1}^{N} \delta_{x_i(t)}.$$

Cette identité exprime que, pour tout ensemble $A \subset \mathbb{R}^d$, $\mu_t(A)$ est le nombre d'entités qui se trouvent dans la zone A à l'instant t : on a obtenu une donnée de nature eulérienne.

Bien que μ_t soit définie à partir des $x_i(t)$, l'information contenue dans μ est dégradée par rapport à la collection (x_i) puisque l'identité de chaque particule a été perdue dans la somme : on ne sait plus qui est où. Plus précisément, si les positions sont distinctes deux à deux, une même mesure μ_t correspond à un grand nombre de scénarios possibles pour les entités. Formellement, pour toute permutation $\varphi \in S_N$, on a :

$$\sum_{i=1}^{N} \delta_{x_i(t)} = \sum_{i=1}^{N} \delta_{x_{\varphi(i)}(t)}.$$

Ce manque d'information empêche souvent le passage du point de vue eulérien au

point de vue lagrangien. C'est un problème mal posé : plusieurs scénarios lagrangiens correspondent aux données eulériennes et il n'y a pas *a priori* un scénario qui soit plus plausible que les autres, à moins de rajouter des hypothèses.

Une autre différence importante entre les deux points de vue se trouve au niveau de la récolte des données. Reprenons le cas de déplacements au sein d'un territoire donné par exemple. Pour récupérer des données eulériennes, on peut, par exemple, installer des capteurs infrarouges aux portes des trains circulant sur le territoire pour compter les montées et les descentes des voyageurs ou on peut compter le nombre de voitures sur les tronçons de route à l'aide des caméras. Pour la récupération de données lagrangiennes, on peut suivre les usagers tout au long de leur déplacement à l'aide des données GPS de leur téléphone ou bien mettre en place une enquête pour demander directement aux gens quel parcours ils ont empruntés comme cela avait été fait à Rennes en 2023, mais ces deux dernières options demandent des dispositifs plus conséquents. Dans la réalité, il est souvent plus simple de récupérer des données eulériennes que des données lagrangiennes.

Ainsi, dans la suite de ce rapport, nous nous intéresserons au passage du point de vue eulérien vers le point de vue lagrangien pour la modélisation des déplacements sur un réseau de transport en commun linéaire.

3 Réseau linéaire : exemple d'une ligne de bus

3.1 Cadre théorique

Nous nous intéressons ici à une ligne de bus avec $N+1$ arrêts numérotés de 0 à N et nous regardons seulement le sens croissant des arrêts. Nous supposons que nous avons pu récupérer des données eulériennes sur les déplacements des usagers de la ligne : à chaque arrêt $i \in [\![0; N]\!]$, on a compté le nombre de montées μ_i, le nombre de descentes ν_i et le nombre de personnes à voyager entre l'arrêt i et l'arrêt $i+1$, ρ_i. On note alors $\mu, \nu \in \mathbb{R}^{N+1}$ et $\rho \in \mathbb{R}^N$ les vecteurs associés et on considère que $\nu_0 = \mu_N = 0$. Enfin, on introduit la quantité $b \in \mathbb{R}^{N+1}$ où $b_i = \mu_i - \nu_i$ le bilan de montée-descente de l'arrêt i.

L'objectif est de retrouver les déplacements de chaque usager à partir de μ, ν et ρ. On souhaite savoir combien de personnes sont montées à l'arrêt i pour descendre à l'arrêt $j = \gamma_{ij}$. On note $\gamma = (\gamma_{ij})_{ij}$ le plan origine-destination. On définit également le plan origine-destination normalisé $\tilde{\gamma}$ par :

$$\tilde{\gamma}_{ij} = \frac{\gamma_{ij}}{\mu_i} \qquad i \leq N, \quad i \leq j \leq N.$$

On note Γ l'ensemble des plans Origine-Destination γ qui vérifient les données de montées et des descentes, μ et ν :

$$\Gamma = \{\gamma \quad | \quad \sum_{i=0}^{N} \gamma_{ij} = \nu_j \ \text{ et } \sum_{j=0}^{N} \gamma_{ij} = \mu_i\}.$$

Ainsi défini, $\gamma_i = (\gamma_{ij})_j$ est une loi de probabilité supportée par $[\![i+1; N]\!]$ et qui décrit la distribution des destinations des voyageurs montés à l'arrêt i.

Tout d'abord, on peut s'intéresser aux liens entre les différentes quantités eulériennes : ρ, ν, μ et b.

Proposition :

$$\forall i \in [\![0; N-1]\!], \quad \rho_i = \sum_{j=0}^{i} bj \ .$$

Démonstration :
On procède par récurrence.
- Initialisation : au départ de la première station, on a :

$$\rho_0 = \mu_0 = \mu_0 - \nu_0 = b_0$$

- Hérédité : supposons l'hypothèse vraie pour un $i \in [\![0; N-1]\!]$. Montrons qu'elle est vraie à l'étape $i+1$:
ρ_{i+1} = les voyageurs entre i et $i+1$ + les voyageurs montants en $i+1$ - les voyageurs descendants en $i+1 = \rho_i + \mu_{i+1} - \nu_{i+1} = \sum_{j=0}^{i+1} b_j$.

Proposition : On fixe $\rho_{-1} = \rho_N = 0$. On a alors :

$$\forall i \in [\![0; N]\!], \quad b_i = \rho_i - \rho_{i-1}$$

Démonstration :
b_i = la différence entre les voyageurs partant de i et ceux partis de $i-1 = \rho_i - \rho_{i-1}$.

On a également une relation entre γ, μ et b sous la forme $G\mu = b$ où G est une matrice qui dépend du plan normalisé $\tilde{\gamma}$.

Proposition :

$$\begin{pmatrix} \sum_{j=1}^{N} \tilde{\gamma}_{0j} & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ -\tilde{\gamma}_{01} & \sum_{j=2}^{N} \tilde{\gamma}_{1j} & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & \ddots & 0 & \cdots & \cdots & \vdots \\ -\tilde{\gamma}_{0i} & \cdots & -\tilde{\gamma}_{i-1i} & \sum_{j=i+1}^{N} \tilde{\gamma}_{ij} & 0 & \cdots & 0 \\ \vdots & & & \ddots & \ddots & \cdots & \vdots \\ -\tilde{\gamma}_{0N-1} & \cdots & \cdots & \cdots & \cdots & \tilde{\gamma}_{N-1N} & 0 \\ -\tilde{\gamma}_{0N} & \cdots & \cdots & \cdots & \cdots & -\tilde{\gamma}_{N-1N} & 0 \end{pmatrix} \begin{pmatrix} \mu_0 \\ \vdots \\ \vdots \\ \mu_i \\ \vdots \\ \mu_{N-1} \\ \mu_N \end{pmatrix} = \begin{pmatrix} b_0 \\ \vdots \\ \vdots \\ b_i \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix}$$

Démonstration :

$$b_i = \mu_i - \nu_i = \sum_{j=i+1}^{N} \gamma_{ij} - \sum_{j=0}^{i-1} \gamma_{ji} = \sum_{j=i+1}^{N} \tilde{\gamma}_{ij} \times \mu_i - \sum_{j=0}^{i-1} \tilde{\gamma}_{ji} \times \mu_j = (\sum_{j=i+1}^{N} \tilde{\gamma}_{ij}) \times \mu_i - \sum_{j=0}^{i-1} \tilde{\gamma}_{ji} \times \mu_j.$$

Mais avec nos données eulériennes, on ne connait pas précisément γ ni $\tilde{\gamma}$, néanmoins nous avons plusieurs informations. On se concentrera sur γ :

- On étudie la ligne de bus dans un seul sens (ici dans le sens des indices croissants). Donc pour tout $i \in [\![0; N]\!]$ et pour tout $j < i$, $\gamma_{ij} = 0$. La partie inférieure gauche de la matrice est nulle.

- Un usager ne peut pas monter et descendre à la même station, la diagonale est donc nulle.

- Pour toute colonne d'indice j, $\sum_{i=0}^{N} \gamma_{ij} = \nu_j$. On peut en déduire que $\gamma_{01} = \nu_1$.

- Pour toute ligne d'indice i, $\sum_{j=0}^{N} \gamma_{ij} = \mu_i$. On peut en déduire que $\gamma_{N-1N} = \mu_N$.

On a donc une matrice de la forme suivante :

$$\gamma = \left(\begin{array}{cccccc|c}
0 & \nu_1 & \gamma_{02} & \cdots & \cdots & \gamma_{0N} & \sum \gamma_{0j} = \mu_0 \\
\vdots & \ddots & \gamma_{12} & \ddots & & \vdots & \sum \gamma_{1j} = \mu_1 \\
\vdots & & \ddots & \ddots & \ddots & \vdots & \sum \gamma_{ij} = \mu_j \\
\vdots & & & \ddots & \gamma_{N-2N-1} & \gamma_{N-2N} & \sum \gamma_{N-2j} = \mu_{N-2} \\
\vdots & & & & \ddots & \mu_{N-1} & \sum \gamma_{N-1j} = \mu_{N-1} \\
0 & \cdots & \cdots & \cdots & \cdots & 0 & \sum \gamma_{Nj} = \mu_N \\
\hline
\sum \gamma_{i0} = \nu_0 & \sum \gamma_{i1} = \nu_1 & \sum \gamma_{i2} = \nu_2 & \sum \gamma_{ij} = \nu_j & \sum \gamma_{iN-1} = \nu_{N-1} & \sum \gamma_{iN} = \nu_N &
\end{array}\right)$$

Pour trouver les valeurs des termes γ_{ij} restants, on peut mettre en place différents algorithmes, nous allons en voir deux.

3.2 Algorithme naïf

Le premier algorithme est un algorithme "naturel", mis en place pour se faire la main sur un exemple avec seulement quelques arrêts et de petites valeurs.

On suppose que les valeurs de μ, ν, ρ sont valides : il y a au moins une matrice γ qui correspond à ces données. On dira qu'une matrice est correctement remplie si la somme de lignes et la somme des colonnes correspond aux vecteurs de montées et de descentes.

Voici un exemple pour une ligne de bus à cinq arrêts (1) :

Algorithm 1 Algorithme de remplissage de matrice

Initialiser une matrice M remplie de 0.
if la matrice est correctement remplie **then**
La matrice nulle est la seule solution, c'est le cas où il n'y a aucun voyageur, arrêt.
else
Faire une hypothèse sur γ_{02}, commencer par $\gamma_{02} = 0$.
(*) On regarde ce que cela implique : y a-t-il des cases dont on est sûr de connaitre la valeur à partir des hypothèses sur la valeur de la somme des colonnes et des lignes ? On remplit alors ces cases avec les valeurs correspondantes en faisant attention aux contradictions (ex : une case doit valoir 2 pour que la condition sur la somme de la ligne soit vérifiée mais elle doit valoir 3 pour que la condition sur la colonne soit vérifiée).
if on a une contradiction **then**
On revient à l'hypothèse précédemment faite (et on enlève toutes les implications que cette hypothèse avait entrainées) et si on le peut, on augmente de 1 la valeur de l'hypothèse et on reprend à (*). Si ça n'est pas possible, on revient encore à l'hypothèse d'avant et ainsi de suite jusqu'à ce qu'on puisse augmenter une hypothèse de 1 pour reprendre à (*) et si on ne peut pas trouver de telle hypothèse, arrêt.
else
if la matrice est correctement remplie **then**
On sauvegarde la matrice
On revient à l'hypothèse précédemment faite (et on enlève toutes les implications que cette hypothèse avait entrainées) et si on le peut, on augmente de 1 la valeur de l'hypothèse et on reprend à (*). Si ça n'est pas possible, on revient encore à l'hypothèse d'avant et ainsi de suite jusqu'à ce qu'on puisse augmenter une hypothèse de 1 pour reprendre à (*) et si on ne peut pas trouver de telle hypothèse, arrêt.
else
On fait une nouvelle hypothèse sur la prochaine case non connue (on progresse ligne par ligne et colonne par colonne) et on reprend à (*).
end if
end if
end if

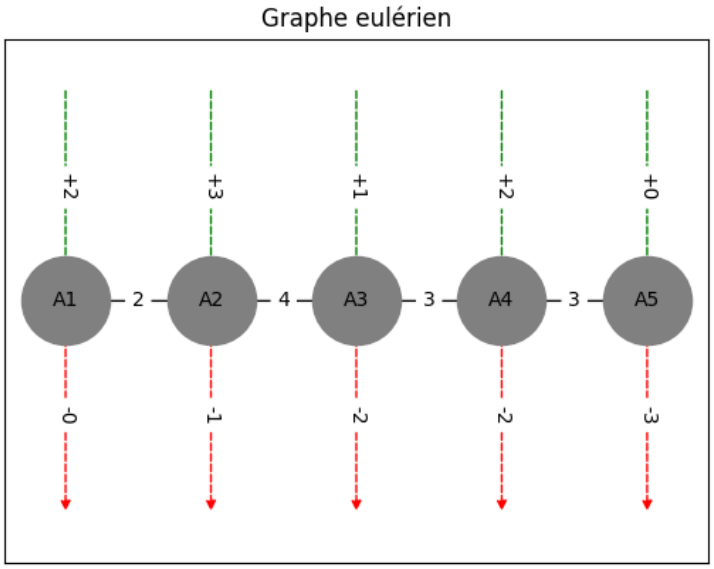

FIGURE 1 – Représentation ligne de bus à 5 arrêts à partir de données eulériennes

On prend : $\mu = (2,\ 3,\ 1,\ 2,\ 0)$, $\quad \nu = (0,\ 1,\ 2,\ 2,\ 3)$, $\quad \rho = (2,\ 4,\ 3,\ 3)$.

La première matrice γ trouvée par l'algorithme est la suivante :

$$\gamma_1 = \left(\begin{array}{ccccc|c} 0 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 2 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 2 & 2 & 3 & \end{array}\right)$$

— On suppose que γ_{02} vaut 0. Or, $\gamma_{02} + \gamma_{12} = 2$ donc γ_{12} vaut 2.

La prochaine case inconnue est 03.

— On suppose que γ_{03} vaut 0.
Or, $\gamma_{03} + \gamma_{04} = 1$ donc γ_{04} vaut 1.
Or, $\gamma_{04}+\gamma_{14}+\gamma_{24}+\gamma_{34} = 3$ donc $\gamma_{14} = \gamma_{24} = 0$.
Or, $\gamma_{12} + \gamma_{13} + \gamma_{14} = 3$ donc γ_{13} vaut 1.
Or, $\gamma_{03} + \gamma_{13} + \gamma_{23} = 2$ donc $\gamma_{23} = 1$

Voici les 4 autres matrices γ qui vérifient les conditions sur μ, ν et ρ :

$$\gamma_2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_3 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_4 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \gamma_5 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

3.3 Algorithme de recherche exhaustive

Le deuxième algorithme est un algorithme de recherche exhaustive (backtracking) qui est assez proche de l'algorithme naïf. Il consiste à explorer toutes les configurations possibles et à retenir celles qui satisfont les conditions. Voici le code Python de l'algorithme, on peut retrouver le code entier en annexe du rapport (8) ou sur GitHub.

```
def euler_to_lagrange(m, v):
    """
    :param m: une liste d'entiers, les montees a chaque arret
    :param v: une liste d'entiers, les descentes a chaque arret
    :return: une liste contenant toutes les matrices OD correspondant aux
        donnees de montees et de descentes
    """
    n = len(m)
    resultats = []

    # Fonction recursive pour trouver les grilles
    def backtrack(grille, m_rest, v_rest, ligne, col):
        # Cas 1 : On a rempli la derniere cellule. On verifie si la grille est
            valide
        #        Si elle est valide, on la sauvegarde dans les resultats.
        if ligne == n:
            if all(sum(grille[i]) == m[i] for i in range(n)) and all(
                    sum(grille[i][j] for i in range(n)) == v[j] for j in
                        range(n)):
                resultats.append([row[:] for row in grille])
            return

        # Cas 2 : On a rempli la derniere colonne de la ligne actuelle. On
            passe a la ligne suivante
        if col == n:
            # On appelle la fonction backtrack pour continuer a remplir la
                matrice
            # en se placant sur la ligne +1 et sur la premiere colonne
            backtrack(grille, m_rest, v_rest, ligne + 1, 0)
            return

        # Cas 3 : On est sur la diagonale ou la partie inferieure gauche, la
            valeur doit etre 0.
        if ligne >= col:
```

```
            grille[ligne][col] = 0 # On met le coef a 0
            backtrack(grille, m_rest, v_rest, ligne, col + 1) # On continue a
                remplir la grille.

        # Autre cas : On appelle la fonction backtrack pour chaque valeur que
            peut prendre la cellule
        else:
            # On definit la valeur maximale que peut prendre la cellule a
                partir de m_rest et v_rest.
            max_val = min(m_rest[ligne], v_rest[col])
            for val in range(max_val + 1):
                grille[ligne][col] = val
                # On modifie les valeurs de m_rest et v_rest pour remplir les
                    cellules restantes
                # Et respecter les conditions sur les sommes de lignes et de
                    colonnes
                m_rest[ligne] -= val
                v_rest[col] -= val

                # On appelle la fonction backtrack recursive pour remplir la
                    cellule suivante
                backtrack(grille, m_rest, v_rest, ligne, col + 1)

                # On retablie les valeurs de m_rest et v_rest pour tester la
                    nouvelle possibilite
                m_rest[ligne] += val
                v_rest[col] += val
                grille[ligne][col] = 0

    # On initialise une grille vide
    grille_vide = [[0] * n for _ in range(n)]

    # On appelle la fonction recursive backtrack pour remplir la grille et
        tester les possibilites.
    backtrack(grille_vide, m.copy(), v.copy(), 0, 0)

    return resultats
```

Pour le réseau de bus à 5 arrêts (1), on retrouve bien les mêmes cinq matrices Origine-Destination qu'avec l'algorithme naïf.

