Optimisation des flux aériens

48604

TIPE - 2018-19

« Le trafic aérien mondial va encore doubler d’ici à 2037 »

— L’ASSOCIATION DES COMPAGNIES AÉRIENNES INTERNTATIONALES

« Le trafic aérien mondial va encore doubler d'ici à 2037 »

— L'ASSOCIATION DES COMPAGNIES AÉRIENNES INTERNTATIONALES

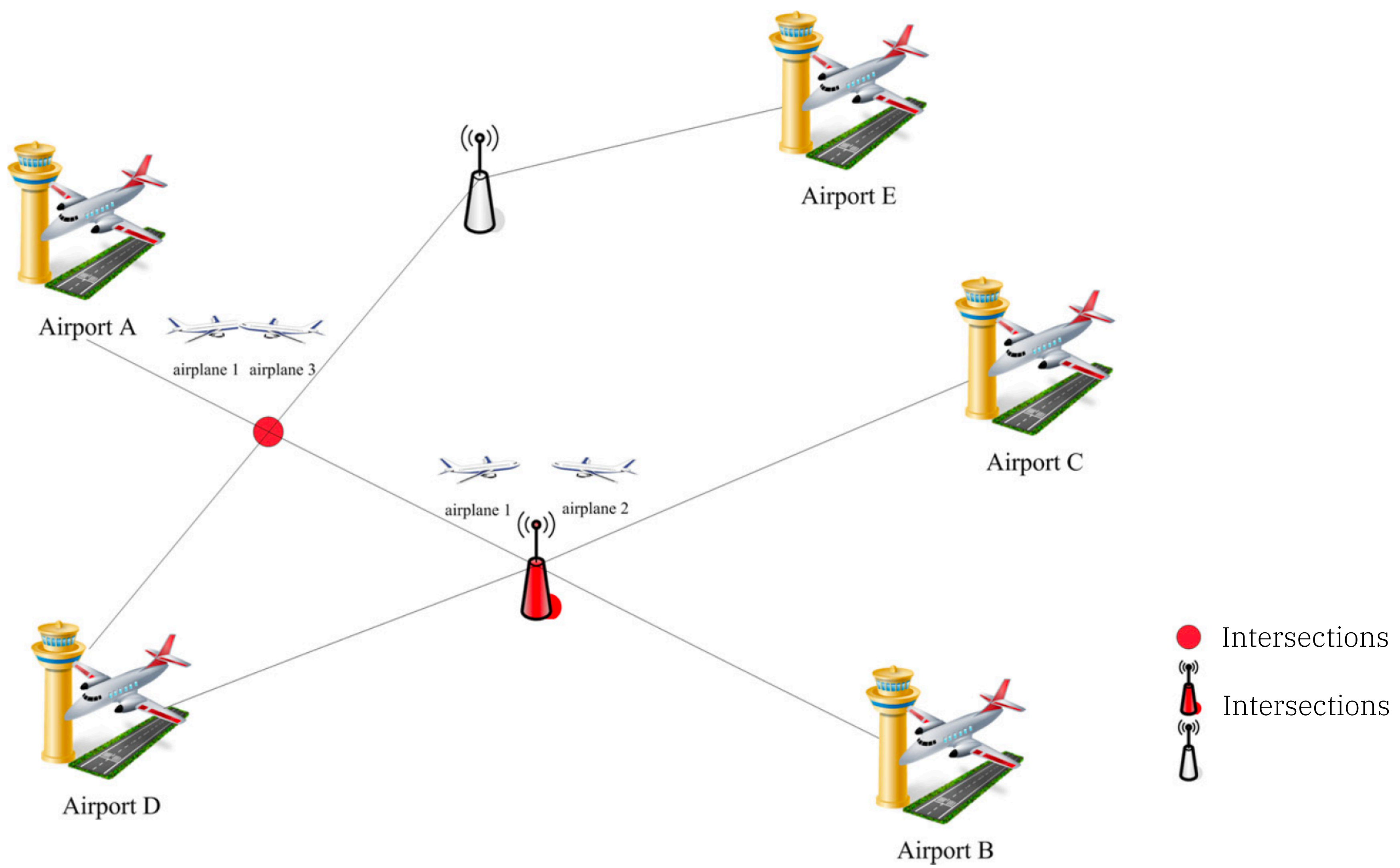


Fig. 1. Un exemple de réseau de routes aériennes

Problématique

Au vu de l'importance du réseau de routes aériennes dans la gestion du trafic aérien,

Problématique

Au vu de l'importance du réseau de routes aériennes dans la gestion du trafic aérien,

Proposer un **algorithme** construisant le **réseau de routes aériennes optimal** pour la France, satisfaisant à la fois les **compagnies aériennes** et le **contrôle du trafic aérien**.

Problématique

Au vu de l'importance du réseau de routes aériennes dans la gestion du trafic aérien,

Proposer un **algorithme** construisant le **réseau de routes aériennes optimal** pour la France, satisfaisant à la fois les **compagnies aériennes** et le **contrôle du trafic aérien**.

Plan

Problématique

Au vu de l'importance du réseau de routes aériennes dans la gestion du trafic aérien,

Proposer un **algorithme** construisant le **réseau de routes aériennes optimal** pour la France, satisfaisant à la fois les **compagnies aériennes** et le **contrôle du trafic aérien**.

Plan

Problématique

Au vu de l'importance du réseau de routes aériennes dans la gestion du trafic aérien,

> Proposer un **algorithme** construisant le **réseau de routes aériennes optimal** pour la France, satisfaisant à la fois les **compagnies aériennes** et le **contrôle du trafic aérien**.

Plan

Problématique

Au vu de l'importance du réseau de routes aériennes dans la gestion du trafic aérien,

Proposer un **algorithme** construisant le **réseau de routes aériennes optimal** pour la France, satisfaisant à la fois les **compagnies aériennes** et le **contrôle du trafic aérien**.

Plan

Remarque : On se place en **2D**, en ne considérant ni la montée ni la descente des avions.