On a également pu tester l'algorithme sur d'autres petites lignes de bus. Un autre exemple avec 6 arrêts (2) et des valeurs un peu plus grandes :

$$\mu = (5,\ 4,\ 6,\ 3,\ 1,\ 0), \quad \nu = (0,\ 2,\ 4,\ 3,\ 5,\ 5)$$

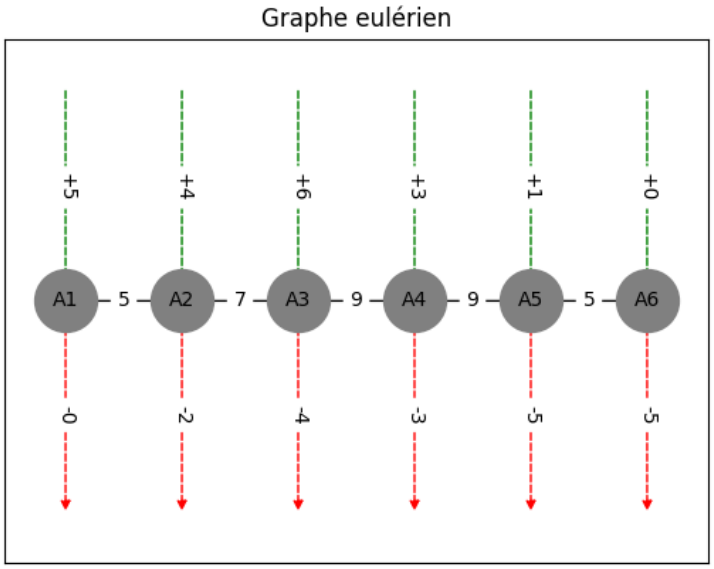

FIGURE 2 – Représentation ligne de bus à 6 arrêts à partir de données eulériennes

Avec l'algorithme exhaustif, on obtient 200 matrices Origine-Destination qui correspondent aux données de montées et descentes.

On voit que le nombre de scénarios correspondant aux données eulériennes augmente très rapidement dès lors que l'on augmente le nombre d'arrêts et les valeurs dans les vecteurs μ et ν. Cette augmentation du nombre de possibilités s'accompagne également par une forte augmentation du temps de calcul (14.9 secondes pour le graphe à 6 arrêts contre 0.02 secondes pour le graphe à 5 arrêts).

En fait, la complexité de l'algorithme exhaustif est exponentielle dans le pire cas. En effet : pour ligne à N arrêts, il faut remplir une matrice Origine-Destination de taille N^2. Pour chaque coefficient γ_{ij}, il faut tester toutes les valeurs possibles entre 0 et $min(m_i; v_j)$ sauf pour ceux qui sont fixées à zéro qui représentent environ la moitié des cellules. La complexité totale peut donc être approximée par $O(k^{N^2/2})$ où k est le nombre moyen de valeurs possibles par cellules.

Ainsi, avec cet algorithme, on est assez vite limités par les capacités de calcul de la machine. On doit se restreindre à quelques arrêts avec peu de voyageurs, ce qui n'est pas très adapté à une utilisation réelle.

4 Sélection d'un plan Origine-Destination

Les deux algorithmes décrits ci-dessus, renvoient l'ensemble des matrices correspondantes aux données de montées et de descentes, mais il est intéressant de se demander si certains scénarios sont plus plausibles que d'autres et comment les sélectionner. On s'intéresse à la notion d'entropie :

Définition :
On considère une variable aléatoire discrète p qui prend ses valeurs dans un ensemble de cardinal n. La loi de p est décrite par :

$$p = (p_1, p_2, ..., p_n), \quad p_i \geq 0, \quad \sum p_i = 1.$$

On définit l'entropie de la variable aléatoire p comme

$$S(p) = \sum p_i log(p_i).$$

Remarque : Il existe également une définition de l'entropie où l'on prend l'opposé de cette définition.

Pour une ligne de bus donnée, on introduit $K = \sum_{i,j} \gamma_{ij}$ le nombre de voyageurs total. On peut alors voir le plan Origine-Destination γ comme une variable aléatoire dont les issues sont les trajets $i \longrightarrow j$ et dont la loi de probabilité est décrite par :

$$\mathbb{P}(\gamma = i \longrightarrow j) = \bar{\gamma}_{ij} = \frac{\gamma_{ij}}{K}.$$

Proposition : L'entropie d'un plan Origine-Destination γ est donc :

$$S(\gamma) = \sum_{i,j} \bar{\gamma}_{ij} log(\bar{\gamma}_{ij}).$$

On peut interpréter l'entropie comme une mesure du désordre ou, dans notre cas, une mesure de la diversité des déplacements des voyageurs. Une entropie minimale correspond à une répartition relativement uniforme entre les trajets possibles, il est difficile de prédire le trajet que fera un voyageur. Au contraire, une grande entropie traduit la présence de trajets plus populaires que d'autres.

Pour sélectionner un scénario (ou une matrice). Nous faisons l'hypothèse d'une répartition uniforme des trajets et on cherche donc la matrice Origine-Destination qui minimise l'entropie.

Proposition : C'est un problème de minimisation d'une fonctionnelle strictement convexe sur un compact convexe, il admet donc une unique solution. (cf. section 4.2.1)

4.1 Calcul de l'entropie sur l'ensemble des solutions

La première approche envisagée a été de calculer l'entropie de toutes les matrices Origine-Destination correspondant à des vecteurs montées et descentes donnés et de finalement renvoyer la matrice minimisant l'entropie. Pour cela, on a implémenté une fonction Python.

Pour la ligne de bus à 5 arrêts (1), c'est la matrice suivante qui minimise l'entropie :

$$\gamma_1 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

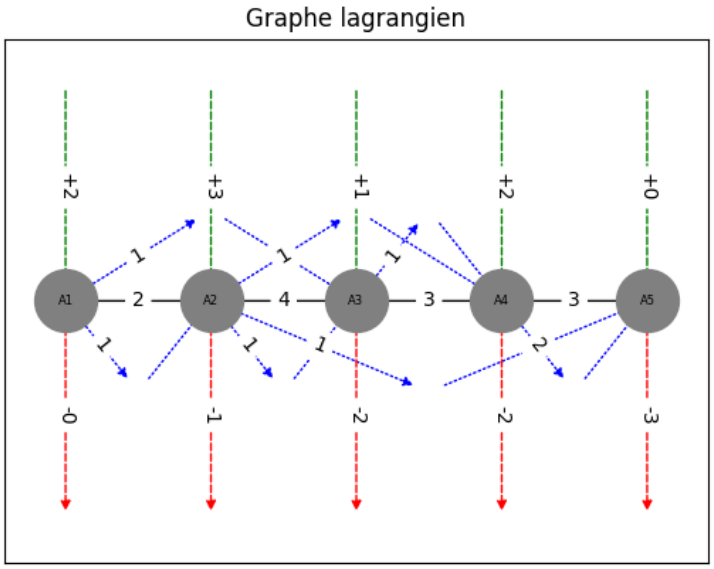

FIGURE 3 – Représentation ligne de bus à 5 arrêts minimisant l'entropie

Les valeurs et la longueur de cette ligne étant très petites, on ne peut pas vraiment interpréter le résultat. La figure (3) représente le graphe de la ligne de bus avec en bleu les trajets des voyageurs.

Pour la ligne à six arrêts (2), c'est la matrice suivante qui minimise l'entropie. La figure (4) représente le graphe de la ligne de bus avec en bleu les trajets des voyageurs.

$$\gamma = \begin{pmatrix} 0 & 2 & 2 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

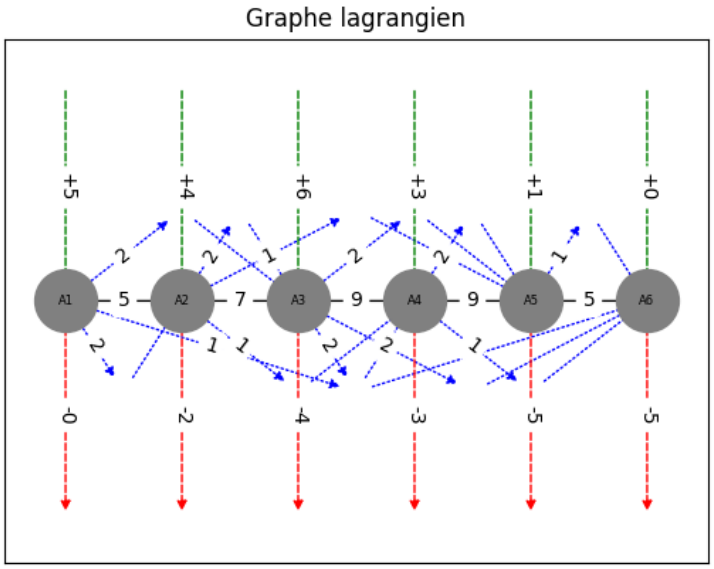

FIGURE 4 – Représentation ligne de bus à 6 arrêts minimisant l'entropie

Dans ce cas, on voit apparaitre plus clairement la forte répartition des trajets : il y a de nombreux trajets différents et aucun n'est privilégié, les voyageurs sont répartis assez uniformément.

L'avantage de cette méthode est qu'elle renvoie une matrice correcte (*ie* qui respecte totalement les contraintes de montées et de descentes) et qu'elle est composée d'entiers. Cependant, un important désavantage est qu'elle nécessite d'avoir au préalable calculé l'ensemble des matrices (entières) correspondantes, et comme nous l'avons vu précédemment, cela peut demander énormément de temps si la ligne est longue ou avec beaucoup de voyageurs. Cela peut donc fonctionner pour de petites lignes avec de petites valeurs (moins d'une dizaine d'arrêts et une dizaine de passagers) mais ne semble pas pertinente au-delà. Il est donc naturel de se tourner vers des méthodes permettant de calculer directement la matrice minimisant l'entropie.

4.2 Recherche par des méthodes d'optimisation

Pour cela, nous allons utiliser des méthodes dites d'optimisation et plus précisément de minimisation que nous allons appliquer à l'entropie.

Pour une ligne de bus à N arrêts, il y a $d = \frac{N(N-1)}{2}$ données à trouver (matrice triangulaire strictement supérieure). La fonction à minimiser est donc :

$$\begin{array}{rcl} S : \mathbb{R}^d & \to & \mathbb{R} \\ x & \mapsto & \sum_{i=1}^{d} x_i log(x_i). \end{array}$$

Le problème de minimisation est donc :

$$\inf_{x \in \mathbb{R}^d} \; S(x) \quad (P_S)$$

Nous allons tout d'abord poser un cadre théorique de la minimisation puis nous l'appliquerons à notre problématique.

4.2.1 Théorie de l'optimisation

Cette section se base sur le cours d'optimisation de Quentin Merigot [Merigot(2020)].

Définition :
Soit $f \in C^1(\mathbb{R}^d)$,
1. On appelle minimiseur global (ou minimiseur) tout élément $x^* \in \mathbb{R}^d$ vérifiant $f(x^*) = \inf_{R^d} f$. On note $arg\min_{\mathbb{R}^d} f$ l'ensemble des minimiseurs de f (qui peut être vide) *ie* :

$$arg\min_{R^d} f = \{x \in \mathbb{R}^d | f(x) = \inf_{\mathbb{R}^d} f.\}$$

2. On appelle suite minimisante pour (P) toute suite $x^{(0)}, ..., x^{(k)}$, ... d'éléments de $\mathbb{R}^d$ telle que $\lim_{k\to+\infty} f(x^{(k)}) = \inf_{x\in\mathbb{R}^d} f(x)$.

Il existe plusieurs méthodes qui permettent de résoudre des problèmes de minimisation sans contraintes : gradient à pas fixe, à pas optimal, gradient préconditionné à rebroussement, méthode de Newton, etc. qui ne seront pas développées ici.

Dans notre cas, les matrices Origine-Destination recevables sont contraintes par les données de montées et de descentes. Nous allons donc nous intéresser aux méthodes d'optimisation sous contraintes.

Pour ce type de problème, les contraintes sont définies par des inégalités avec des fonctions convexes $c_1, \ldots, c_l \in C^1(\mathbb{R}^d)$ qui forment alors un ensemble $K \subseteq \mathbb{R}^d$, convexe fermé :

$$K = \{x \in \mathbb{R}^d \mid c_1(x) \leq 0, \ldots, c_l(x) \leq 0\}.$$

On appelle chacune des fonctions $c_1, \ldots, c_l$ une contrainte du problème et on note alors le problème d'optimisation pour une fonction f convexe comme :

$$P := \min_{c_1(x),\ldots,c_l(x)} f(x).$$

Proposition : (Existence d'une solution)
Un problème d'optimisation sous contraintes tel que défini précédemment admet une solution si l'une des conditions suivantes est vérifiée :
1. K est compact, non vide et f est continue.
2. K est fermé, non vide, f est continue et $\lim_{\|x\|\to+\infty} f(x) = +\infty$.

Proposition : (Unicité)
Si la fonction f est strictement convexe et si l'ensemble K est convexe, alors le problème d'optimisation admet au plus une solution.

Proposition : (Lien entre la héssienne et la stricte convexité)
En notant $(e_i)_{1\leq i,j\leq d}$ est la base canonique de $\mathbb{R}^d$, si $\forall x \in \mathbb{R}^d$, $\quad D^2 f(x) = (\frac{\partial^2 f}{\partial e_i \partial e_j}(x))_{1\leq i,j\leq d}$ (héssienne de f) est définie positive, alors f est strictement convexe.

Pour faire la démonstration de cette proposition, nous utilisons les quatre lemmes suivants :

Lemme : (1) Soit $f \in C^1(\mathbb{R}^d)$. Étant donnés $x \in \Omega$ et $v \in \mathbb{R}^d$, on définit $x_t = x + tv$ et $g(t) = f(x_t)$. Alors,

$$g'(t) =< \nabla f(x_t)|v > .$$

Lemme : (2) Soit $\Omega \subset \mathbb{R}^d$ ouvert, $f \in C^2(\Omega)$, $x \in \Omega$, $v \in \mathbb{R}^d$ et $g\ t \ \mapsto f(x_t)$ où $x_t = x + tv$. Alors,

$$g''(t) =< D^2 f(x_t)v|v > .$$

Démonstration : (2) On fait le calcul en coordonnées, en notant $(e_k)_k$ la base canonique :

$$g(t) = f(x + \sum_x tv_k e_k)$$

$$g'(t) = \sum_i v_i \frac{\partial f}{\partial e_i}(x + \sum_k tv_k e_k) =< \nabla f(x_t)|v >$$

$$g''(t) = \sum_i \sum_j v_i v_i \frac{\partial^2 f}{\partial e_j \partial e_i}(x + \sum_k tv_k e_k) =< D^2 f(x_t)v|v > . \quad \square$$

Lemme : (Taylor-Lagrange). Soir $f \in C^2(\mathbb{R}^d)$, $x, v \in \mathbb{R}^d$ et $x_t = x + tv$. Alors,

$$\forall t \geq 0, \exists s \in [0, t], \quad f(x_t) = f(x) + t < \nabla f(x)|v > + \frac{t^2}{2} < D^2 f(x_s)v|v > .$$

Lemme : (3) Les propositions suivantes sont équivalentes :
1. f est (strictement) convexe sur $\mathbb{R}^d$
2.

$\forall x, y \in \mathbb{R}^d,$ la fonction $g : t \in [0, 1] \mapsto f((1 - t)x + ty)$ est (strictement) convexe.

3.

$\forall x, y \in \mathbb{R}^d,\ f(y) \geq f(x) + < y - x|\nabla f(x) >$ (stricetement convexe si inégalité stricte).

4.

$\forall x, y \in \mathbb{R}^d,\ < \nabla f(x) - \nabla f(y)|x - y > \geq 0$ (stricetement convexe si inégalité stricte).

Démonstration : (3)
1. $\iff$ 2. par application directe de la définition.
2. $\implies$ 3. Soit $x, y \in \mathbb{R}^d$ et $g : t \mapsto f(x_t)$ où $x_t = (1 - t)x + tv = x + t(y - x)$ qui est convexe par hypothèse. Par convexité, on a $g(t) \leq g(0) + tg'(0)$ soit par le lemme 1 : $f(x) + < \nabla f(x)|y - x > \leq f(y)$.
3. $\implies$ 4. Il suffit de sommer l'inégalité (3) et la même inégalité où l'on inverse les rôles de x et de y.

4. $\Longrightarrow$ 2. Soit encore $g : t \mapsto f(x_t)$. Comme $g'(t) =< \nabla f(x_t)|y - x >$ par le lemme (1), l'inégalité 4. appliquée en x_s et x_t (où t>s) nous donne

$$g'(t) - g'(s) =< \nabla f(x_t) - \nabla f(x_s)|y - x >= \frac{1}{t-s} < \nabla f(x_t) - \nabla f(x_s)|x_t - x_s > \geq 0$$

et g' est donc croissante sur [0,1]. Ainsi, g est convexe. □

Démonstration : (proposition) On considère $x, y \in \Omega$ et $g(t) = f(x_t)$ où $x_t = (1-t)x+ty$. Alors, par le lemme (2) avec $v = y - x$, on a $g''(t) =< D^2 f(x_t)(y_x)|(y - x) >$ est strictement positif par hypothèse, donc par le lemme (Taylor-Lagrange) on a :

$$g(1) = g(0) + g'(0 = +\frac{s^2}{2} g''(s) > g(0) + g'(0).$$

Or, $f(y) = g(1)$ et $f(x) = g(0)$. Donc, on a, $f(y) > f(x)+ < \nabla f(x)|y - x >$ et donc d'après le lemme (3) donc f est strictement convexe. □

Lemme : Si les fonctions $c_1, \ldots, c_l\, \mathbb{R}^d \to \mathbb{R}$ sont continues et convexes, alors l'ensemble $K = \{x \in \mathbb{R}^d \mid c_1(x) \leq 0, \ldots, c_l(x) \leq 0\}$ est fermé et convexe.

Démonstration :
K est fermé :
Soit (x_n) une suite d'élément de K convergent vers une limite x, montrons que $x \in K$. Par hypothèse, comme $x_n \in K$, $c_i(x_n) \leq 0$, $\forall i \in \{1, \ldots, l\}$. En passant à la limite $n \to +\infty$ en utilisant la continuité de c_i, on en déduit que $c_i(x) \leq 0, \forall i \in \{1, \ldots, l\}$. Donc x appartient à K et donc K est fermé.

K est convexe :
Soient $x, y \in K$ et $x_t = (1 - t)x + ty$. Comme $x, y \in K$, on a pour tout $i \in \{1, \ldots, l\}$, $c_i(x) \leq 0$ et $c_i(y) \leq 0$. Pour tout $t \in [0, 1]$, on a donc :

$$c_i(x_t) \leq (1 - t)c_i(x) + tc_i(y) \leq 0$$

Donc x_t appartient à K et donc K est convexe. □

Méthode pour trouver le minimiseur :
Pour résoudre les problèmes d'optimisation sous contraintes $min_K f$ où $K = \{x \quad | \quad \forall i, c_i(x) \leq 0\}$, une idée peut être de se ramener à un problème d'optimisation sans contraintes :

$$\min_{\mathbb{R}^d} f_\epsilon$$

où est f_ϵ la somme de f et de termes qui vont exploser lorsque les conditions ne seront pas vérifiées. Précisément, pour $\epsilon > 0$, on pose :

$$P_\epsilon = \min_{x \in \mathbb{R}^d} f_\epsilon(x) \text{ où } f_\epsilon(x) = f(x) + \frac{1}{\epsilon} \sum_{i=1}^{l} max(c_i(x), 0)^2.$$

Dans cette formulation du problème, les points $x \in \mathbb{R}^d$ tels que $c_i(x) > 0$ sont pénalisés au sens où si ϵ est très petit alors $\frac{1}{\epsilon} max(c_i(x), 0)^2$ peut devenir très grand, ce qui va

dissuader le choix de ce point dans le problème d'optimisation sur $\mathbb{R}^d$:

$$\forall x \in \mathbb{R}^d, \lim_{\epsilon \to 0} \frac{1}{\epsilon} \sum_{i=1}^{l} max(c_i(x), 0)^2 = \begin{cases} 0 & \text{si } c_i(x) \leq 0 \\ +\infty & \text{sinon.} \end{cases}$$

Proposition : Soit $f \in C^0(\mathbb{R}^d)$ strictement convexe et vérifiant $\lim_{\|x\| \to +\infty} f(x) = +\infty$, alors :
1. Les problèmes P_ϵ et P admettent chacun un unique minimiseur, noté x^* et x_ϵ^*.
2. $\lim_{\epsilon \to 0} x_\epsilon^* = x^*$.

Démonstration :
1. L'existence d'une solution au problème d'optimisation sans contraintes P_ϵ se déduit du fait que f_ϵ est continue et que $\lim_{\|x\| \to +\infty} f_\epsilon(x) \geq \lim_{\|x\| \to +\infty} f(x) = +\infty$. Tandis que l'existence de solution au problème avec contraintes P se déduit du fait que $\lim_{\|x\| \to +\infty} f(x) = +\infty$ et que K est fermé (lemme précédent). Pour l'unicité de la solution au problème P, il suffit de remarquer que f est strictement convexe (par hypothèse) et que K est convexe (lemme précédent à nouveau). Enfin, pour l'unicité de la solution du problème P_ϵ il faut montrer que f_ϵ est strictement convexe. Soient $x, y \in \mathbb{R}^d, t \in [0, 1]$ et $x_t = (1-t)x + ty$. Alors,

$$c_i(x_t) \leq (1-t)c_i(x) + tc_i(y).$$

De sorte que :

$$max(c_i(x_t), 0) \leq max((1-t)c_i(x) + tc_i(y), 0) \leq (1-t)max(c_i(x), 0) + tmax(c_i(y), 0).$$

Puis en utilisant la convexité de $r \in \mathbb{R} \mapsto r^2$, on a :

$$max(c_i(x_t), 0)^2 \leq [(1-t)max(c_i(x), 0) + tmax(c_i(y), 0)]^2 \leq (1-t)max(c_i(x), 0)^2 + tmax(c_i(y), 0)^2.$$

On en déduit que les fonctions $x \mapsto max(c_i(x), 0)^2$ sont convexes et f_ϵ est donc strictement convexe comme combinaison linéaire à coefficients positifs de fonctions convexes et d'une fonction strictement convexe.

2. Soit $x^* \in K$ le minimiseur de P. Alors comme $c_i(x^*) \leq 0$, on a :

$$P_\epsilon \leq f_\epsilon(x^*) = f(x^*) + \frac{1}{\epsilon} \sum_{i=1}^{l} max(c_i(x^*), 0)^2 = f(x^*) = P.$$

Ainsi,

$$P_\epsilon = f(x_\epsilon^*) + \frac{1}{\epsilon} \sum_{i=1}^{l} max(c_i(x_\epsilon^*), 0)^2 \leq P.$$

Comme $\lim_{\|x\| \to +\infty} f(x) = +\infty$, la fonction f est minorée par $m \in \mathbb{R}$. On en déduit donc que :

$$\forall i \{1, ..., l\}, \quad \frac{1}{\epsilon} max(c_i(x_\epsilon^*), 0) \leq \frac{1}{\epsilon} \sum_{j=1}^{l} max(c_j(x_\epsilon^*), 0)^2 \leq P - f(x_\epsilon^*) \leq P - m$$

ou encore

$$max(c_i(x_\epsilon^*), 0)^2 \leq \epsilon(P - m).$$

Soit $\bar{x}$ une valeur d'adhérence de la suite x_ϵ^* lorsque $\epsilon \to 0$. Alors, par continuité,

$$max(c_i(\bar{x}), 0)^2 \leq 0$$

ce qui montre que $\bar{x} \in K$. De plus, pour tout $\epsilon > 0$, $f(x_\epsilon) \leq P_\epsilon \leq P$ de sorte que $f(\bar{x}) \leq P$. On en déduit que $\bar{x}$ minimise f sur K, soit $\bar{x} = x^*$. Pour conclure, on remarque que la suite (x_ϵ^*) est bornée et admet une valeur d'adhérence de sorte que $\lim_{\epsilon \to 0} x_\epsilon^* = x^*$. $\square$

4.2.2 Application à l'entropie

Vérifions maintenant que le problème de minimisation de l'entropie avec les contraintes de montées et de descentes se trouvent dans le cadre théorique décrit précédemment.

Tout d'abord, vérifions que $S : x \in \mathbb{R}^d \mapsto \sum_{i=1}^d x_i log(x_i)$, est C^2, strictement convexe et vérifie $\lim_{\|x\| \to +\infty} f(x) = +\infty$.

1. Continuité :
La fonction log est C^∞ sur $\mathbb{R}_*^+$.

2. Stricte convexité :
Pour cela, nous allons utiliser la première proposition de la section 4.2.1 en étudiant la hessienne de S, $\quad \forall i, j \in \{1, ..., d\}$:

$$i \neq j : \frac{\partial^2 S}{\partial_i \partial_j}(x) = 0$$

$$i = j : \frac{\partial^2 S}{\partial_i^2}(x) = \frac{1}{x_i}.$$

Donc la hessienne de $D^2S(x)$ pour $x \in \mathbb{R}^d$ est une matrice diagonale, donc le spectre est $\{\frac{1}{x_1}, ..., \frac{1}{x_d}\}$ tous non-nuls donc $D^2S(x)$ est définie positive et donc S est strictement convexe.

3. Limite :

$$\forall x_i \in \mathbb{R}, \lim_{x_i \to +\infty} x_i log(x_i) = +\infty$$

donc on a $\forall x \in \mathbb{R}^d \lim_{\|x\| \to +\infty} f(x) = +\infty$.

Transformons les contraintes de montées et de descente en fonctions convexes et continues $c_i : x \in \mathbb{R}^d \mapsto \mathbb{R}$.

— On a tout d'abord la contrainte de la définition du logarithme (qui correspond également à la réalité) : les x_i sont des réels positifs. Donc $\forall i \in \{1, ..., d\}$, $c_i(x) = -x_i \leq 0$.

— On a les contraintes sur les données des montées $\mu \in \mathbb{R}^N$ qui sont égales aux sommes sur les lignes. Comme ce sont des égalités, on pose deux contraintes opposées à chaque fois. Pour faciliter l'écriture, on se replace dans la vue matricielle :
$c_{d+i} = \sum_{j=1}^{N} \gamma_{ij} - \mu_i \leq 0$
$c_{d+N+i} = -(\sum_{j=1}^{N} \gamma_{ij} - \mu_i) \leq 0$

— On a les contraintes sur les données de descentes. À nouveau, ce sont des égalités, donc on pose deux inégalités pour chaque colonne.
$c_{d+2N+j} = \sum_{i=1}^{N} \gamma_{ij} - \nu_j \leq 0$
$c_{d+3N+j} = -(\sum_{i=1}^{N} \gamma_{ij} - \nu_j) \leq 0$

Toutes les fonctions de contraintes sont des fonctions polynomiales du premier degré donc sont convexes et continues alors $K = \{x \in \mathbb{R}^d \mid c_1(x) \leq 0, \ldots, c_l(x) \leq 0\}$ est un convexe fermé.

Notre problème vérifie donc bien toutes les conditions d'un problème de minimisation sous contraintes : il existe bien une unique solution que l'on va pouvoir approcher à l'aide de méthodes d'optimisation, notamment la méthode de pénalisation.

$$P_{S\epsilon} = \min_{\mathbb{R}^d} \sum_{i=1}^{d} x_i log(x_i) + \frac{1}{\epsilon} \sum_{j=1}^{l} max(c_i(x), 0)^2$$

4.2.3 Méthodes d'optimisation en python

La suite naturelle est d'implémenter ces méthodes en Python et de les tester sur différentes données.

Une des difficultés du code a été les passages de la vue vectorielle ($\mathbb{R}^d$), qui correspond au vecteur des fonctions d'optimisation, à la vue matricielle ($\mathbb{R}^{N \times N}$) qui correspond à nos données et notamment aux contraintes. Voici un exemple de passage de l'un à l'autre pour une matrice Origine-Destination de taille $N = 5$ et $d = 10$:

$$\gamma = \begin{pmatrix} \gamma_{00} & \gamma_{01} & \gamma_{02} & \gamma_{03} & \gamma_{04} \\ \gamma_{10} & \gamma_{11} & \gamma_{12} & \gamma_{13} & \gamma_{14} \\ \gamma_{20} & \gamma_{21} & \gamma_{22} & \gamma_{23} & \gamma_{24} \\ \gamma_{30} & \gamma_{31} & \gamma_{32} & \gamma_{33} & \gamma_{34} \\ \gamma_{40} & \gamma_{41} & \gamma_{42} & \gamma_{43} & \gamma_{44} \end{pmatrix} = \begin{pmatrix} 0 & x_1 & x_2 & x_3 & x_4 \\ 0 & 0 & x_5 & x_6 & x_7 \\ 0 & 0 & 0 & x_8 & x_9 \\ 0 & 0 & 0 & 0 & x_{10} \end{pmatrix} \rightarrow x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{pmatrix}$$

Première méthode : La première méthode d'optimisation explorée est une de celles proposées par la bibliothèque Scipy, il s'agit de la fonction MINIMIZE avec la méthode TRUST-CONSTR (Trust-Region Constrained Algorithm). Pour cette méthode, il

est nécessaire de définir des bornes pour les valeurs $[0, +\infty[$ ainsi qu'une matrice de contraintes. C'est une matrice A telle que :

$$Ax = \begin{pmatrix} \mu_0 \\ \vdots \\ \mu_N \\ \nu_0 \\ \vdots \\ \nu_N \end{pmatrix} \quad \text{Pour le cas } N = 4, A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Il est tout à fait possible de généraliser la matrice A, mais son écriture, notamment pour les descentes, n'est pas très pratique. Il est plus simple de la coder en repassant par la vue matricielle. On passe en argument la fonction ainsi que son gradient et sa hessienne pour éviter que leur calcul par Scipy amènent à de trop grandes approximations.

$$S(x) = \sum_{i=1}^{n} x_i \times \log(x_i)$$

$$\nabla S(x)_i = \log(x_i) + 1$$

$$D^2(x)_{i,j} = \begin{cases} 0 & \text{si } i \neq j \\ \frac{1}{x_i} & \text{sinon.} \end{cases}$$

Cependant, la fonction log et la fonction inverse n'étant pas définies pour des valeurs nulles ou négatives, les fonctions ont été légèrement modifiées pour éviter les erreurs lors de l'exécution du code :

```
def entropie(x):
    res = 0
    # Entropie
    for x_i in x:
        if x_i > 0:
            res += x_i * np.log(x_i)
    return res

# Definition de la jacobienne associee
def jacobian_entropie(x):
    res = []
    for x_i in x:
        if x_i > 0:
            res += [np.log(x_i) + 1]
        else:
            res += [0]
    return res

# Definition de la hessienne associee
def hessian_entropie(x):
    res = []
    for i in range(len(x)):
```

```
            ligne_res = []
            for j in range(len(x)):
                if i != j:
                    ligne_res.append(0)
                else:
                    ligne_res.append(1 / x[i])
            res.append(ligne_res)
        return res
```

Seconde méthode : La seconde méthode explorée a été une méthode de pénalisation comme décrite dans la section précédente. Pour cela, on se base également sur la bibliothèque Scipy et la fonction MINIMIZE mais cette fois, on utilise la méthode NELDER-MEAD qui n'utilise pas de borne et de contraintes linéaires. La fonction passée en argument est donc :

$$\begin{aligned} S_\epsilon(x) = &\sum_{i=1}^{N}(x_i log(x_i) + \frac{1}{\epsilon}\max(-i,0)^2) \\ &+ \sum_{i=1}^{N}\frac{1}{\epsilon}(\sum_{j \text{ sur la ligne i}} x_j - \mu_i)^2 \\ &+ \sum_{j=1}^{N}\frac{1}{\epsilon}(\sum_{i \text{ sur la colonne j}} x_i - \nu_j)^2 \end{aligned}$$

Comme pour la première méthode, on a également calculé le gradient pour éviter les erreurs d'approximation. Pour cela, on va également devoir utiliser la vue matricielle. On suppose que x_i est sur la ligne k et la colonne l :

$$\begin{aligned} \nabla S_\epsilon(x)_i = &\log(x_i) + 1 \\ &+ \frac{2}{\epsilon}(\text{ somme des termes sur la même ligne que } x_i \text{ en vue matricielle} - \mu_k) \\ &+ \frac{2}{\epsilon}(\text{ somme des termes sur la même colonne que } x_i \text{ en vue matricielle} - \mu_l) \end{aligned}$$

Il a fallu ensuite définir une valeur pour le terme de pénalisation $\frac{1}{\epsilon}$. En effet, si on le choisit trop petit, les contraintes ne seront pas forcément respectées, mais si on le choisit trop grand, il est trop restrictif et on ne peut pas aboutir à un résultat convenable. Pour cela, on a effectué des tests sur des vecteurs de différentes longueurs, il est ressorti que prendre $\epsilon = 0.01$ ou $\frac{1}{\epsilon} = 100$ était le choix qui amenait à la meilleure qualité, notion définie (4.2.4) un peu plus tard dans la partie test des méthodes.

Enfin, pour les deux méthodes, il est nécessaire de choisir un vecteur initial $x_0 \in \mathbb{R}^d$. Pour cela, on a fait le choix du vecteur correspondant au produit des marginales (en vue matricielle) : $\gamma_{ij} = \mu_i * \nu_j$. Ce choix permet de se placer dès le début dans l'espace des vecteurs respectant les contraintes.

4.2.4 Tests des méthodes d'optimisation Python

Le code python pour chacun de ces tests se trouve en annexe.

Test sur des vecteurs aléatoires :

On effectue un premier test sur des vecteurs générés aléatoirement à partir d'un nombre de voyageurs et d'un nombre d'arrêt donné. On fait le choix de tester les performances des deux méthodes sur deux aspects : le temps et la qualité du résultat. Pour définir la qualité du résultat, on ne peut pas se baser sur la distance à la matrice OD puisque que nous n'avons que les vecteurs de montées et de descentes, à la place, on s'intéresse donc au respect des contraintes. Pour une matrice M de taille n, on définit sa qualité comme :

$$\text{Qualité}(M) = (\sum_{i=1}^{n}(\sum_{j=1}^{n} M_{ij}) - \mu_i)^2 + (\sum_{j=1}^{n}(\sum_{i=1}^{n} M_{ij}) - \nu_j)^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} \max(0; -M_{ij})^2.$$

Plus la qualité est proche de 0, meilleur est le résultat.