I. Conception du réseau de routes aériennes

Construction d'un réseau de routes aériennes, uniquement à partir des ***positions des aéroports*** *et de la* ***demande de trafic aérien.***

Réseau de routes aériennes

Définition (réseau de routes aériennes)
Un réseau de routes aériennes est défini comme un **graphe orienté** (S, A) où
$S = \{ \textit{aéroports et intersections} \}$ et $A = \{ \textit{routes} \}$.

Réseau de routes aériennes

Définition (réseau de routes aériennes)
Un réseau de routes aériennes est défini comme un **graphe orienté** (S, A) où $S = \{\textit{ aéroports et intersections }\}$ et $A = \{\textit{ routes }\}$.

Remarque (description d'un réseau de routes aériennes)

- Les **aéroports** et les **intersections** sont repérés par leur **position**.
- Les **routes** sont représentées par une **matrice** M (matrice d'adjacence) :

$$M \in M_{|S|}(\{0,1\}) \quad m_{ij} = \begin{cases} 1 & \text{si } (i, j) \text{ est une route} \\ 0 & \text{sinon} \end{cases}$$

Réseau de routes aériennes

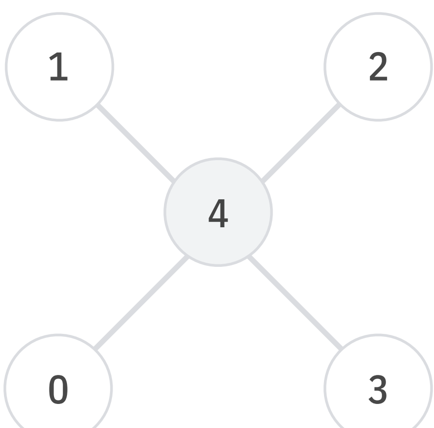
Définition (réseau de routes aériennes)
Un réseau de routes aériennes est défini comme un **graphe orienté** (S, A) où $S = \{ \textit{aéroports et intersections} \}$ et $A = \{ \textit{routes} \}$.

Remarque (description d'un réseau de routes aériennes)

- Les **aéroports** et les **intersections** sont repérés par leur **position**.
- Les **routes** sont représentées par une **matrice** M (matrice d'adjacence) :

$$M \in M_{|S|}(\{0,1\}) \quad m_{ij} = \begin{cases} 1 & \text{si } (i,j) \text{ est une route} \\ 0 & \text{sinon} \end{cases}$$

Exemple




```
pos_aeroports = [[x₀, y₀],[x₁, y₁],[x₂, y₂],[x₃, y₃]]

pos_intersections = [[x₄, y₄]]
```

Et, `reseau_routes` =

0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	1	1	1	0

Paramètres

`pos_aeroports`

`demande_trafic_aérien`

Paramètres

`pos_aeroports`

`demande_trafic_aérien`

	Abscisse	Ordonnée
CDG	667079.03	6878961.72
ORY	654368.815	6846800.311
NCE	1039439.967	6294712.859
MRS	878952.491	6262572.594
LYS	861749.256	6516589.866
TLS	568424.501	6282575.068
MPL	777541.40	6276100.35
GNB	882397.22	6476061.00
NTE	350936.371	6683464.008
BOD	519794.435	6417108.796
SXB	1041569.89	6836626.22

Liste des positions d'aéroports.
(List of list of floats)

Paramètres

`pos_aeroports`

	Abscisse	Ordonnée
CDG	667079.03	6878961.72
ORY	654368.815	6846800.311
NCE	1039439.967	6294712.859
MRS	878952.491	6262572.594
LYS	861749.256	6516589.866
TLS	568424.501	6282575.068
MPL	777541.40	6276100.35
GNB	882397.22	6476061.00
NTE	350936.371	6683464.008
BOD	519794.435	6417108.796
SXB	1041569.89	6836626.22

Liste des positions d'aéroports.
(List of list of floats)

`demande_trafic_aérien`

	CDG	ORY	NCE	MRS	LYS	TLS	MPL	GNB	NTE	BOD	SXB
CDG	0	0	10	10	5	9	4	5	6	6	4
ORY	0	0	40	24	4	41	17	4	5	18	0
NCE	12	38	0	0	5	10	0	6	5	10	9
MRS	8	28	0	0	4	3	0	3	14	12	8
LYS	5	3	6	3	0	13	0	0	14	14	5
TLS	9	41	10	3	13	0	0	13	12	0	10
MPL	4	20	0	0	0	0	0	0	6	13	1
GNB	5	4	6	3	0	13	0	0	15	14	5
NTE	6	5	5	14	14	12	6	15	0	4	8
BOD	7	19	10	12	14	0	13	14	4	0	3
SXB	4	0	5	8	9	10	1	5	8	3	0

4 avions se déplacent de CDG à SXB (par jour)

Matrice de la demande de trafic (par jour) .
(Ndarray of int)

Algorithme

Pour construire le réseau de routes, on fait d'abord l'hypothèse que **toutes les trajectoires des avions sont des segments** reliant deux aéroports.

Algorithme

Pour construire le réseau de routes, on fait d'abord l'hypothèse que **toutes les trajectoires des avions sont des segments** reliant deux aéroports.

`pos_aeroports`

↓

`demande_trafic_aerien`

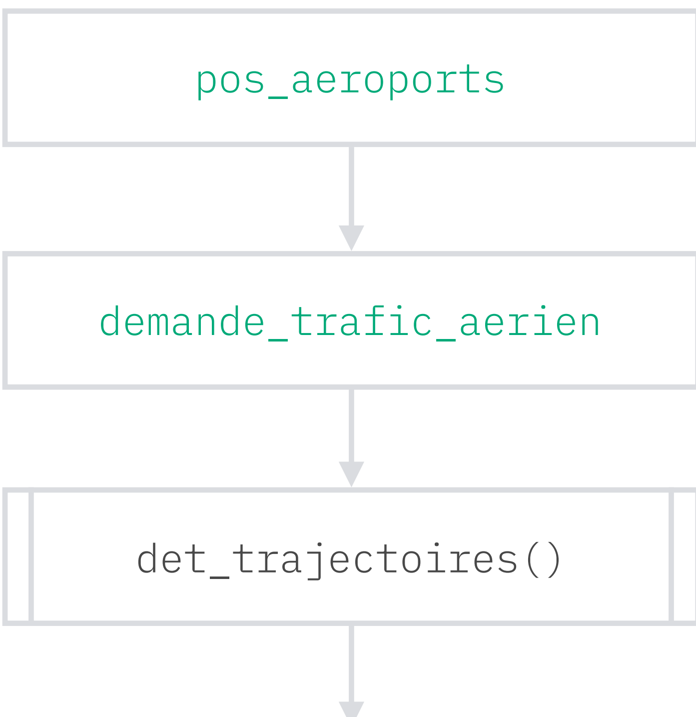
↓

```
pos_aeroports = [[x₀, y₀],[x₁, y₁],[x₂, y₂],[x₃, y₃]]
```

$$\text{demande} = \begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \\ 5 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}$$