Voici les résultats pour 5 essais pour des lignes de 5 à 10 arrêts avec à chaque fois (8×nombre d'arrêts) voyageurs. Attention, l'échelle des ordonnées n'est pas la même pour les deux graphes.

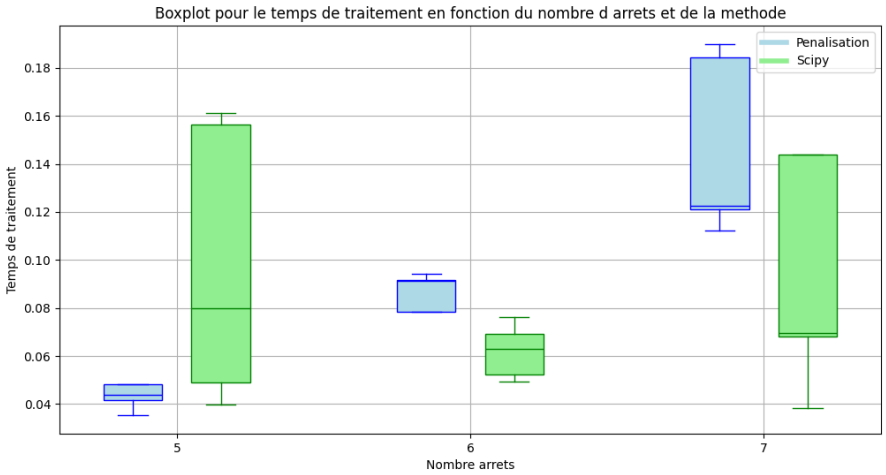

FIGURE 5 – Temps de traitement pour des lignes de taille 5 à 7 arrêts

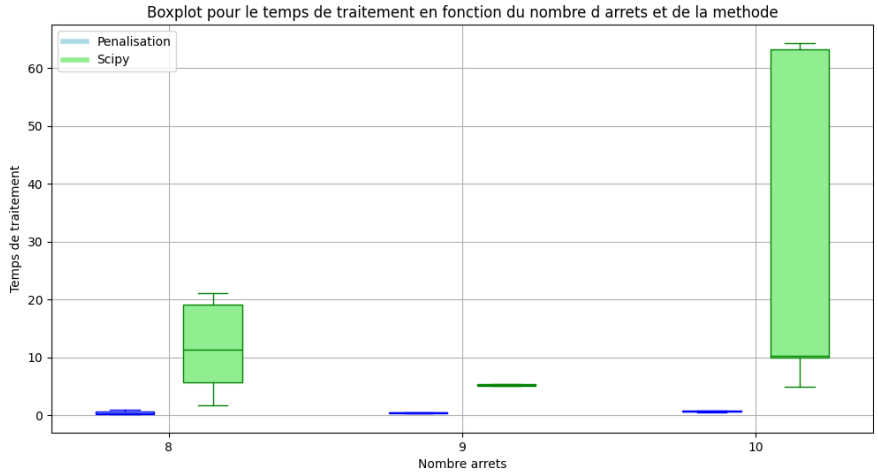

FIGURE 6 – Temps de traitement pour des lignes de taille 8 à 10 arrêts

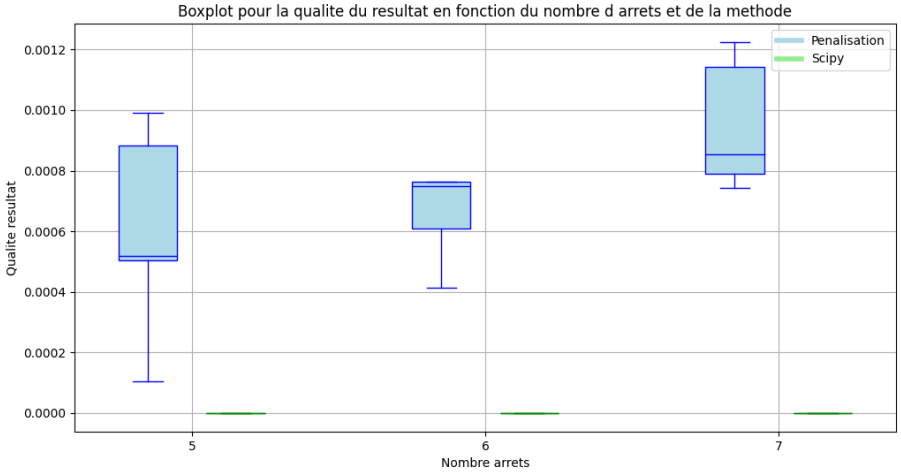

FIGURE 7 – Qualité pour des lignes de taille 5 à 7 arrêts

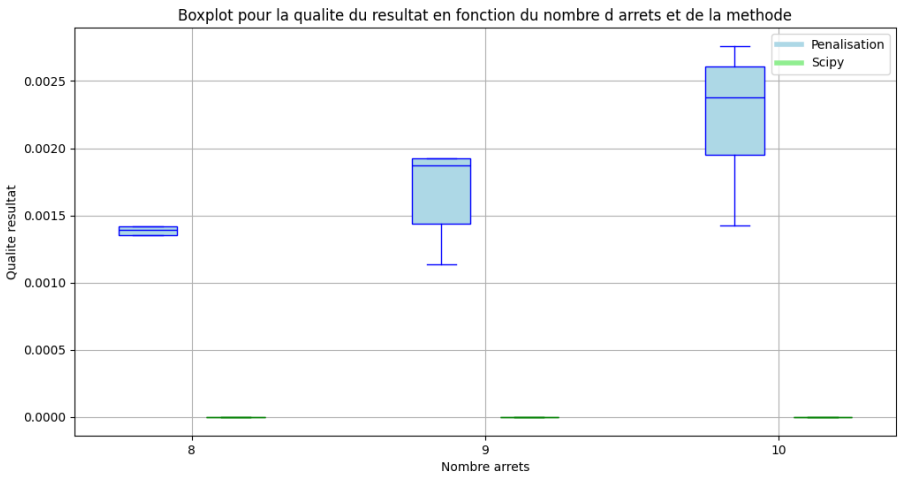

FIGURE 8 – Qualité pour des lignes de taille 8 à 10 arrêts

On voit que le temps de traitement est très faible pour les deux méthodes lorsque la ligne compte peu d'arrêts, mais devient beaucoup plus important pour la méthode scipy à partir de huit arrêts. Quant à la qualité du résultat, le respect des contraintes est très bon pour la méthode Scipy mais est un peu moins bonne pour la méthode de pénalisation (10^{-3}).

Il est intéressant de savoir si le résultat trouvé respecte bien les contraintes, mais cela ne nous donne aucune information quant à la qualité du résultat vis-à-vis de la réalité : le résultat est-il proche de la vraie répartition des voyageurs sur la ligne de transport ? On pourrait regarder la norme subordonnée des matrices trouvées et faire la différence avec celle de la vraie matrice Origine Destination, mais cette vision est plutôt adaptée à des applications linéaires et l'est moins pour le cadre dans lequel on se place. On introduit donc deux autres moyens de comparer les matrices trouvées et les vraies matrices : l'entropie relative et la distance au sens des moindres carrés.

Définition :
On note

$$P_N = \{\mu = (\mu_1, \dots, \mu_N) \in \mathbb{R}_+^N, \quad \sum_i \mu_i = 1\}$$

l'espace des lois de probabilités sur l'ensemble à N élément et $\overset{o}{P}_N$ le sous-ensemble des μ tels que pour tout i, $\mu_i > 0$. On définit l'entropie relative de μ par rapport à η par

$$\mu \in P_N \mapsto S_\eta(\mu) = \sum_i \mu_i log(\frac{\mu_i}{\eta_i}) = \sum_i \frac{\mu_i}{\eta_i} log(\frac{\mu_i}{\eta_i})\eta_i$$

Remarque : L'entropie relative est également appelée divergence de Kullback-Leibler et notée $D_{KL}(\mu|\nu)$.

Proposition :

$$\forall \eta \in \overset{o}{P}_N, \mu \in P_N, \quad S_\eta(\mu) \geq 0$$

Démonstration :

$$S_\eta(\mu) = \sum_i \mu_i log(\frac{\mu_i}{\eta_i}) = -\sum \mu_i log(\frac{\eta_i}{\mu_i})$$

Or le logarithme est une fonction concave donc par l'inégalité de Jensen on a :

$$\sum \mu_i log(\frac{\eta_i}{\mu_i}) \leq log(\sum \mu_i \frac{\eta_i}{\mu_i}) \leq log(\sum \eta_i) \leq log(1) \leq 0.$$

Donc on a bien $S_\eta(\mu) \geq 0$.

Proposition :

$$\mu = \eta \Rightarrow S_\eta(\mu) = 0.$$

Démonstration :

$$\forall i \in \{1, \ldots, N\}, \quad \mu_i = \eta_i \text{ donc } log(\frac{\mu_i}{\eta_i}) = 0 \text{ donc } S_\eta(\mu) = 0.$$

Plus l'entropie relative est faible plus les deux matrices sont semblables. Les méthodes de minimisation ayant tendance à donner des résultats avec peu de coefficients valant 0, on va plutôt regarde $S_{\text{trouvé}}(vrai)$ que $S_{vrai}(\text{trouvé})$ pour éviter quelques divisions par 0 mais dans les faits on peut avoir des valeurs négatives ou nulles. On choisit donc de coder ainsi l'entropie relative :

```
def distance_entropie_relative(matriceOD, matrice2, n):
    """
    :param matriceOD: matrice origine destination (avec des 0 potentiellement)
    :param matrice2: matrice trouvee par optimisation
    :param n: taille des matrices
    :return: l'entropie relative S_matrice2(matrice_OD)
    """
    res = 0
    for i in range(n):
        for j in range(i + 1, n):
            if matrice2[i][j] > 0 and matriceOD[i][j] > 0:
                res += matriceOD[i][j] * np.log(matriceOD[i][j] /
                    matrice2[i][j])
    return res
```

Définition :
On définit la distance au sens des moindres carrés comme :

$$\text{Dist}_{\text{mc}}(M_1,\ M_2) = \sum_i \sum_j (M_{1_{ij}} - M_{2_{ij}})^2.$$

Tout comme l'entropie relative, c'est une donnée positive qui est nulle lorsque $M_1 = M_2$. On va chercher à rendre cette valeur la plus petite possible. L'avantage de cette comparaison est qu'elle fonctionne même pour les matrices avec des coefficients nuls ou négatifs.

On s'intéresse également à l'entropie relative et la distance au sens des moindres carrés pour la matrice des marginales ($\gamma_{ij} = \mu_i \times \nu_j$). On peut alors regarder l'écart relatif entre les résultats pour l'une des deux méthodes et les résultats pour la matrice des marginales :

Définition :

$$\text{Ecart relatif entropie relative} = \frac{S_{marginales}(vrai) - S_{trouve}(vrai)}{S_{marginales}(vrai)}.$$

Définition :

$$\text{Ecart relatif distance moindres carrés} = \frac{\text{Dist}_{mc}(\text{marginales}, \text{vrai}) - \text{Dist}_{mc}(\text{trouve}, \text{vrai})}{\text{Dist}_{mc}(\text{marginales}, \text{vrai})}.$$

Remarque : On fait le choix de ne pas mettre de valeur absolue au numérateur pour savoir si le résultat trouvé par minimisation de l'entropie est une amélioration (écart relatif positif) ou non (écart relatif negatif) par rapport au croisement des marginales.

Test sur des lignes à 5 et 6 arrêts :
Le premier test que l'on effectue avec ces deux moyens de comparaison est celui sur les lignes à 5 et 6 arrêts que nous avions définies précédemment et pour lesquelles nous avons l'ensemble des matrices recevables et, parmi elles, celle qui minimise l'entropie (avec des entiers).

Ligne à 5 arrêts (arrondis à 4 chiffres après la virgule) :

$$\gamma_{\text{Scipy}} = \begin{pmatrix} 0 & 0.125 & 0.0527 & 0.04388 & 0.02842 \\ 0 & 0 & 0.1973 & 0.12393 & 0.05377 \\ 0 & 0 & 0 & 0.08219 & 0.04281 \\ 0 & 0 & 0 & 0 & 0.25 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_{\text{Pénalisation}} = \begin{pmatrix} 0 & 0.125 & 0.0625 & 0.04167 & 0.02083 \\ 0 & 0 & 0.1875 & 0.0125 & 0.0625 \\ 0 & 0 & 0 & 0.08333 & 0.04167 \\ 0 & 0 & 0 & 0 & 0.25 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_{\text{Min entropie valeurs entières normalisée}} = \begin{pmatrix} 0 & 0.125 & 0.125 & 0 & 0 \\ 0 & 0 & 0.125 & 0.125 & 0.125 \\ 0 & 0 & 0 & 0.125 & 0 \\ 0 & 0 & 0 & 0 & 0.25 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Ligne 5 arrêts	Pénalisation - Epsilon	Scipy	Marginales	Ecart-relatif - Penalisation	Ecart-relatif - Scipy
Entropie relative	0,1649	0,2099	0,7356	0,7759	0,7147
Moindres carrés	0,0168	0,0219	0,0630	0,7337	0,6519
Durée	0,0229	0,0478	/		

FIGURE 9 – Résultat comparaison pour la ligne à 5 arrêts

Pour cette ligne, on observe une amélioration de l'entropie relative et de la distance au sens des moindres carrés pour les deux méthodes avec des résultats légèrement meilleurs pour la méthode de pénalisation. On remarque surtout que le temps de traitement est très rapide. Pour les deux lignes, la durée du traitement est de quelques centièmes de secondes.

Ligne à 6 arrêts (arrondis à 4 chiffres après la virgule) :

$$\gamma_{\text{Scipy}} = \begin{pmatrix} 0 & 0.1053 & 0.0902 & 0.0226 & 0.0251 & 0.0201 \\ 0 & 0 & 0.1203 & 0.0301 & 0.0334 & 0.0267 \\ 0 & 0 & 0 & 0.1053 & 0.1169 & 0.0936 \\ 0 & 0 & & 00 & 0.0877 & 0.0702 \\ 0 & 0 & 0 & 0 & 0 & 0.0526 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_{\text{Pénalisation}} = \begin{pmatrix} 0 & 0.1043 & 0.0913 & 0.0252 & 0.2763 & 0.0221 \\ 0 & 0 & 0.1190 & 0.0328 & 0.036 & 0.0288 \\ 0 & 0 & 0 & 0.1061 & 0.1167 & 0.0931 \\ 0 & 0 & 0 & 0 & 0.0886 & 0.0707 \\ 0 & 0 & 0 & 0 & 0 & 0.0553 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_{\text{Min entropie valeurs entières normalisée}} = \begin{pmatrix} 0 & 0.1053 & 0.1053 & 0 & 0 & 0.0526 \\ 0 & 0 & 0.1053 & 0.0526 & 0.0526 & 0 \\ 0 & 0 & 0 & 0.1053 & 0.1053 & 0.1053 \\ 0 & 0 & 0 & 0 & 0.1053 & 0.0526 \\ 0 & 0 & 0 & 0 & 0 & 0.0526 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Ligne 6 arrêts	Pénalisation - Epsilon	Scipy	Marginales	Ecart-relatif - Penalisation	Ecart-relatif - Scipy
Entropie relative	0,0950	0,1117	0,6152	0,8455	0,8184
Moindres carrés	0,0051	0,0051	0,0322	0,8416	0,8406
Durée	0,0684	0,0293	/		

FIGURE 10 – Résultats des comparaisons pour la ligne à 6 arrêts

Pour la ligne à 6 arrêts, on observe des résultats similaires à ceux pour la ligne à 5 arrêts : une très bonne amélioration du résultat vis-à-vis de l'entropie relative et de la distance au sens des moindres carrés pour les deux méthodes et un temps de traitement très faible de l'ordre du centième de seconde.

Test sur des données réelles :

Il semble pertinent de comparer également les deux algorithmes sur de vraies données. Par le biais de Rennes Métropole, nous avons accès à de véritables données de déplacements issues de l'enquête réalisée au printemps 2023 où des matrices Origine-Destination avait été construites pour chaque ligne sur des tranches horaires de 15 minutes. On s'intéresse aux données de différentes lignes avec des tailles différentes et des types de trajets différents : les deux lignes de métro pendant une journée de semaine (15 arrêts chacune), la ligne C4 (35 arrêts) et la ligne C7 (19 arrêts). À chaque fois, nous avons travaillé sur la plage horaire de 8h45-8h59.