Algorithme

Pour construire le réseau de routes, on fait d'abord l'hypothèse que **toutes les trajectoires des avions sont des segments** reliant deux aéroports.

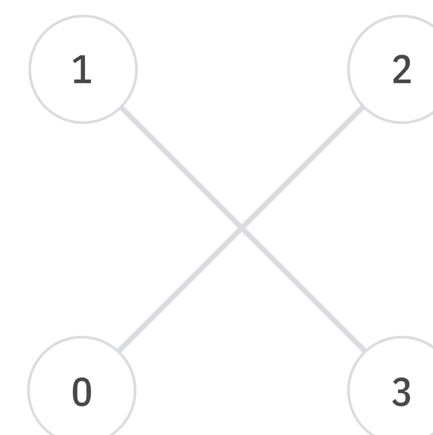



```
pos_aeroports = [[x₀, y₀],[x₁, y₁],[x₂, y₂],[x₃, y₃]]
```

demande =

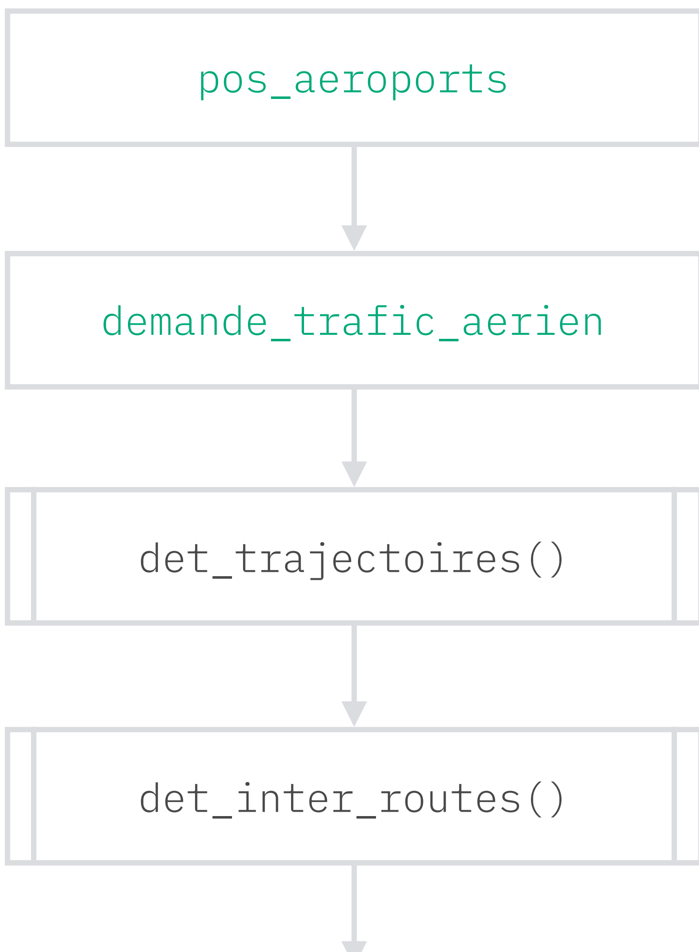
0	0	5	0
0	0	0	9
5	0	0	0
0	9	0	0

```
trajectoires = [[2, 0],[3, 1]]
```



Algorithme

Pour construire le réseau de routes, on fait d'abord l'hypothèse que **toutes les trajectoires des avions sont des segments** reliant deux aéroports.



pos_aeroports = [[x₀, y₀],[x₁, y₁],[x₂, y₂],[x₃, y₃]]

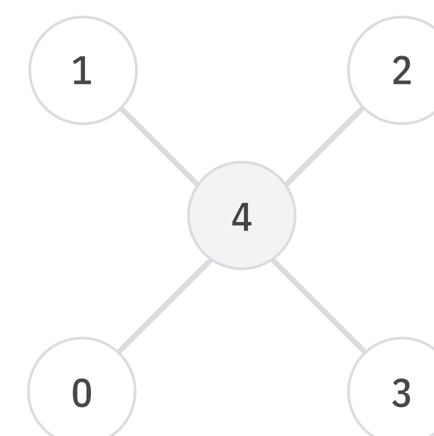
demande =

0	0	5	0
0	0	0	9
5	0	0	0
0	9	0	0

trajectoires = [[2, 0],[3, 1]]

pos_intersections = [[x4, y4]]

routes = [[0, 4],[1, 4],[2, 4],[3, 4]]



Algorithme

Pour construire le réseau de routes, on fait d’abord l’hypothèse que **toutes les trajectoires des avions sont des segments** reliant deux aéroports.

`pos_aeroports`

```
pos_aeroports = [[x0, y0],[x1, y1],[x2, y2],[x3, y3]]
```

`demande_trafic_aerien`

demande =

0	0	5	0
0	0	0	9
5	0	0	0
0	9	0	0

`det_trajectoires()`

```
trajectoires = [[2, 0],[3, 1]]
```

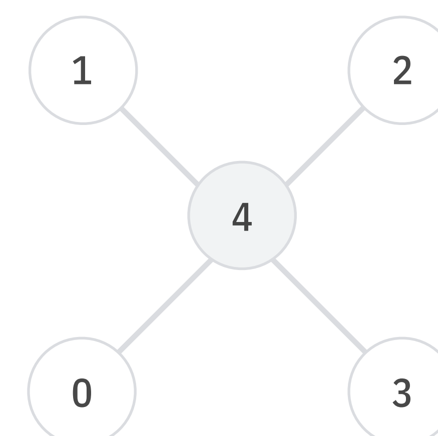
`det_inter_routes()`

```
pos_intersections = [[x4, y4]]
routes = [[0, 4],[1, 4],[2, 4],[3, 4]]
```

`rep_routes()`

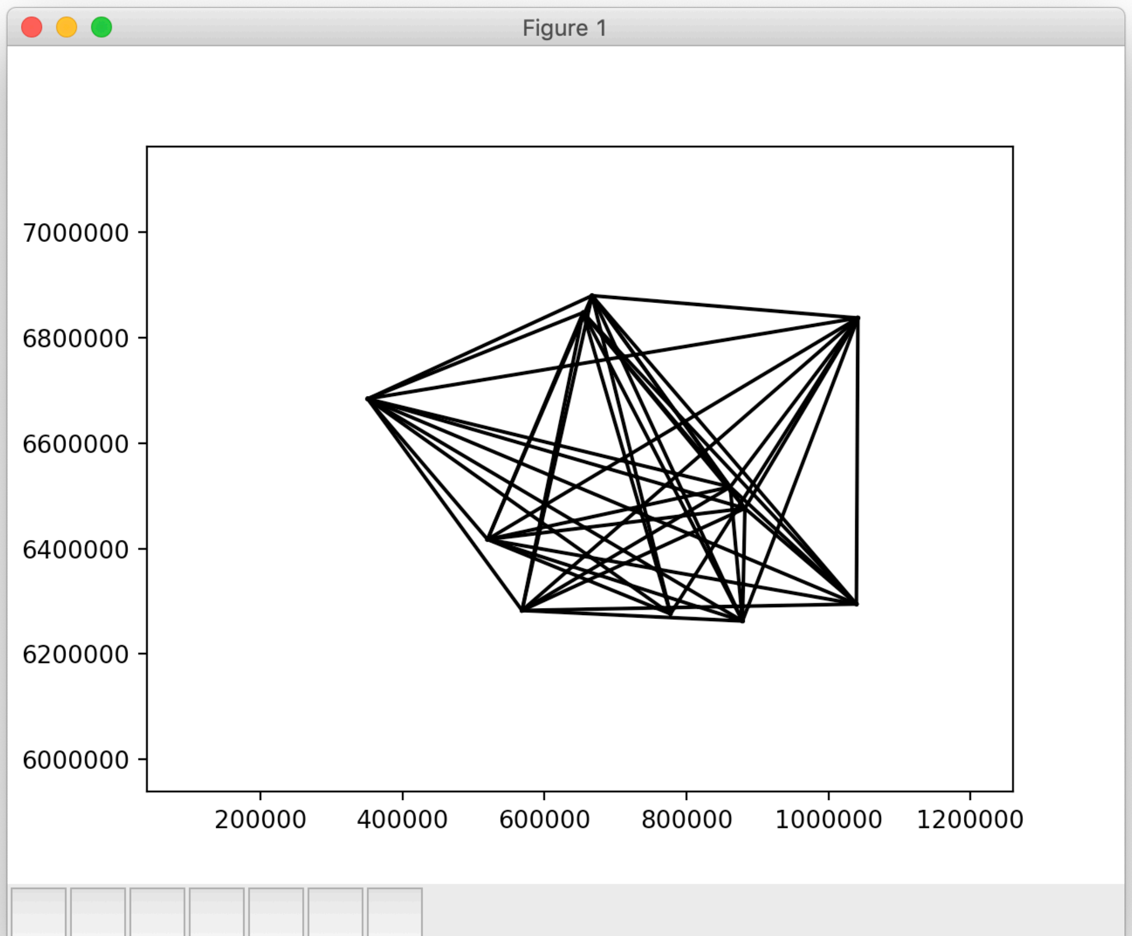
reseau_routes =

0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	1	1	1	0



Résultat

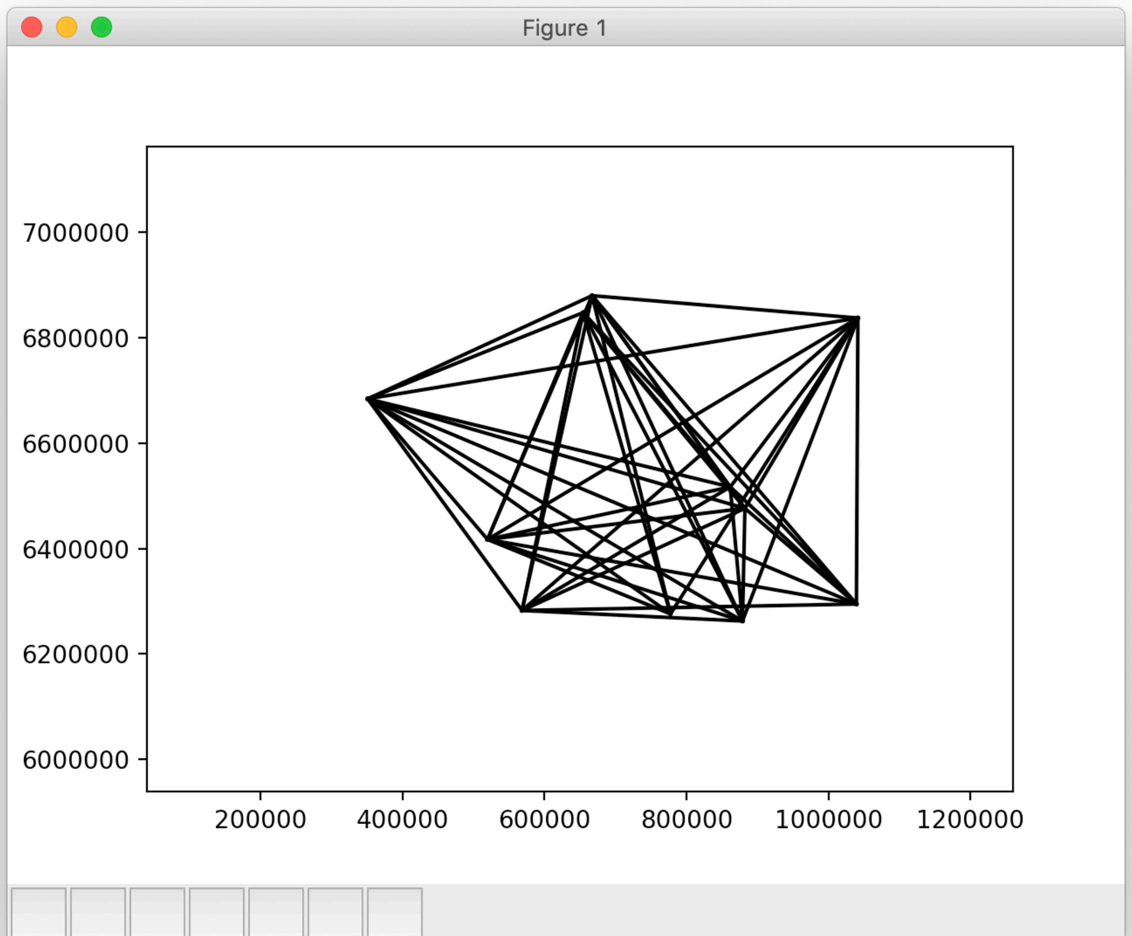
A partir de **11 aéroports** et la **demande de trafic aérien** donnant le nombre d'avions directs circulant entre deux aéroports (par jour),