Ligne A - JOB	Pénalisation - Epsilon	Scipy	Marginales	Ecart-relatif - Penalisation	Ecart-relatif - Scipy
Entropie relative	0,2524	0,3567	0,5287	0,5226	0,3254
Moindres carrés	0,0115	0,0109	0,0115	-0,0038	0,0487
Durée	17,6405	92,6947	/		

FIGURE 11 – Résultats des comparaisons pour la ligne a du métro rennais direction Kennedy un jour de semaine

Ligne A - Samedi	Pénalisation - Epsilon	Scipy	Marginales	Ecart-relatif - Penalisation	Ecart-relatif - Scipy
Entropie relative	0,3157	0,3967	0,5692	0,4454	0,3031
Moindres carrés	0,0119	0,0104	0,0125	0,0456	0,1718
Durée	9,6170	74,8283	/		

FIGURE 12 – Résultats des comparaisons pour la ligne a du métro rennais direction Kennedy un samedi

Ligne B	Pénalisation - Epsilon	Scipy	Marginales	Ecart-relatif - Penalisation	Ecart-relatif - Scipy
Entropie relative	0,3204	0,3618	0,4857	0,3403	0,2552
Moindres carrés	0,0135	0,0121	0,0120	-0,1246	-0,0136
Durée	17,3614	30,7588			

FIGURE 13 – Résultats des comparaisons pour la ligne b du métro rennais direction Viasilva

Ligne C4	Pénalisation - Epsilon	Scipy	Marginales	Ecart-relatif - Penalisation	Ecart-relatif - Scipy
Entropie relative	0,9449		1,4431	0,3452	/
Moindres carrés	0,0257		0,0202	-0,2707	/
Durée	668,4649				

FIGURE 14 – Résultats des comparaisons pour la ligne de bus C4 direction ZA St Sulpice

Ligne C7	Pénalisation - Epsilon	Scipy	Marginales	Ecart-relatif - Penalisation	Ecart-relatif - Scipy
Entropie relative	0,4419	0,3054	0,8695	0,4917	0,6488
Moindres carrés	0,0143	0,0094	0,0219	0,3472	0,5689
Durée	32,1254	115,0079			

FIGURE 15 – Résultats des comparaisons pour la ligne de bus C7 direction Bruz

Pour des données réelles et des lignes plus longues, les résultats sont plus mitigés. Tout d'abord on peut observer la forte augmentation du temps de traitement de la méthode Scipy avec le nombre d'arrêts : quelques centièmes de secondes pour 5-6 arrêts contre 2 minutes pour les 19 arrêts de la ligne C7 et pas de résultats après plus d'une heure pour les 35 arrêts de la ligne C4. On observe également une augmentation du temps de traitement pour la méthode de pénalisation mais elle reste bien moins importante. En suite, à l'exception de la ligne C7, les résultats des deux méthodes sont à peine meilleurs voire pires que ceux de la matrice de croisement des marginales pour la distance aux moindres carrés. Cependant, pour l'entropie relative les deux méthodes de minimisation restent plus intéressantes que la matrice de croisement des marginales et notamment la méthode de pénalisation qui a de meilleurs résultats pour la plupart des lignes.

Ces trois tests nous ont permis de voir que la qualité de la méthode de pénalisation n'était pas très bonne comparer à celle de Scipy. Les méthodes de pénalisation et SciPy se sont montrées performantes en termes de temps et d'amélioration par rapport à la matrice de croisement des marginales pour l'entropie relative ainsi que pour la distance au sens des moindres carrés pour un nombre très réduit d'arrêts. Cependant, les deux méthodes ont de mauvais résultats dans l'ensemble pour des lignes avec plus d'arrêts : Scipy devient très lent et les deux méthodes ont de mauvais résultats pour la comparaison avec la distance au sens des moindres carrés. De plus, même si résultats sont plus positifs pour l'entropie relative, il faut garder en mémoire que le calcul de l'entropie relative est un peu fallacieux dans le code puisque l'on ignore les valeurs négatives et nulles, les résultats ne sont peut-être pas réllement aussi bons.

5 Deux autres méthodes probabilistes

Dans le cadre de sa thèse [Coulaud(2022)], Rémi Coulaud s'est intéressé à la charge à bord des zones des rames de la ligne H du transilien. L'objectif était de prévoir, à partir des données de montées et de descentes de chaque zone, les déplacements des voyageurs entre les zones et ainsi connaitre le taux d'occupation de celles-ci. Ce problème est assez proche de celui de la sélection d'une matrice Origine-Destination et il est possible d'adapter les deux méthodes probabilistes proposées à notre problématique. Pour cela, nous allons tout d'abord évoquer quelques notions de probabilités et statistiques [Schwarzentruber(2024)] puis présenter les deux modèles proposés par [Coulaud(2022)] et leurs résultats.

5.1 Notions de probabilités et statistiques

Définition : (Loi multinomiale)
On dit qu'un vecteur aléatoire $(X_1, \dots, X_r)$ suit une loi multinomiale de paramètres $n, p_1, \dots, p_r$ avec $p_i \in]0;1[$ et $p_1 + \cdots + p_r = 1$ si :

— $X_1(\Omega) = \cdots = X_r(\Omega) = \{0, \dots, n\}$.

— La loi de $(X_1, \dots, X_r)$ est donnée par :

$$P((X_1, \dots, X_r) = (k_1, \dots, k_r)) = \frac{n!}{k_1! \dots k_r!} p_1^{k_1} \dots p_r^{k_r}$$

si $k_1 + \cdots + k_r = n$, 0 sinon.

Exemple : Une urne contenant des boules de r couleurs notées $c_1, \dots, c_r$. On note p_i la proportion de boules de couleur c_i. On effectue n tirages avec remise, et on note X_i le nombre de boules de couleur c_i tirées. Alors $(X_1, \dots, X_r)$ suit une loi multinomiale de paramètres $n, p_1, \dots, p_r$.

Définition : (Matrice stochastique)
On dit que $A \in \mathcal{M}_n(\mathbb{R})$ est une matrice stochastique si tous ses coefficients sont compris entre 0 et 1 et si la somme de chaque ligne est égale à 1. De plus, elle est dite bistochastique si la somme de chaque colonne est égale à 1.

Lorsque l'on souhaite déterminer un paramètre θ d'une loi, une des méthodes couramment utilisées est celle du maximum de vraisemblance. La vraisemblance mesure de combien des données $x_1, \dots, x_n$ collent à la mesure de probabilité de paramètre θ. Pour mesurer la grandeur de ces probabilités, on les multiplie.

Définition : (Vraisemblance d'une loi discrète)
Soit θ le paramètre d'un modèle statistique où la loi de X est discrète. Soit $x_1, \dots, x_n$ les données. La vraisemblance des données $x_1, \dots, x_n$ est :

$$\mathcal{L}(x_1, \dots, x_n; \theta) := \prod_{i=1}^{n} \mathbb{P}(X = x_i; \theta)$$

Exemple : Soit X une variable aléatoire suivant une loi de Bernoulli de paramètre θ : $\mathbb{P}(X = 1) = \theta$ et $\mathbb{P}(X = 0) = 1 - \theta$. On considère les données $x_1, \dots, x_n$. La vraisemblance de ces données est :

$$\mathcal{L}(x_1, \dots, x_n; \theta) = \theta^{\sum_{i=1}^n x_i} (1 - \theta)^{\sum_{i=1}^n 1 - x_i}$$

Pour une loi continue, les probabilités $\mathbb{P}(X = x_i; \theta)$ sont nulles. Donc à la place, on regarde la fonction de densité de la loi $p(.; \theta)$.

Définition : (Vraisemblance d'une loi continue)

$$\mathcal{L}(x_1, \dots, x_n; \theta) := \prod_{i=1}^{n} p(x_i; \theta)$$

On cherche alors à maximiser la vraisemblance, *i.e.* à trouver la valeur du paramètre θ pour laquelle la vraisemblance est maximale pour des données $x_1, \ldots, x_n$ fixées. La valeur θ pour laquelle la vraisemblance est maximale est appelée estimateur du maximum de vraisemblance.

Définition : (Estimateur du maximum de vraisemblance)
Un estimateur de maximum de vraisemblance est une fonction $\hat{\theta}_n$ telle que

$$\hat{\theta}_n(x_1, \ldots, x_n) \in argmax_\theta \, \mathcal{L}(x_1, \ldots, x_n; \theta)$$

En pratique, il est souvent plus simple de calculer la log-vraisemblance $l = log(\mathcal{L})$ grâce aux propriétés du logarithme et par notamment son caractère monotone, la valeur qui maximise la log-vraisemblance maximisera également la vraisemblance.

5.2 Modélisation des déplacements au sein d'une rame

5.2.1 Problématique et lien avec la reconstruction des matrices Origine-Destination

Pour ce problème, on dispose de données sur les montées et descentes pour un train et une journée donnés et pour chaque gare et chaque zone des rames qui sont issues de capteurs infra-rouges placés au niveau des portes. Les voyageurs se déplacent ensuite librement entre les I zones d'une même rame et l'objectif est de déterminer le taux d'occupation de chacune des zones pour en informer les usagers, leur permettre de se répartir et d'augmenter leur confort.

Plusieurs hypothèses sont faites :
Hypothèse 1 : Les usagers montent dans une zone i, se déplacent et s'installent en zone j et descendent en zone j. Ils ne se re-déplacent pas pour descendre.

Hypothèse 2 : La charge à bord d'une gare donnée ne dépend que de la charge à bord de la gare précédente.

Hypothèse 3 : Les déplacements des personnes montées en i ne sont pas affectés par ceux des personnes montées en j.

L'objectif est d'obtenir la matrice $P = (p_{i,j})$ qui décrit les probabilités pour une personne montée dans une zone i de se déplacer en zone j (et de descendre en j).

La recherche de γ la matrice Origine-Destination d'une ligne de bus est un cas particulier de l'estimation de $P = (p_{i,j})$ où il y a seulement deux gares : les gens montent, se déplacent dans la rame jusqu'à leur zone de descente et descendent à la gare suivante. On obtient alors la matrice des probabilités de déplacement d'un usagé monté dans une zone et en multipliant par le nombre de personnes montées, on peut alors retrouver la matrice Origine-Destination :

$$\gamma = \left(\begin{array}{ccc|c} p_{1,1}\mu_1 & \cdots & p_{1,I}\mu_1 & \mu_0 \\ \vdots & \ddots & \vdots & \vdots \\ p_{I,1}\mu_I & \cdots & p_{I,I}\mu_I & \mu_I \\ \hline \nu_1 & \cdots & \nu_I & \end{array}\right)$$

Il faut cependant rajouter des contraintes supplémentaires dans le problème pour coller au cadre du problème des matrices OD :

— Il faut que P soit bistochastique et non plus seulement stochastique pour que la contrainte sur la somme sur les colonnes soit bien vérifiée.

— Il faut poser la contrainte d'avoir P triangulaire strictement supérieure à droite puisque les voyageurs de la ligne de bus vont vers le sens croissant des arrêts et ne peuvent pas revenir en arrière.

Il faut également poser la contrainte

Pour estimer P, deux méthodes sont proposées : une méthode de minimisation des moindres carrés sous contraintes et une méthode probabiliste.

Voici les notations utilisées par [Coulaud(2022)] :

$\mathcal{N}$: l'ensemble des couples (k, d) de trains et de jours.

$a_s^{k,d} = (a_{s,1}^{k,d}, \cdots, a_{s,I}^{k,d})$ le vecteur des descentes à la gare s pour le train k le jour d.

$b_s^{k,d} = (b_{s,1}^{k,d}, \cdots, b_{s,I}^{k,d})$ le vecteur des montées à la gare s pour le train k le jour d.

$x_{s,\cdot} = \sum_{i=1}^{I} x_{s,i}$ la somme selon les zones pour une gare s.

$x_{\cdot,i} = \sum_{s=1}^{S} x_{s,i}$ la somme selon les gares pour une zone i.

$x_{\cdot,\cdot} = i = 1 \sum_{s=1}^{S} x_{s,i}$ la somme selon les gares et les zones.

5.2.2 Moindres carrés sous contraintes

Le premier modèle proposé par [Coulaud(2022)] est un modèle de minimisation des moindres carrés.

On considère le problème comme un problème de régression linéaire multivariée sous contraintes où le vecteur ligne des descentes en fin de trajet $a_{\cdot} = (a_{\cdot,1}, \ldots, a_{\cdot,I})$ est le vecteur des variables à expliquer, le vecteur ligne des montées par zone $b_{\cdot} = (b_{\cdot,1}, \ldots, b_{\cdot,I})$ est la variable explicative et les coefficients de la régression sont les $p_{i,j}$ de la matrice P. Les contraintes sont que la somme des coefficients sur chaque ligne est égal à 1 et que les $p_{i,j}$ sont compris entre 0 et 1 (*ie.* $P = (p_{ij})$ est stochastique). Le problème à résoudre est le suivant :

$$A_{\cdot} = b_{\cdot} P + \epsilon$$

où ϵ est le vecteur des résidus.

Les observations se font sur l'ensemble des trajets (k, d) ce qui nous donne le problème des moindres carrés suivant :

$$\sum_{(k,d)\in\mathcal{N}} \|a_{\cdot}^{k,d} - b_{\cdot}^{k,d}P\|_2^2 = \sum_{(k,d)\in\mathcal{N}} \sum_{j=1}^{I} (a_{\cdot,j}^{k,d} - \sum_{i=1}^{I} b_{\cdot,i}^{k,d} p_{i,j})^2$$

L'estimation de P est alors la solution du problème des moindres carrés :

$$argmin_P \sum_{(k,d)\in\mathcal{N}} \|a_{\cdot}^{k,d} - b_{\cdot}^{k,d}P\|_2^2, \quad P \text{ est stochastique}$$

Il s'agit d'un problème d'optimisation quadratique sous contraintes linéaires que l'on peut résoudre numériquement.

5.2.3 Maximum de vraisemblance

Le second modèle proposé par [Coulaud(2022)] est une modèle par maximisation de la vraisemblance.

Pour chaque zone i parmi les I zones, les déplacements des voyageurs sont modélisés par un vecteur aléatoire $U_{\cdot,i}$ à I composantes où chaque $U_{\cdot,i}$ suit une loi multinomiale $\mathcal{M}(b_{\cdot,i}, p_{i,1}, \ldots, p_{i,I})$ avec $b_{\cdot,i}$ le nombre de personnes montées en zone i et $p_{i,j}$ la probabilité qu'un voyageur monté en i se déplace en zone j. L'ensemble des $p_{i,j}$ forme alors la matrice de passage P. L'hypothèse que les déplacements des personnes montées en i ne sont pas affectés par ceux des personnes montées en j se traduit par $\forall i \neq j \in \{1, \ldots, I\}$ $U_{\cdot,i} \perp\!\!\!\perp U_{\cdot,j}$.

Le vecteur des descentes par zones $A_{\cdot}$ est donc la somme des déplacements des voyageurs par zone :

$$A_{\cdot} = \sum_{j=1}^{I} U_{\cdot,j}$$

$A_{\cdot}$ est donc une somme de lois multinomiales indépendantes, mais pas identiques. On ne connait *a priori* pas la loi de $A_{\cdot}$ mais une approximation est proposée : une loi multinomiale $\mathcal{M}(b_{\cdot,\cdot}, \pi_{\cdot,1}, \ldots, \pi_{\cdot,I})$ avec $\pi_{\cdot}, j = \sum_{i=1}^{I} r_{\cdot,i} p_{i,j}$ où $r_{\cdot,i} = \frac{b_{\cdot,i}}{b_{\cdot,\cdot}}$. La probabilité de déplacement de i à j est pondérée par la proportion de personnes à monter en zone i.

La loi approximée de $A_{\cdot}$ a donc pour densité :

$$\mathbb{P}(A_{\cdot} = a_{\cdot}; b_{\cdot}) = \prod_{j=1}^{I} \frac{b_{\cdot,\cdot}!}{a_{\cdot,j}!} (\sum_{i=1}^{I} r_{\cdot,i} p_{i,j})^{a_{\cdot,j}}$$

On va alors chercher P permettant de maximiser la vraisemblance de cette loi.

La partie de la log-vraisemblance associée à une observation (k, d) ne dépendant que des paramètres est :

$$l(a_{\cdot}^{k,d}; P, b_{\cdot}^{k,d}) = \sum_{j=1}^{I} a_{\cdot,j}^{k,d} log(\sum_{i=1}^{I} r_{\cdot,i}^{k,d} p_{i,j})$$

Et comme toutes les observations (k, d) sont supposées indépendantes et ont la même loi, cette partie de la log-vraisembalnce devient :

$$l(P) = \sum_{(k,d)\in\mathcal{N}} \sum_{j=1}^{I} a_{\cdot,j}^{k,d} log(\sum_{i=1}^{I} r_{\cdot,i}^{k,d} p_{i,j})$$

Remarque : : La partie de la log-vraisemblance qui n'apparait pas ne dépend que des montées et des descentes, il n'est donc pas nécessaire de les prendre en compte pour une maximisation de la log-vraisemblance.

La recherche de la matrice stochastique P qui permet de maximiser la log-vraisemblance est donc un problème d'optimisation convexe qui peut être résolu numériquement.

5.2.4 Résultats

Pour évaluer les performances des deux méthodes d'estimation, il a été choisi de comparer la qualité des prévisions des descentes par les deux méthodes avec la stratégie où l'on suppose qu'il n'y a pas de déplacement, qui était alors la solution adoptée par Transilien. Cette comparaison se base sur un jeu de données avec plus de 3500 trajets effectués entre Gare du Nord et Pontoise en 2021. Le jeu est ensuite séparé en un jeu d'entrainement (75%) et un jeu de test (25%) et trois métriques différentes sont utilisées pour évaluer la qualité de l'estimation notée $\hat{a}_{\cdot}$:

— La moyenne des erreurs absolues :

$$\frac{1}{\text{Card(test)}} \sum_{(k,d)\in\text{test}} \|a_{\cdot}^{k,d} - \hat{a}_{\cdot}^{k,d}\|_1$$

— la moyenne du pourcentage des erreurs absolues :

$$\frac{1}{\text{Card(test)}} \sum_{(k,d)\in\text{test}} \|\frac{a_{\cdot}^{k,d} - \hat{a}_{\cdot}^{k,d}}{a_{\cdot}^{k,d}}\|_1$$

— la racine de la moyenne des erreurs au carré :

$$\sqrt{\frac{1}{\text{Card(test)}} \sum_{(k,d)\in\text{test}} \|a_{\cdot}^{k,d} - \hat{a}_{\cdot}^{k,d}\|_2^2}$$

Il ressort des tests que les deux méthodes d'estimation sont plus performantes que la stratégie où l'on suppose qu'il n'y a pas de déplacements avec une diminution par deux de l'erreur pour les deux méthodes qui ont des performances similaires. Cependant, on observe des différences entre les matrices de passage trouvées pour les rames avant et les

rames arrière et [Coulaud(2022)] indique donc qu'il est important de prendre le placement à quai et le sens de circulation en compte pour prévoir correctement les descentes.

Ainsi, ces deux méthodes sont deux pistes intéressantes à adapter au problème de modélisation des trajets effectués par les voyageurs. Il serait par exemple intéressant de les tester sur les données fournies par Rennes Métropole.

6 Conclusion

Nous avons finalement deux algorithmes qui permettent de décrire l'ensemble des scénarios. Cependant, du fait de leur complexité élevée, le temps de calcul devient trop grand pour les lignes de plus d'une dizaine d'arrêts avec beaucoup de voyageurs et ne sont donc pas utilisables dans la réalité.

La sélection d'un plan Origine-Destination par l'entropie s'est montré plus pertinente que l'estimation par le croisement des marginales pour de petites lignes avec un temps de calcul très faible. Cependant, on obtient des résultats plus mitigés pour les données réelles où les lignes sont plus longues et avec plus de voyageurs par la longueur du temps de calcul et la faible différence de qualité de prédiction par rapport au croisement des marginales.

Les deux méthodes issues de la thèse de Rémi Coulaud sont intéressantes et permettraient peut-être d'apporter de meilleurs résultats que la minimisation de l'entropie. Le jeu de données de Rennes Métropole pourrait par exemple permettre de tester ces méthodes.

7 Remerciements

Je tiens à remercier Sylvain Faure et Bertrand Maury de m'avoir pris en stage, de m'avoir accordé du temps et d'avoir guidé mon travail durant ces six semaines. Je voudrais également remercier toute l'équipe d'analyse numérique et équations aux dérivées partielles de m'avoir fait découvrir leur laboratoire, leur métier et leur recherche. La recette du gâteau au chocolat et au sarrasin se trouve en annexe (8.5).

Références

[Coulaud(2022)] R. Coulaud. *Modélisation et prévision des variables d'exploitation ferroviaire et de flux de voyageurs en zone dense*. PhD thesis, Université Paris-Saclay, 2022.

[Maury(2024)] B. Maury. Cours de modélisation mathématique, 2024.

[Merigot(2020)] Q. Merigot. Algorithmes d'optimisation, 2020.

[Schwarzentruber(2024)] F. Schwarzentruber. Cours de probabilités et statistiques, 2024.

8 Annexe

On peut retrouver l'ensemble des codes Python sur le repository github du stage. Le code est séparé en 4 fichiers :

— (8.1) graphes.py qui permet l'affichage des graphes de lignes de bus (sections 2-3).

— (8.2) bus.py où se trouvent les fonctions de passage du point de vue lagrangien à eulérien et inversement (sections 2-3).

— (8.3) extraction_donnees.py qui contient la fonction d'extraction des données des fichiers xlsx de Rennes Métropole (section 4).

— (8.4) minimisation_entropie.py qui contient les fonctions de minimisation de l'entropie pour la sélection d'un plan origine-destination pour les méthodes de pénalisation et scipy ainsi que les fonctions pour les tests réalisés (section 4).

8.1 graphes.py

```
import matplotlib.pyplot as plt
import networkx as nx
from bus import lagrange_to_euler


# Fonctions de conversion
def mv_to_b(m, v):
    """
    :param m: liste d'entier des montees a chaque arret
    :param v: liste d'entier des descentes a chaque arret
    :return: liste d'entier du bilan des montees et descentes a chaque arret
    """
    return [m[i] - v[i] for i in range(0, len(m))]


def mv_to_p(m, v):
    """
    :param m: liste d'entier des montees a chaque arret
    :param v: liste d'entier des descentes a chaque arret
    :return: la liste du nombre de voyageurs au depart du point i
    """
    p = [m[0]]
    for i in range(1, len(m) - 1):
        p += [p[i - 1] + m[i] - v[i]]

    return p


def p_to_b(p):
```

```
    """
    :param p: liste d'entier du nombre de personnes transportees entre les
        arrets
             p[i] = personnes transportees entre l'arret i et l'arret i+1
    :return: liste d'entier du bilan des montees et descentes a chaque arret
    """
    return [p[0]] + [p[i] - p[i - 1] for i in range(1, len(p))] + [-p[len(p)
        - 1]]


def b_to_p(b):
    """
    :param b: liste d'entier du bilan des montees et descentes a chaque arret
    :return: liste d'entier du nombre de personnes transportees entre les
        arrets
             p[i] = personnes transportees entre l'arret i et l'arret i+1
    """
    p = [b[0]]
    for i in range(1, len(b) - 1):
        p = p + [p[i - 1] + b[i]]
    return p


# Fonctions de conversion
def lagrange_to_graph(noms, M):
    """
    :param M: matrice OD carree d'entiers ou m_ij = nombre de personnes
        montees en i allant en j
    :return: le graphe networkx associe
    """
    # Initialisation du graphe
    G = nx.DiGraph()

    # Creation des arrets comme noeuds
    arrets = noms
    G.add_nodes_from(arrets)

    # Dessiner le graphe sans les noeuds fictifs
    pos = nx.spring_layout(G)

    # Definition des positions des noeuds reels
    for i in range(len(arrets)):
        pos[arrets[i]] = (0.5 * i, 0)

    # Creation des arretes de la ligne de bus
    trajets_ligne = [(noms[i], noms[i + 1]) for i in range(0, len(noms) - 1)]
    G.add_edges_from(trajets_ligne)

    # Creation des arretes pour les trajets des voyageurs (gamma_ij) et des
        noeuds ficitfs pour ces arretes
```

```
trajets_voyageurs_depart = []
trajets_voyageurs_arrivee = []
nbr_voyageurs = []
compteur = 0
for i in range(len(M[0])):
    for j in range(i + 1, len(M[0])): # Matrice triangulaire superieure
        nb = M[i][j]
        if nb > 0:
            trajets_voyageurs_depart.append((arrets[i],
                f"edge_{compteur}"))
            trajets_voyageurs_arrivee.append((f"edge_{compteur}",
                arrets[j]))
            val = 0.04
            if compteur % 2 == 0:
                val = -val
            pos[f"edge_{compteur}"] = ((pos[arrets[i]][0] +
                pos[arrets[j]][0]) / 2, pos[arrets[i]][1] + val)
            compteur += 1
            nbr_voyageurs.append(nb)

G.add_edges_from(trajets_voyageurs_depart)
G.add_edges_from(trajets_voyageurs_arrivee)

# Creation des arretes rentrantes pour les voyageurs montants
for arret in arrets:
    G.add_edge(f"start_{arret}", arret)

# Creation des arretes sortante pour les voyageurs descendants
for arret in arrets:
    G.add_edge(arret, f"end_{arret}")

# Positionnement des noeuds fictifs de montees et descentes
for arret in arrets:
    pos[f"start_{arret}"] = (pos[arret][0], pos[arret][1] + 0.1)
    pos[f"end_{arret}"] = (pos[arret][0], pos[arret][1] - 0.1)

# Dessiner les noeuds reels
nx.draw_networkx_nodes(G, pos, nodelist=arrets, node_color='grey',
    node_size=1000)

# Ajouter les labels aux noeuds reels
nx.draw_networkx_labels(G, pos, labels={arret: arret for arret in
    arrets}, font_size=6)

# Dessiner les aretes incluant celles des noeuds fictifs
## Trajets de la ligne de bus
nx.draw_networkx_edges(G, pos, edgelist=trajets_ligne,
    edge_color='black', arrows=True, arrowstyle='-|>')

## Trajets des voyageurs
```

```
    nx.draw_networkx_edges(G, pos, edgelist=trajets_voyageurs_depart,
        edge_color='blue', style='dotted', arrows=True,
                           arrowstyle='-|>')
    nx.draw_networkx_edges(G, pos, edgelist=trajets_voyageurs_arrivee,
        edge_color='blue', style='dotted', arrows=True,
                           arrowstyle='-|>')
    ## montees
    nx.draw_networkx_edges(G, pos, edgelist=[(f"start_{arret}", arret) for
        arret in arrets], edge_color='green',
                           style='dashed', arrows=True, arrowstyle='-|>')
    ## Descentes
    nx.draw_networkx_edges(G, pos, edgelist=[(arret, f"end_{arret}") for
        arret in arrets], edge_color='red',
                           style='dashed', arrows=True, arrowstyle='-|>')

    # Calcul de donnees
    m_M, v_M = lagrange_to_euler(M)
    p_M = mv_to_p(m_M, v_M)

    # Afficher les etiquettes des arretes
    # Afficher le nombre de voyageurs entre chaque arret de la ligne
    edge_labels = {(arrets[i], arrets[i + 1]): str(p_M[i]) for i in
        range(len(arrets) - 1)}

    # Afficher le nombre de voyageurs pour chaque trajet gamma_i,j
    edge_labels.update({trajets_voyageurs_depart[k]: nbr_voyageurs[k] for k
        in range(len(trajets_voyageurs_depart))})

    # Afficher les etiquettes des arretes rentrantes et sortantes
    edge_labels.update({("start_" + arrets[i], arrets[i]): "+" + str(m_M[i])
        for i in range(len(arrets))})
    edge_labels.update({(arrets[i], "end_" + arrets[i]): "-" + str(v_M[i])
        for i in range(len(arrets))})
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
        font_color='black')

    plt.title("Graphe lagrangien")
    plt.show()
    return None


def euler_to_graph(noms, m, v):
    """
        :param m: liste d'entiers des montees
        :param v: liste d'entiers des descentes
        :return: le graphe networkx associe
    """

    # Initialisation du graphe
    G = nx.DiGraph()
```

```
# Creation des arrets comme noeuds
arrets = noms
G.add_nodes_from(arrets)

# Creation des arretes pour les trajets entre stations
trajets = [(noms[i], noms[i + 1]) for i in range(0, len(noms) - 1)]
G.add_edges_from(trajets)

# Creation des arretes rentrantes pour les voyageurs montants
for arret in arrets:
    G.add_edge(f"start_{arret}", arret)

# Creation des arretes sortante pour les voyageurs descendants
for arret in arrets:
    G.add_edge(arret, f"end_{arret}")

# Dessiner le graphe sans les noeuds fictifs
pos = nx.spring_layout(G)

# Definition des positions des noeuds reels
for i in range(len(arrets)):
    pos[arrets[i]] = (len(m) * 2 * i, 0)

# Positionnement des noeuds fictifs
for arret in arrets:
    pos[f"start_{arret}"] = (pos[arret][0], pos[arret][1] + 0.05)
    pos[f"end_{arret}"] = (pos[arret][0], pos[arret][1] - 0.05)

# Dessiner les noeuds reels
nx.draw_networkx_nodes(G, pos, nodelist=arrets, node_color='grey',
    node_size=1000)

# Dessiner les aretes incluant celles des noeuds fictifs
nx.draw_networkx_edges(G, pos, edgelist=trajets, edge_color='black',
    arrows=True, arrowstyle='-|>')
nx.draw_networkx_edges(G, pos, edgelist=[(f"start_{arret}", arret) for
    arret in arrets], edge_color='green',
                       style='dashed', arrows=True, arrowstyle='-|>')
nx.draw_networkx_edges(G, pos, edgelist=[(arret, f"end_{arret}") for
    arret in arrets], edge_color='red',
                       style='dashed', arrows=True, arrowstyle='-|>')

# Ajouter les labels aux noeuds reels
nx.draw_networkx_labels(G, pos, labels={arret: arret for arret in
    arrets}, font_size=10)

# Afficher les etiquettes des arretes entre arrets
p = mv_to_p(m, v)
```

```
    edge_labels = {(arrets[i], arrets[i + 1]): str(p[i]) for i in
        range(len(arrets) - 1)}

    # Afficher les etiquettes des arretes rentrantes et sortantes
    edge_labels.update({("start_" + arrets[i], arrets[i]): "+" + str(m[i])
        for i in range(len(arrets))})
    edge_labels.update({(arrets[i], "end_" + arrets[i]): "-" + str(v[i]) for
        i in range(len(arrets))})
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
        font_color='black')

    plt.title("Graphe eulerien")
    plt.show()

    return G
```

8.2 bus.py

```
import numpy as np


# Passage de donnees lagrangiennes a euleriennes
def somme_ligne(M, i):
    """
    :param M: Une matrice d'entiers
    :param i: une entier correspondant a une ligne
    :return: la somme des valeurs de toutes les coefficients de la ligne i
    """
    res = 0
    for j in range(0, len(M)):
        res += M[i][j]
    return res


def somme_colonne(M, j):
    """
    :param M: Une matrice d'entiers
    :param j: Un entier correspondant a une colonne
    :return: La somme des coefficients de la colonne j
    """
    res = 0
    for i in range(0, len(M)):
        res += M[i][j]
    return res


def lagrange_to_euler(M):
    """
```

```
    :param M: matrice OD carree d'entiers ou m_ij = nombre de personnes
        montees en i allant en j
    :return: m liste d'entiers des montees a chaque arret
           v liste d'entiers des descentes a chaque arret
    """
    m = [somme_ligne(M, i) for i in range(0, len(M))]
    v = [somme_colonne(M, j) for j in range(0, len(M))]
    return m, v


# Passage de donnees euleriennes a donnees lagrangiennes
def euler_to_lagrange(m, v):
    """
    :param m: une liste d'entiers, les montees a chaque arret
    :param v: une liste d'entiers, les descentes a chaque arret
    :return: une liste contenant toutes les matrices OD correspondant aux
        donnees de montees et de descentes
    """
    n = len(m)
    resultats = []

    # Fonction recursive pour trouver les grilles
    def backtrack(grille, m_rest, v_rest, ligne, col):
        # Cas 1 : On a rempli la derniere cellule. On verifie si la grille est
            valide
        #        Si elle est valide, on la sauvegarde dans les resultats.
        if ligne == n:
            if all(sum(grille[i]) == m[i] for i in range(n)) and all(
                   sum(grille[i][j] for i in range(n)) == v[j] for j in
                       range(n)):
                resultats.append([row[:] for row in grille])
            return

        # Cas 2 : On a rempli la derniere colonne de la ligne actuelle. On
            passe a la ligne suivante
        if col == n:
            # On appelle la fonction backtrack pour continuer a remplir la
                matrice
            # en se placant sur la ligne +1 et sur la premiere colonne
            backtrack(grille, m_rest, v_rest, ligne + 1, 0)
            return

        # Cas 3 : On est sur la diagonale ou la partie inferieure gauche, la
            valeur doit etre 0.
        if ligne >= col:
            grille[ligne][col] = 0 # On met le coef a 0
            backtrack(grille, m_rest, v_rest, ligne, col + 1) # On continue a
                remplir la grille.
```

```
            # Autre cas : On appelle la fonction backtrack pour chaque valeur que
                peut prendre la cellule
            else:
                # On definit la valeur maximale que peut prendre la cellule a
                    partir de m_rest et v_rest.
                max_val = min(m_rest[ligne], v_rest[col])
                for val in range(max_val + 1):
                    grille[ligne][col] = val
                    # On modifie les valeurs de m_rest et v_rest pour remplir les
                        cellules restantes
                    # Et respecter les conditions sur les sommes de lignes et de
                        colonnes
                    m_rest[ligne] -= val
                    v_rest[col] -= val

                    # On appelle la fonction backtrack recursive pour remplir la
                        cellule suivante
                    backtrack(grille, m_rest, v_rest, ligne, col + 1)

                    # On retablie les valeurs de m_rest et v_rest pour tester la
                        nouvelle possibilite
                    m_rest[ligne] += val
                    v_rest[col] += val
                    grille[ligne][col] = 0

    # On initialise une grille vide
    grille_vide = [[0] * n for _ in range(n)]

    # On appelle la fonction recursive backtrack pour remplir la grille et
        tester les possibilites.
    backtrack(grille_vide, m.copy(), v.copy(), 0, 0)

    return resultats


def print_euler_to_lagrange(m, v):
    """
    Affiche dans la console Python toutes les matrices OD correspondants a m
        et v
    :param m: liste d'entiers des montees a chaque arret
    :param v: liste d'entiers des descentes a chaque arret
    """
    # On recupere les matrices
    grilles = euler_to_lagrange(m, v)

    # On les affiche
    print(str(len(grilles)) + " grilles valides trouvees")
    for i, grille in enumerate(grilles):
        print(f"Grille {i + 1} :")
        for row in grille:
```

```
            print(row)
        print()
    return


def minisation_entropie(grilles):
    """
    :param grilles: Une liste de matrices
    :return: Une matrice qui minimise l'entropie
    """

    # Calcul du nombre total de voyageurs (identique pour toutes les grilles)
    K = 0
    first_grille = grilles[0]
    for i in range(len(first_grille)):
        for j in range(len(first_grille[0])):
            K += first_grille[i][j]

    # Definition "thermodynamique" avec une entropie positive donc on veut
       minimiser l'entropie
    def calcul_entropie(grille):

        sum = 0.0
        for i in range(len(grille)):
            for j in range(len(grille[i])):
                if grille[i][j] != 0:
                    sum = sum + (grille[i][j] / K * np.log2(grille[i][j] / K))
        return sum

    entropies = [calcul_entropie(grille) for grille in grilles]
    indice_grille_min_entropie = entropies.index(min(entropies))
    return grilles[indice_grille_min_entropie]


def euler_to_best_lagrange(m, v):
    """
    :param m: Une liste d'entiers des montees
    :param v: Une liste d'entiers des descentes
    :return: Une matrice OD qui verifie m et v qui minimise l'entropie
    """
    grilles = euler_to_lagrange(m, v)
    best_grille = minisation_entropie(grilles)
    return best_grille


def affiche_matrice(M):
    """
    Affiche la matrice M dans la console Python
    :param M:
    """
```

```
    for ligne in M:
        print(str(ligne) + '\n')


if __name__ == "__main__":
    # Test sur la ligne a de metro
    noms_arrets_ligne_A = ["Poterie", "Blosne", "Triangle", "Italie", "HF",
        "Clem", "JC", "Gares", "CDG", "Repu",
                           "StAnne", "Anatole Fr", "PC", "Villejean", "Kenndy"]
    mA = [40, 37, 38, 39, 45, 36, 35, 50, 38, 50, 55, 35, 35, 32, 0]
    vA = [0, 33, 35, 38, 42, 35, 33, 52, 40, 47, 49, 38, 40, 45, 38]
    # print_euler_to_lagrange(mA, vA)

    # Test sur une ligne a 6 arrets
    noms_arrets6 = ["A1", "A2", "A3", "A4", "A5", "A6"]
    m6 = [5, 4, 6, 3, 1, 0]
    v6 = [0, 2, 4, 3, 5, 5]
    # print_euler_to_lagrange(m6, v6)
    best = minisation_entropie(euler_to_lagrange(m6, v6))
    # euler_to_graph(noms_arrets6, m6, v6)
    print("Matrice minimisant l'entropie pour ligne a 6 arrets")
    affiche_matrice(best)

    # Test sur une ligne a 5 arrets
    noms_arrets5 = ["A1", "A2", "A3", "A4", "A5"]
    m5 = [2, 3, 1, 2, 0]
    v5 = [0, 1, 2, 2, 3]
    # print_euler_to_lagrange(m5, v5)
    best2 = minisation_entropie(euler_to_lagrange(m5, v5))
    # euler_to_graph(noms_arrets5, m5, v5)
    print("Matrice minimisant l'entropie pour ligne a 5 arrets")
    affiche_matrice(best2)
```