On obtient **762 intersections**.

Résultat

A partir de **11 aéroports** et la **demande de trafic aérien** donnant le nombre d’avions directs circulant entre deux aéroports (par jour),



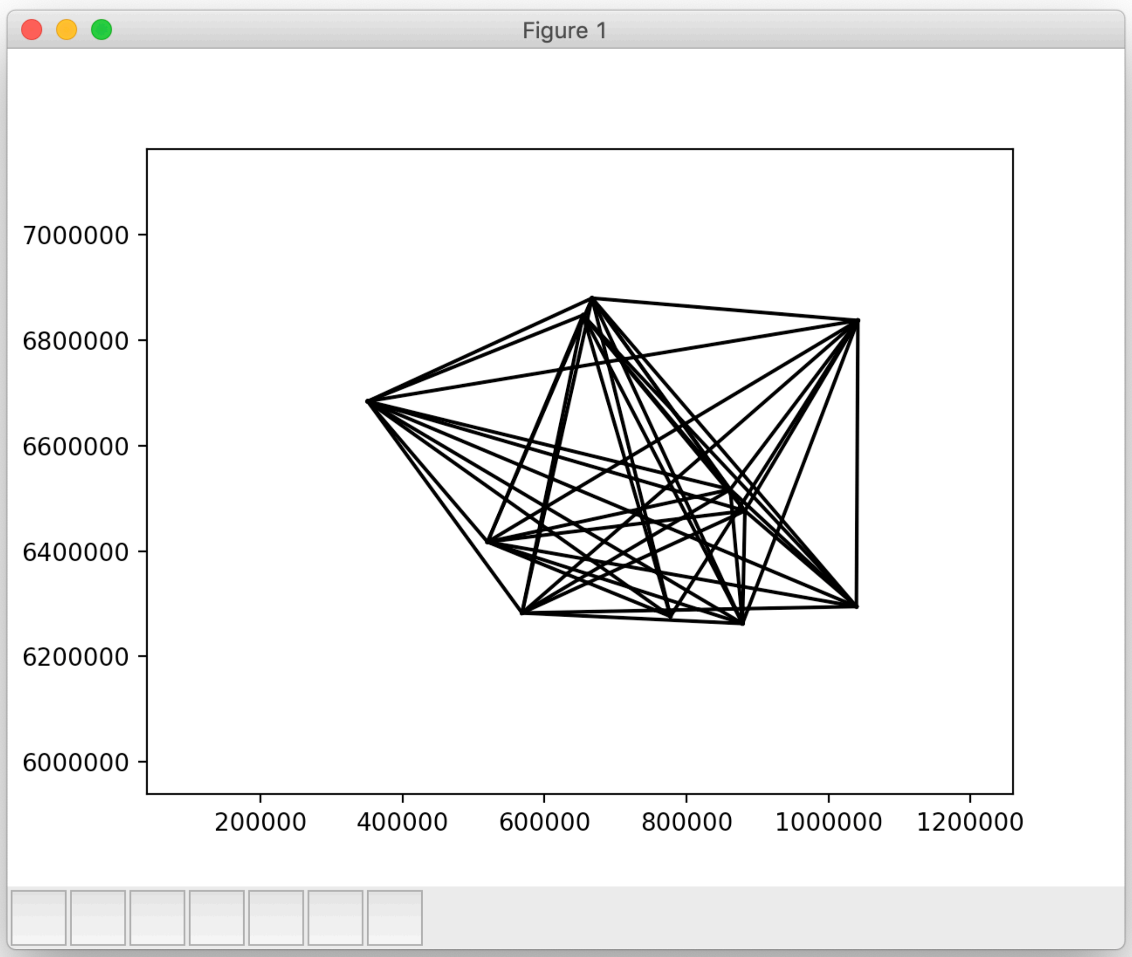
On obtient **762 intersections.**

Remarques

- La distance totale parcourue est minimale (trajectoires = segments).
- Un **grand nombre d’intersections** observé.
- Certaines intersections sont **trop proches** les unes des autres.

Résultat

A partir de **11 aéroports** et la **demande de trafic aérien** donnant le nombre d'avions directs circulant entre deux aéroports (par jour),



On obtient **762 intersections.**

Remarques

- La distance totale parcourue est minimale (trajectoires = segments).
- Un **grand nombre d'intersections** observé.
- Certaines intersections sont **trop proches** les unes des autres.