8.3 extraction_donnees.py

```
import pandas as pd


def extraction_donnees(path, sheet, usecols, first_row):
    noms = pd.read_excel(path, sheet_name=sheet, usecols=usecols,
        skiprows=first_row - 3, nrows=1)
    numpy_noms = noms.to_numpy()

    donnes = pd.read_excel(path, sheet_name=sheet, usecols=usecols,
        skiprows=first_row - 1)
    donnes = donnes.fillna(0)
    numpy_donnees = donnes.to_numpy()
```

```
    return numpy_donnees, numpy_noms


path = "C:/Users/garan/Documents/Stage L3/Code/bus/donnees/LA_JOB.xlsx"
sheet = "LAS2_trhor15=t_0845-0859"
usecols = 'C:Q'

if __name__ == "__main__":
    donnes, noms = extraction_donnees(path, sheet, usecols, 7)
    print(donnes)
    print(noms)
```

8.4 minimisation_entropie.py

```
import scipy.optimize
import time
import random
import pickle
import os
import numpy as np
from extraction_donnees import extraction_donnees
import matplotlib.pyplot as plt
from bus import lagrange_to_euler


def affiche_matrice_propre(M):
    """
    Affiche la matrice M dans la console Python
    :param M: une matrice
    """
    # Determiner la largeur maximale d'un element du tableau pour l'alignement
    largeur_max = 9
    for ligne in M:
        # Joindre les elements de la ligne avec un espace et les aligner a
            droite selon la largeur maximale
        ligne_formatee = " ".join(f"{str(round(item, 5)):>{largeur_max}}" for
            item in ligne)
        print(ligne_formatee)
    print('\n')
    return None


def normalisation_vecteurs(m, v):
    """
    :param m: un liste d'entiers (montees)
    :param v: une liste d'entiers (descentes)
    :return: les vecteurs m et v normalises.
    """
```

```
    K = sum(m)
    invK = 1 / K
    normalized_m = [m_i * invK for m_i in m]
    normalized_v = [v_i * invK for v_i in v]
    return normalized_m, normalized_v


def normalisation_matrice(matrice):
    """
    :param matrice: une matrice de FLOAT
    :return: matrice normalisee
    """
    K = np.sum(matrice)
    invK = 1 / K

    # Copie la partie superieure de la matrice
    matrice_normalisee = np.triu(matrice, 1)

    # Normalisation de la partie superieure de la matrice
    matrice_normalisee *= invK

    return matrice_normalisee


def generation_matrice_numeros(n):
    """
    :param n: entier, taille de la matrice
    :return: une matrice avec les index de chaque case pour le vecteur
        colonne associe et des -1 pour les autres
    """
    m = np.full((n, n), -1, dtype=int)
    index = 0
    for i in range(n):
        for j in range(i + 1, n):
            m[i, j] = index
            index += 1
    return m


def vecteur_initial(m, v, matrice_numeros, n):
    """
    :param m: liste d'entier (montees)
    :param v: liste d'entier (descentes)
    :param matrice_numeros: la matrice avec les indices de vecteur
    :param n: taille des listes
    :return: le vecteur initial pour la minimisation qui correspond au
        produit des marginales (m_i * v_j = gamma_ij)
    """
    d = int((n - 1) * n / 2)
    x0 = [0] * d
```

```
    for i in range(n - 1):
        for j in range(i + 1, n):
            index = matrice_numeros[i][j]
            x0[index] = m[i] * v[j]
    return x0


def initialise_matrice_from_vect(x, n):
    """
    :param x: un vecteur d'entiers de taille d = n*(n-1)/2
    :param n: la taille de la matrice
    :return: la matrice de taille n dont les coefficients sur la partie
        superieure (stricte) droite
    correspondent au vecteur x.
    """
    matrice = np.zeros((n, n), dtype=float)
    index = 0
    for i in range(n):
        for j in range(i + 1, n):
            matrice[i][j] = x[index]
            index += 1
    return matrice


def qualite_resultat(vect_resultat, m, v, n):
    """
    :param vect_resultat: un vecteur avec les resultats d'une optimisation
    :param m: les montees normalisees
    :param v: les descentes normalisees
    :param n: la taille de la matrice
    :return: la qualite du resultat vis-a-vis du respect des contraintes
    """
    matrice_resultat = initialise_matrice_from_vect(vect_resultat, n)
    dist = 0

    # Convertir les listes m et v en tableaux NumPy pour les operations
        vectorisees
    m = np.array(m)
    v = np.array(v)

    # Calcul de la distance pour le respect des sommes sur les lignes et les
        colonnes
    somme_lignes = np.sum(matrice_resultat, axis=1)
    somme_colonnes = np.sum(matrice_resultat, axis=0)
    dist += np.sum((somme_lignes - m) ** 2) + np.sum((somme_colonnes - v) **
        2)

    # Distance pour le respect des valeurs >= 0
    vect_resultat = np.array(vect_resultat)
    dist += np.sum(vect_resultat[vect_resultat < 0] ** 2)
```

```
    return dist


# Methode Trust-Region Constrained Algorithm de Scipy

def generation_matrice_contraintes(n, matrice_numeros):
    """
    :param n: la taille de la matrice
    :param matrice_numeros: la matrice avec les numeros vecteur
    :return: la matrice de contraintes A tel que Ax = m++v
    """
    d = (n - 1) * n // 2

    # Matrice vierge pour les contraintes
    a = np.zeros((2 * n, d), dtype=int)

    # Contraintes de montees
    for i in range(n):
        for j in range(i + 1, n):
            temp_index = matrice_numeros[i][j]
            if temp_index != -1:
                a[i, temp_index] = 1

    # Contraintes de descentes
    for j in range(n):
        for i in range(j):
            temp_index = matrice_numeros[i][j]
            if temp_index != -1:
                a[n + j, temp_index] = 1

    return a


def optimisation_scipy(m, v, n):
    """
    :param m: liste des montees
    :param v: liste des descentes
    :param n: la taille des listes
    :return: le resultat de la minimisation de l'entropie pour la methode
        scipy avec contraintes.
    """
    d = (n - 1) * n // 2

    def entropie(x):
        res = np.sum(x[x > 0] * np.log(x[x > 0]))
        return res

    def jacobian_entropie(x):
        res = np.zeros_like(x)
```

```
        positive_indices = x > 0
        res[positive_indices] = np.log(x[positive_indices]) + 1
        return res

    def hessian_entropie(x):
        d = len(x)
        res = np.zeros((d, d))
        positive_indices = x > 0
        res[np.diag_indices_from(res)] = np.where(positive_indices, 1 / x, 0)
        return res

    # Matrice avec les numeros d'indices pour le passage de la vue
        vectorielle a la vue matricielle
    matrice_numeros = generation_matrice_numeros(n)

    # On cree la matrice de contraintes.
    montes_descentes = np.concatenate((m, v))
    matrice_contraintes = generation_matrice_contraintes(n, matrice_numeros)
    contraintes_montes_descentes =
        scipy.optimize.LinearConstraint(matrice_contraintes, montes_descentes,
                                                              montes_descentes)

    # On defini les bornes des valeurs ([0 ; +inf[)
    bnds = [(0, None) for _ in range(d)]

    # Vecteur initial, croisement des marginales
    x0 = vecteur_initial(m, v, matrice_numeros, n)

    # Calcul du resultat par Scipy
    resultat = scipy.optimize.minimize(entropie, x0, jac=jacobian_entropie,
        hess=hessian_entropie,
                                    method='trust-constr', bounds=bnds,
                                    constraints=contraintes_montes_descentes)
    vect_resultat = resultat.x
    qual_resultat = qualite_resultat(vect_resultat, m, v, n)

    return vect_resultat, qual_resultat


# Methode de penalisaiton
def index_ligne_colonne(index, matrice_numeros, n):
    """
    :param index: l'index d'un x_k dans le vecteur x
    :param matrice_numeros: matrice avec les index de x, -1 sinon.
    :param n: taille de la matrice
    :return: les indices i(ligne) et j (colonne) ou se trouve l'element x_i
        dans la matrice numeros
    """
    # Convertir la matrice en un array NumPy
    matrice_numeros = np.array(matrice_numeros)
```

```
    # Trouver les indices ou la valeur est egale a l'index
    result = np.where(matrice_numeros == index)

    if result[0].size > 0 and result[1].size > 0:
        return result[0][0], result[1][0]
    else:
        return -1, -1


def liste_numeros_meme_ligne(i, matrice_numeros, n):
    """
    :param i: un numero de ligne
    :param matrice_numeros: matrice avec les index de x, -1 sinon.
    :param n: taille de la matrice
    :return: la liste des indices qui sont sur la ligne i.
    """
    # Convertir la matrice en un array NumPy
    matrice_numeros = np.array(matrice_numeros)

    # Selectionner la ligne i a partir de la matrice et ignorer les valeurs -1
    ligne_i = matrice_numeros[i, i + 1:n]

    # Filtrer les valeurs -1
    res = ligne_i[ligne_i != -1]

    return list(res)


def liste_numeros_meme_colonne(j, matrice_numeros):
    """
    :param j: un numero de colonne
    :param matrice_numeros: matrice avec les index de x, -1 sinon.
    :return: la liste des indices qui sont sur la colonne j.
    """
    res = []
    for i in range(0, j):
        res.append(matrice_numeros[i][j])
    return res


def penalisation(m, v, eps, n):
    """
    :param m: liste des montees (normalise)
    :param v: liste des descentes (normalise)
    :param eps: valeur de la penalisation
    :param n: longueur des listes
    :return: la matrice OD minimisant l'entropie par la methode de
        penalisation
    """
```

```
inv_eps = 1 / eps
matrice_numeros = generation_matrice_numeros(n)

# Definition de la fonction a minimiser avec l'ajout des contraintes
def entropie_et_contraintes(x):
    res = 0
    x_pos = x[x > 0]
    res += np.sum(x_pos * np.log(x_pos))

    matrice = initialise_matrice_from_vect(x, n)
    for i in range(n - 1):
        indices_ligne = matrice_numeros[i, i + 1:n]
        sum_ligne = np.sum(x[indices_ligne[indices_ligne != -1]])
        res += inv_eps * (sum_ligne - m[i]) ** 2

    for j in range(1, n):
        indices_colonne = matrice_numeros[0:j, j]
        sum_colonne = np.sum(x[indices_colonne[indices_colonne != -1]])
        res += inv_eps * (sum_colonne - v[j]) ** 2

    res += np.sum(np.maximum(-inv_eps * x, 0) ** 2)
    return res

# Definition de la jacobienne
def jacobian_entropie_et_contraintes(x):
    res = []
    index = 0
    for x_i in x:
        val = 0
        if x_i > 0:
            val += np.log(x_i) + 1
        i, j = index_ligne_colonne(index, matrice_numeros, n)
        liste_numeros_ligne = liste_numeros_meme_ligne(i, matrice_numeros,
            n)
        somme_ligne = 0
        for k in liste_numeros_ligne:
            somme_ligne += x[k]
        somme_ligne = 2 * inv_eps * (somme_ligne - m[i])
        val += somme_ligne

        liste_numeros_colonne = liste_numeros_meme_colonne(j,
            matrice_numeros)
        somme_colonne = 0
        for k in liste_numeros_colonne:
            somme_colonne += x[k]
        somme_colonne = 2 * inv_eps * (somme_colonne - v[j])
        val += somme_colonne
        res.append(val)
        index += 1
    return res
```

```
    # Fonction scipy
    x0 = vecteur_initial(m, v, matrice_numeros, n)
    # resultat = scipy.optimize.minimize(entropie_et_contraintes, x0,
        jac=jacobian_entropie_et_contraintes)
    resultat = scipy.optimize.minimize(entropie_et_contraintes, x0)

    return resultat


def variation_epsilon(m, d, n):
    """
    :param m: liste d'entiers des montees (normalise)
    :param d: liste d'entiers des descentes (normalise
    :param n: la longueur des listes
    :return: On renvoie le vecteur resultat et sa qualite. Si pas de resultat
        on renvoie un vecteur vide et une qualite
     = -1
    """
    qualite = -1
    eps = 0.01
    resultat = penalisation(m, d, eps, n)
    res_trouve = resultat.success
    vector = resultat.x
    if res_trouve:
        qualite = qualite_resultat(vector, m, d, n)
    return vector, qualite


# Gradient a pas fixe (pas utilise)
def gradient_pas_fixe(x0, pas, itmax, erreur, fct_gradient, m, v):
    """
    :param x0: vecteur initial (taille d)
    :param pas: float
    :param itmax: nombre maximal d'iterations
    :param erreur: seuil d'erreur
    :param fct_gradient: fonction gradient associee a la fonction a minimiser
    :param m: liste d'entiers des montees (normalise)
    :param v: lsite d'entiers des descentes (normalise)
    :return: le vecteur resultat apres itmax iterations ou si l'erreur seuil
        a ete atteinte et la qualite du resutlat
    """
    # On pose l'initialisation
    res = [x0]
    iteration = 0
    qual = 0
    n = len(m)

    while iteration < itmax:
```

```
        # Si on a pas depasse le nombre maximal d'itereations, alors on
            applique l'algorithme
        xk = res[iteration]
        xk1 = np.array(xk - pas * fct_gradient(xk))
        res.append(xk1) # On ajoute l'iteration en cours a la liste des iteres.

        # On regarde si le critere d'arret d'etre suffisammment proche de la
            solution est verifie
        qual = qualite_resultat(xk1, m, v, n)
        erreurk1 = qual # On calcule l'erreur
        if erreurk1 <= erreur:
            # On est suffisament proche de la solution, on arrete l'algorithme.
            iteration = itmax
        else:
            # On est pas encore assez proche, on va faire une autre iteration.
            iteration += 1

    return res, qual


# Fonctions pour les tests :
def distance_moindres_carres(matrice1, matrice2, n):
    """
    :param matrice1: Une matrice OD
    :param matrice2: Une matrice OD
    :param n: taille des matrices
    :return: la distance entre les deux matrices coefficient par coefficient
    """
    matrice1 = np.array(matrice1)
    matrice2 = np.array(matrice2)

    # Calcul des differences
    differences = matrice1 - matrice2

    # Calcul de la somme des carres des differences au-dessus de la diagonale
        principale
    res = np.sum(differences[np.triu_indices(n, k=1)] ** 2)

    return res


def distance_entropie_relative(matriceOD, matrice2, n):
    """
    :param matriceOD: matrice origine destination (avec des 0 potentiellement)
    :param matrice2: matrice trouvee par optimisation
    :param n: taille des matrices
    :return: l'entropie relative S_matrice2(matrice_OD)
    """
    res = 0
    for i in range(n):
```

```
            for j in range(i + 1, n):
                if matrice2[i][j] > 0 and matriceOD[i][j] > 0:
                    res += matriceOD[i][j] * np.log(matriceOD[i][j] /
                        matrice2[i][j])
        return res


# Test 1 : Comparaison qualite et temps entre les deux methodes a partir de
    donnees euleriennes
def generation_vecteurs_euleriens_aleatoires(nbr_voyageurs, nbr_arrets):
    """
    :param nbr_voyageurs: nombre cumule de voyageurs sur la ligne
    :param nbr_arrets: nombre d'arrets sur la ligne
    :return: une liste correspondant aux montees, une liste correspondant aux
        descentes
    """
    montees = []
    descentes = []
    arret_courant = 0
    nbr_voyageurs_courant = 0
    personnes_restantes = nbr_voyageurs

    for i in range(0, nbr_arrets):

        # Si on est au dernier arret, personne ne monte et tout le monde doit
            descendre
        if i == nbr_arrets - 1:
            montees_i = 0
            descentes_i = nbr_voyageurs_courant

        # Si on est a l'avant-dernier arret, il faut faire monter toutes les
            personnes qui manquent
        elif i == nbr_arrets - 2:
            montees_i = personnes_restantes
            descentes_i = random.randint(0, nbr_voyageurs_courant)

        # Sinon, on fait monter un nombre aleatoire de personnes (parmi les
            nombres de personnes restantes)
        # On fait descendre un nombre aleatoire de personnes (parmi les
            voyageurs qui etaient dans le bus)
        else:
            montees_i = random.randint(0, personnes_restantes)
            descentes_i = random.randint(0, nbr_voyageurs_courant)

        montees.append(montees_i)
        descentes.append(descentes_i)
        personnes_restantes += -montees_i # On enleve les personnes montees
        nbr_voyageurs_courant = nbr_voyageurs_courant + montees_i -
            descentes_i # On met a jour le nombre de voyageurs
```

```
    return montees, descentes


def comparaison_methodes_qualite_temps_vect_aleatoires():
    liste_arrets = np.linspace(8, 10, 3)
    liste_arrets = list(map(int, liste_arrets))
    temps_eps = []
    qualite_eps = []

    temps_scipy = []
    qualite_scipy = []

    for nbr_arrets in liste_arrets:

        # On defini les listes temporaires pour faire la moyenne pour
            nbr_arrets
        temps_eps_temp = []
        qualite_eps_temp = []

        temps_scipy_temp = []
        qualite_scipy_temp = []

        for i in range(5):
            print(str(nbr_arrets) + "." + str(i))
            m, v = generation_vecteurs_euleriens_aleatoires(nbr_arrets * 8,
                nbr_arrets)
            m, v = normalisation_vecteurs(m, v)

            # On teste la methode eps
            time_start_eps = time.time()
            vect_eps, qual_eps = variation_epsilon(m, v, nbr_arrets)
            qualite_eps_temp.append(qual_eps)
            temps_eps_temp.append(time.time() - time_start_eps)

            # On teste la methode scipy
            time_start_scipy = time.time()
            vect_scipy, qual_scipy = optimisation_scipy(m, v, nbr_arrets)
            qualite_scipy_temp.append(qual_scipy)
            temps_scipy_temp.append(time.time() - time_start_scipy)

        temps_eps.append(temps_eps_temp)
        qualite_eps.append(qualite_eps_temp)

        temps_scipy.append(temps_scipy_temp)
        qualite_scipy.append(qualite_scipy_temp)

    # Temps de calcul
    plt.figure(figsize=(12, 6))
    positions_group1 = np.array(range(len(liste_arrets))) * 2.0 - 0.3
    positions_group2 = np.array(range(len(liste_arrets))) * 2.0 + 0.3
```

```
# Boxplots pour le premier groupe
plt.boxplot(temps_eps, positions=positions_group1, widths=0.4,
    patch_artist=True,
            boxprops=dict(facecolor='lightblue', color='blue'),
            medianprops=dict(color='blue'),
            whiskerprops=dict(color='blue'),
            capprops=dict(color='blue'),
            showfliers=False)

# Boxplots pour le deuxieme groupe
plt.boxplot(temps_scipy, positions=positions_group2, widths=0.4,
    patch_artist=True,
            boxprops=dict(facecolor='lightgreen', color='green'),
            medianprops=dict(color='green'),
            whiskerprops=dict(color='green'),
            capprops=dict(color='green'),
            showfliers=False)

# Ajustement des labels et des positions des axes
plt.xticks(range(0, len(liste_arrets) * 2, 2), liste_arrets)
plt.xlabel('Nombre arrets')
plt.ylabel('Temps de traitement')
plt.title('Boxplot pour le temps de traitement en fonction du nombre d
    arrets et de la methode')
plt.grid(True)
plt.legend([plt.Line2D([0], [0], color='lightblue', lw=4),
            plt.Line2D([0], [0], color='lightgreen', lw=4)],
           ['Penalisation', 'Scipy'])

plt.show()

# Temps de calcul
plt.figure(figsize=(12, 6))
positions_group1 = np.array(range(len(liste_arrets))) * 2.0 - 0.3
positions_group2 = np.array(range(len(liste_arrets))) * 2.0 + 0.3

# Boxplots pour le premier groupe
plt.boxplot(qualite_eps, positions=positions_group1, widths=0.4,
    patch_artist=True,
            boxprops=dict(facecolor='lightblue', color='blue'),
            medianprops=dict(color='blue'),
            whiskerprops=dict(color='blue'),
            capprops=dict(color='blue'),
            showfliers=False)

# Boxplots pour le deuxieme groupe
plt.boxplot(qualite_scipy, positions=positions_group2, widths=0.4,
    patch_artist=True,
            boxprops=dict(facecolor='lightgreen', color='green'),
```

```
                medianprops=dict(color='green'),
                whiskerprops=dict(color='green'),
                capprops=dict(color='green'),
                showfliers=False)

    # Ajustement des labels et des positions des axes
    plt.xticks(range(0, len(liste_arrets) * 2, 2), liste_arrets)
    plt.xlabel('Nombre arrets')
    plt.ylabel('Qualite resultat')
    plt.title('Boxplot pour la qualite du resultat en fonction du nombre d
        arrets et de la methode')
    plt.grid(True)
    plt.legend([plt.Line2D([0], [0], color='lightblue', lw=4),
                plt.Line2D([0], [0], color='lightgreen', lw=4)],
               ['Penalisation', 'Scipy'])

    plt.show()


# Test 2, 3 : Comparaison (moindres carres puis entropie relative) matrices
    OD et matrices trouvees par les deux methodes.
def comparaison_mc_entropie(matriceOD, name):
    matriceOD = normalisation_matrice(matriceOD)
    m, v = lagrange_to_euler(matriceOD)
    n = len(m)
    dossier = "C:/Users/garan/Documents/Stage
        L3/Code/bus/resultats_minimisation/"

    # Test sur le vecteur marginales croisees
    x0 = vecteur_initial(m, v, generation_matrice_numeros(n), n)
    matrice_vecteur_marginale = initialise_matrice_from_vect(x0, n)
    entropie_relative_vecteur_marginales =
        distance_entropie_relative(matriceOD, matrice_vecteur_marginale, n)
    mc_vecteur_marginales = distance_moindres_carres(matriceOD,
        matrice_vecteur_marginale, n)
    print("Pour le vecteur croisement des marginales :")
    print("Entropie relative : " + str(entropie_relative_vecteur_marginales))
    print("Distance moindres carres : " + str(mc_vecteur_marginales))
    print("------" + '\n')

    # Test des methodes de minimalisation den l'entropie

    # Test methode penalisation
    time_start = time.time()
    vect_eps, qual_eps = variation_epsilon(m, v, n)
    time_eps = time.time() - time_start

    # On verifie s'il y a bien un resultat
    if len(vect_eps) > 0:
```

```
    # On initialise la matrice a partir du vecteur trouve par la
        minimisation
    matrice_eps = initialise_matrice_from_vect(vect_eps, n)

    # On sauvegarde la matrice dans un pickle
    # Sauvegarder la matrices
    nom_fichier = name + '_eps' + '.pkl'
    with open(os.path.join(dossier, nom_fichier), 'wb') as f:
        pickle.dump(matrice_eps, f)

    # Calcul des distances (entropie relative et distance moindres carres)
    entropie_relative_eps = distance_entropie_relative(matriceOD,
        matrice_eps, n)
    distance_mc_eps = distance_moindres_carres(matriceOD, matrice_eps, n)

    # Affichage du resultat
    print("Resultats methode penalisation :")
    print("Entropie relative : " + str(entropie_relative_eps))
    print("Distance moindres carres " + str(distance_mc_eps))
    print("Duree : " + str(time_eps))
    affiche_matrice_propre(matrice_eps)
else:
    print("Pas de resultats")
    print("Duree traitement : " + str(time_eps))

# Test sur la methode scipy
time_start = time.time()
vect_scipy, qual_scipy = optimisation_scipy(m, v, n)
time_scipy = time.time() - time_start

# On verifie s'il y a un resultat
if len(vect_scipy) > 0:
    # On initialise la matrice a partir du vecteur trouve
    matrice_scipy = initialise_matrice_from_vect(vect_scipy, n)

    # Sauvegarder la matrices
    nom_fichier = name + '_scipy' + '.pkl'
    with open(os.path.join(dossier, nom_fichier), 'wb') as f:
        pickle.dump(matrice_scipy, f)

    # On calcule les distances (entropie relative et moindres carres)
    entropie_relative_scipy = distance_entropie_relative(matriceOD,
        matrice_scipy, n)
    distance_mc_scipy = distance_moindres_carres(matriceOD, matrice_scipy,
        n)

    # Affichage du resultat
    print("Resultats methode Scipy :")
    print("Entropie relative : " + str(entropie_relative_scipy))
    print("Distance moindres carres " + str(distance_mc_scipy))
```

```
        print("Duree : " + str(time_scipy))
        affiche_matrice_propre(matrice_scipy)
    else:
        print("Pas de resultats")
        print("Duree traitement : " + str(time_scipy))

    return None


# Fonction affichage des resultats d'optimisation
def affichage_resultat_opti(m, v, type='scipy'):
    """
    Affiche dans le terminal le resultat de l'optimisation avec la methode
        choisie (normalise si les vect en entree sont
    normalises)
    :param m: liste d'entiers montees (normalise)
    :param v: liste d'entiers descentes (normalise)
    :param type: scipy ou epsilon
    :return: None
    """
    time_start = time.time()
    n = len(m) # Ok
    # On recupere le resultat
    # scipy
    if type == 'scipy':
        vect_resultat, qual_resultat = optimisation_scipy(m, v, n)
    # epsilon
    else:
        vect_resultat, qual_resultat = variation_epsilon(m, v, n)

    # S'il y a un resultat
    if len(vect_resultat) > 0:
        matrice_resultat = initialise_matrice_from_vect(vect_resultat, n)
        affiche_matrice_propre(matrice_resultat)
        print("La qualite du resultat est de :" + '\n' + str(qual_resultat) +
            '\n')
        print("Duree du traitement (secondes) :")
        print(time.time() - time_start)
    else:
        print("Pas de resultat")


if __name__ == "__main__":
    # Donnees :

    # Test 0 : Resultats optimisation pour les vecteurs 5 et 6 / A comparer
        avec les resultats de la recherche exhaustive
    test0 = False
    if test0:
        matrice_entropie5 = [[0.0, 1.0, 1.0, 0.0, 0.0],
```

```
                    [0.0, 0.0, 1.0, 1.0, 1.0],
                    [0.0, 0.0, 0.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 0.0, 2.0],
                    [0.0, 0.0, 0.0, 0.0, 0.0]]
    print("Resultats ligne a 5 arrets")
    comparaison_mc_entropie(matrice_entropie5, name='matrice_entropie5')
    print(
        "-------------------------------------------------------------------------------
    matrice_entropie6 = [[0.0, 2.0, 2.0, 0.0, 0.0, 1.0],
                    [0.0, 0.0, 2.0, 1.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 2.0, 2.0, 2.0],
                    [0.0, 0.0, 0.0, 0.0, 2.0, 1.0],
                    [0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
                    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
    print("Resultats ligne a 6 arrets")
    comparaison_mc_entropie(matrice_entropie6, name='matrice_entropie6')
    print(
        "-------------------------------------------------------------------------------

# Test 1 : Comparaison de la qualite des resultats et du temps pour des
    vecteurs aleatoires.
test1 = True
if test1:
    comparaison_methodes_qualite_temps_vect_aleatoires()

# Test 2 : Comparaison par entropie relative et moindres carres avec
    donnes reelles
test2 = False
if test2:
    print("Resultats ligne A JOB 8h45-8h59" + "\n")
    path_AJOB = 'C:/Users/garan/Documents/Stage
        L3/Code/bus/donnees/LA_JOB.xlsx'
    sheet_AJOB = 'LAS2_trhor15=t_0845-0859'
    usecols_AJOB = 'C:Q'
    firstrow_AJOB = 7
    matrice_AJOB, noms_AJOB = extraction_donnees(path_AJOB, sheet_AJOB,
        usecols_AJOB, firstrow_AJOB)
    comparaison_mc_entropie(matrice_AJOB, name='matrice_AJOB')
    print("-------------------------------------------------------------------------------
    print("Resultats ligne A Samedi 8h45-8h59" + "\n")
    path_ASam = 'C:/Users/garan/Documents/Stage
        L3/Code/bus/donnees/LA_SAMEDI.xlsx'
    sheet_ASam = 'LAS2_trhor15=t_0845-0859'
    usecols_ASam = 'C:Q'
    firstrow_ASam = 7
    matrice_ASam, noms_ASam = extraction_donnees(path_ASam, sheet_ASam,
        usecols_ASam, firstrow_ASam)
    comparaison_mc_entropie(matrice_ASam, name='matrice_ASam')
    print("-------------------------------------------------------------------------------
    print("Resultats ligne B JOB 8h45-8h59" + " \n")
```

```
path_BJOB = 'C:/Users/garan/Documents/Stage
    L3/Code/bus/donnees/LB_JOB.xlsx'
sheet_BJOB = 'LBS2_trhor15=t_0845-0859'
usecols_BJOB = 'C:Q'
firstrow_BJOB = 7
matrice_BJOB, noms_BJOB = extraction_donnees(path_BJOB, sheet_BJOB,
    usecols_BJOB, firstrow_BJOB)
comparaison_mc_entropie(matrice_BJOB, name='matrice_BJOB')
print("-------------------------------------------------------------------------------
print("Resultats ligne C4 JOB 8h45-8h59" + " \n")
path_C4JOB = 'C:/Users/garan/Documents/Stage
    L3/Code/bus/donnees/LC4_JOB.xlsx'
sheet_C4JOB = 'LC4S2_trhor15=t_0845-0859'
usecols_C4JOB = 'C:AK'
firstrow_C4JOB = 7
matrice_C4JOB, noms_C4JOB = extraction_donnees(path_C4JOB,
    sheet_C4JOB, usecols_C4JOB, firstrow_C4JOB)
comparaison_mc_entropie(matrice_C4JOB, name='matrice_C4JOB')
print("-------------------------------------------------------------------------------
print("Resultats ligne C7 JOB 8h45-8h59" + " \n")
path_C7JOB = 'C:/Users/garan/Documents/Stage
    L3/Code/bus/donnees/LC7_JOB.xlsx'
sheet_C7JOB = 'LC7S1_trhor15=t_0845-0859'
usecols_C7JOB = 'C:U'
firstrow_C7JOB = 7
matrice_C7JOB, noms_C7JOB = extraction_donnees(path_C7JOB,
    sheet_C7JOB, usecols_C7JOB, firstrow_C7JOB)
comparaison_mc_entropie(matrice_C7JOB, name='matrice_C7JOB')
print("-------------------------------------------------------------------------------
```

8.5 Recette de gâteau au chocolat et au sarrasin

Sur demande des membres de l'équipe d'Analyse numérique et équations aux dérivées partielles, voici la recette du délicieux gâteau au chocolat et sarrasin. La recette est une adaptation de celle du gâteau au chocolat des écoliers disponible sur Marmiton.

Ingrédients pour environ 8 personnes :

- Chocolat noir : 200g
- Beurre **salé** : 125g
- Œufs : 4
- Sucre : 200g
- Levure chimique : 1 sachet (11g)
- Farine de sarrasin (blé noir) : 100g

Étapes de préparation :

1. Préchauffer le four à 180°C
2. Dans un premier saladier, casser le chocolat et couper le beurre en morceaux et les faire fondre ensemble au micro-ondes jusqu'à ce que le mélange soit lisse.

3. Dans un second saladier, mélanger les œufs et le sucre ensemble jusqu'à ce que le mélange ait bien blanchi (cela peut prendre quelques minutes au batteur électrique).
4. Ajouter la farine et la levure au mélange oeufs/sucre et mélanger jusqu'à obtention d'un mélange homogène.
5. Ajouter le mélange chocolat/beurre à la préparation et mélanger à nouveau.
6. Avec une noisette de beurre, beurrer un moule.
7. Verser l'appareil à gâteau dans le moule et enfourner toujours à 180°C.
8. Selon le moule choisi, le temps de cuisson varie, c'est prêt lorsque la lame d'un couteau ressort propre (mais ceux qui préfèrent les fondants peuvent arrêter la cuisson un peu avant).
9. Bon appétit.