Ainsi, bien que la solution proposée soit **optimale pour les compagnies aériennes**, elle ne l'est **pas pour le contrôle du trafic aérien**. (cf III.)

II. Regroupement d'intersections

Amélioration du réseau de routes aériennes construit précedemment, par l'augmentation de la surface des secteurs associées aux intersections.

Secteurs associés aux intersections

Un **contrôleur du trafic aérien** résout les conflits **au niveau d’une intersection**, modifiant **localement les trajectoires** des avions se situant **dans la partie de l’espace aérien** dont il s’occupe.

Secteurs associés aux intersections

Un **contrôleur du trafic aérien** résout les conflits **au niveau d'une intersection**, modifiant **localement les trajectoires** des avions se situant **dans la partie de l'espace aérien** dont il s'occupe.

Définition (secteur)
Pour chaque intersection ***i***, on définit son **secteur associé S_i** comme l'ensemble des points du plan dont l'intersection la plus proche est cette dernière.

A savoir, $S_i = \{p \in \mathbb{R}^2 \mid \forall j \in I \ \|p - i\| \leq \|p - j\|\}$

Secteurs associés aux intersections

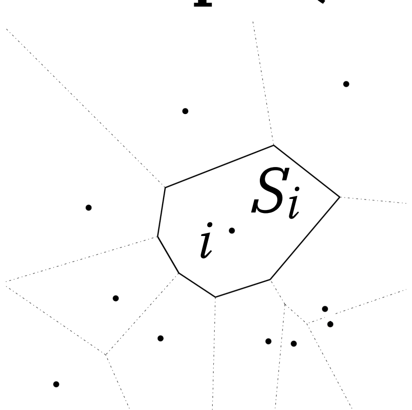
Un **contrôleur du trafic aérien** résout les conflits **au niveau d'une intersection**, modifiant **localement les trajectoires** des avions se situant **dans la partie de l'espace aérien** dont il s'occupe.

Définition (secteur)

Pour chaque intersection $\boldsymbol{i}$, on définit son **secteur associé** $\boldsymbol{S_i}$ comme l'ensemble des points du plan dont l'intersection la plus proche est cette dernière.

A savoir, $S_i = \{p \in \mathbb{R}^2 \mid \forall j \in I \ \|p - i\| \leq \|p - j\|\}$

Exemple (secteur)



Ainsi, le contrôleur de ce secteur modifie les trajectoires des avions se situant **dans le secteur** $\boldsymbol{S_i}$.

↳ Augmentation de l'aire des secteurs (réduction du nombre d'intersections)

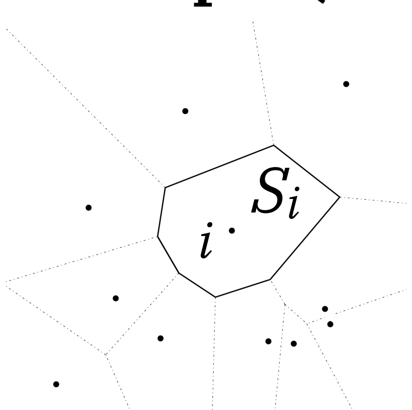
Secteurs associés aux intersections

Un **contrôleur du trafic aérien** résout les conflits **au niveau d'une intersection**, modifiant **localement les trajectoires** des avions se situant **dans la partie de l'espace aérien** dont il s'occupe.

Définition (secteur)
Pour chaque intersection ***i***, on définit son **secteur associé** S_i comme l'ensemble des points du plan dont l'intersection la plus proche est cette dernière.

A savoir, $S_i = \{p \in \mathbb{R}^2 \mid \forall j \in I \; \|p - i\| \leq \|p - j\|\}$

Exemple (secteur)



Ainsi, le contrôleur de ce secteur modifie les trajectoires des avions se situant **dans le secteur S_i.**

↳ Augmentation de l'aire des secteurs (réduction du nombre d'intersections)

Remarque
L'ensemble de ces secteurs forment **le diagramme de Voronoï** associé aux intersections. (créé à l'aide de `Scipy.spatial Voronoi`)

Réduction du nombre d’intersections

Algorithme (k-moyennes)

(nombre de partitions)

`K`

Entrée

`pos_intersections`

(intersections)

Réduction du nombre d’intersections

Algorithme (k-moyennes)

(nombre de partitions)

`K`

Entrée

`pos_intersections`

(intersections)

Partitionnement des intersections en *k* ensembles de façon optimale

`par_intersections()`

(à l’aide de `sklearn.cluster KMeans`)

Réduction du nombre d’intersections

Algorithme (k-moyennes)

(nombre de partitions)

`K`

Entrée

`pos_intersections`

(intersections)

Partitionnement des intersections en *k* ensembles de façon optimale

`par_intersections()`

(à l’aide de `sklearn.cluster KMeans`)

(barycentres des partitions)

`pos_barycentres`

Sortie

`eti_inter_bar`

(Partitions)

Réduction du nombre d'intersections

Algorithme (k-moyennes)

(nombre de partitions)

`K`

Entrée

`pos_intersections`

(intersections)

Partitionnement des intersections en *k* ensembles de façon optimale

`par_intersections()`

(à l'aide de `sklearn.cluster KMeans`)

(barycentres des partitions)

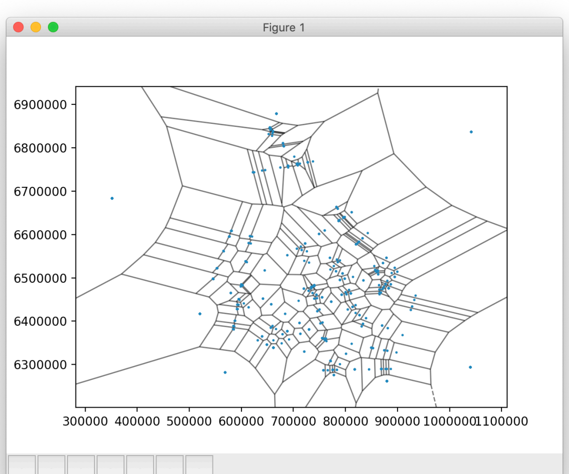
`pos_barycentres`

Sortie

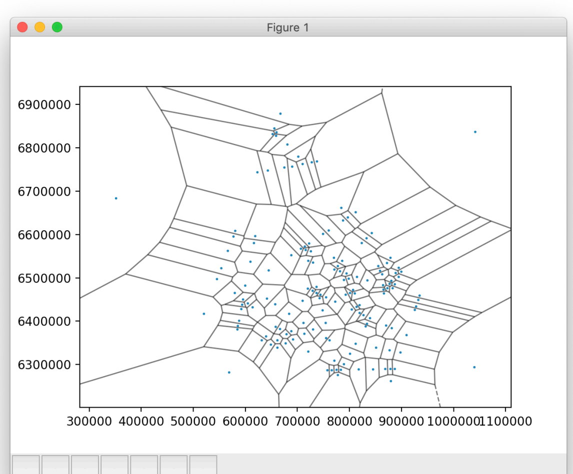
`eti_inter_bar`

(Partitions)

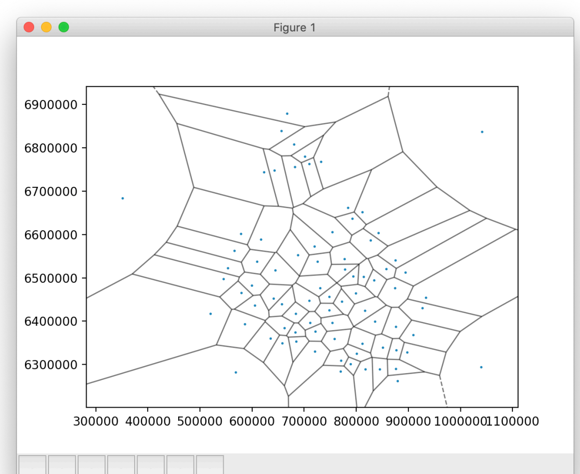
Exemples (diagramme de Voronoï des barycentres pour différents k)



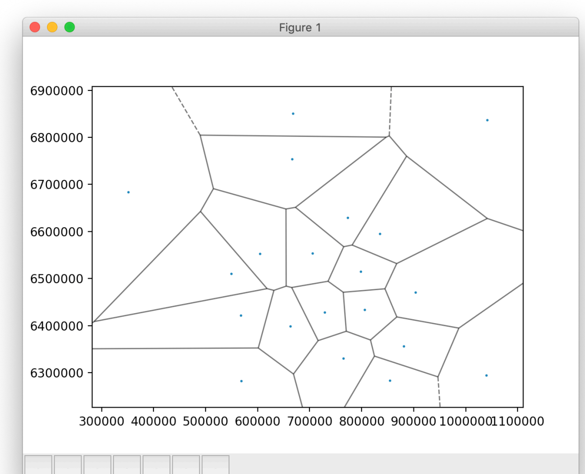
k = 200



k = 150



k = 100



k = 20

Choix de k (densité d'un secteur)

N'ayant pas accès au **nombre d'intersections idéal** fourni par les spécialistes :

Choix de k (densité d'un secteur)

N'ayant pas accès au **nombre d'intersections idéal** fourni par les spécialistes :

Définition (densité d'un secteur)

Pour chaque secteur S_i, on définit sa densité comme : $D_i = \frac{f_i}{A_i}$

- f_i : flux d'avions associé au secteur i.
- A_i : aire du secteur i.

Choix de k (densité d'un secteur)

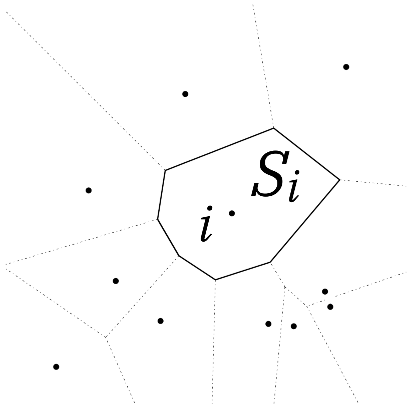
N'ayant pas accès au **nombre d'intersections idéal** fourni par les spécialistes :

Définition (densité d'un secteur)

Pour chaque secteur S_i, on définit sa densité comme : $D_i = \frac{f_i}{A_i}$

- f_i : flux d'avions associé au secteur i.
- A_i : aire du secteur i.

Calcul (aire d'un secteur)



Le secteur S_i n'est rien d'autre qu'un polygone.

Ainsi, l'aire A_i du secteur S_i s'obtient à l'aide de la formule mathématique suivante :

$$A_i = \frac{1}{2} \cdot \sum_{k=0}^{n-1} |x_i y_{i+1} - x_{i+1} yi|$$

Où les sommets du secteur, $P_0, P_1, \ldots, P_n = P_0$ sont rangés dans le sens horaire ou trigonométrique avec $P_i = (x_i, y_i)$

Choix de k (densité d'un secteur)

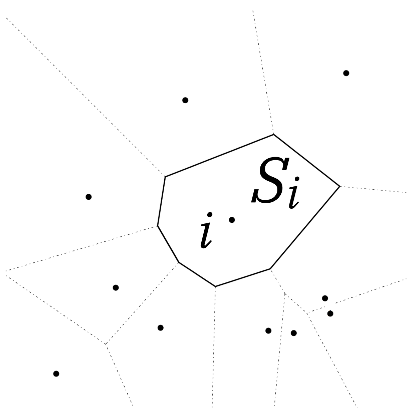
N'ayant pas accès au **nombre d'intersections idéal** fourni par les spécialistes :

Définition (densité d'un secteur)

Pour chaque secteur S_i, on définit sa densité comme : $D_i = \frac{f_i}{A_i}$

- f_i : flux d'avions associé au secteur i.
- A_i : aire du secteur i.

Calcul (aire d'un secteur)



Le secteur S_i n'est rien d'autre qu'un polygone.

Ainsi, l'aire A_i du secteur S_i s'obtient à l'aide de la formule mathématique suivante :

$$A_i = \frac{1}{2} \cdot \sum_{k=0}^{n-1} |x_i y_{i+1} - x_{i+1} yi|$$

Où les sommets du secteur, $P_0, P_1, \dots, P_n = P_0$ sont rangés dans le sens horaire ou trigonométrique avec $P_i = (x_i, y_i)$

Ainsi pour chaque secteur :

`sommets_secteur` → `tri_sommets_secteur()` → `cal_aire_secteur()`

`[[0, 0],[1, 1],[1, 0],[0, 1]]` `[[0, 0],[1, 0],[1, 1],[0, 1]]` `1.`

Choix de k (flux d’un secteur)

On suppose que toutes les **trajectoires des avions** correspondent aux **chemins les plus courts** dans le réseau de routes aériennes.

Choix de k (flux d'un secteur)

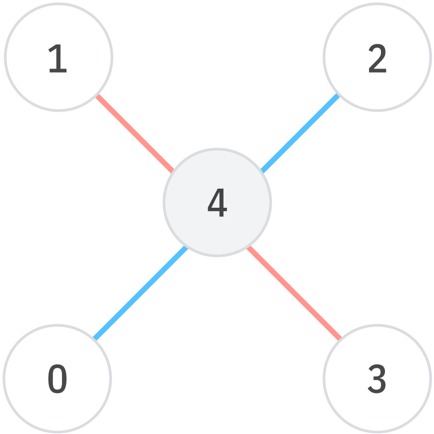
On suppose que toutes les **trajectoires des avions** correspondent aux **chemins les plus courts** dans le réseau de routes aériennes.

demande_trafic_aerien

reseau_routes

$$\texttt{demande\_trafic aérien} = \begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \\ 5 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}$$

$$\texttt{reseau\_routes} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$



Choix de k (flux d’un secteur)

On suppose que toutes les **trajectoires des avions** correspondent aux **chemins les plus courts** dans le réseau de routes aériennes.

demande_trafic_aerien

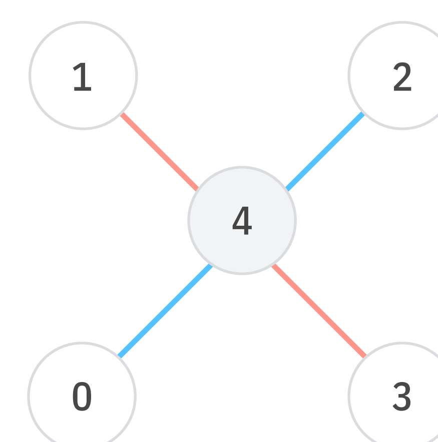
reseau_routes

det_dist_routes()

$$\texttt{demande\_trafic aérien} = \begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \\ 5 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}$$

$$\texttt{reseau\_routes} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$\texttt{distances\_routes} = \begin{bmatrix} 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 1. & 1. & 1. & 1. & 1. \end{bmatrix}$$



Choix de k (flux d’un secteur)

On suppose que toutes les **trajectoires des avions** correspondent aux **chemins les plus courts** dans le réseau de routes aériennes.

demande_trafic_aerien

reseau_routes

det_dist_routes()

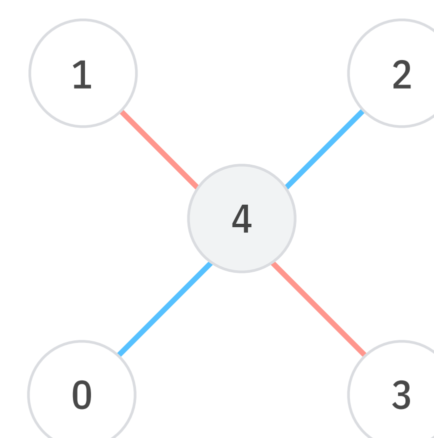
det_flux_routes()

demande_trafic aérien = $\begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \\ 5 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}$

reseau_routes = $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$

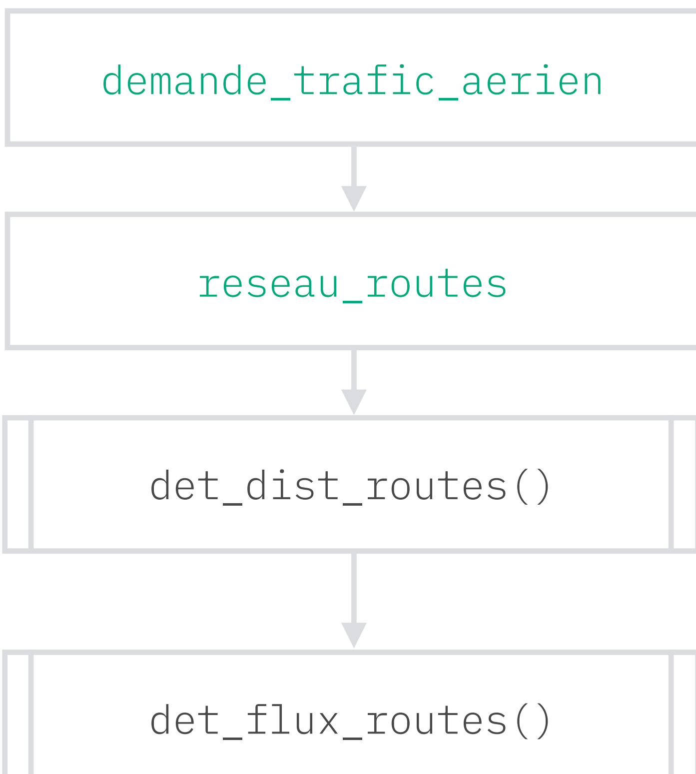
distances_routes = $\begin{bmatrix} 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 1. & 1. & 1. & 1. & 1. \end{bmatrix}$

flux_routes = $\begin{bmatrix} 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 9 \\ 5 & 9 & 5 & 9 & 0 \end{bmatrix}$



Choix de k (flux d'un secteur)

On suppose que toutes les **trajectoires des avions** correspondent aux **chemins les plus courts** dans le réseau de routes aériennes.



demande_trafic aérien =

0	0	5	0
0	0	0	9
5	0	0	0
0	9	0	0

reseau_routes =

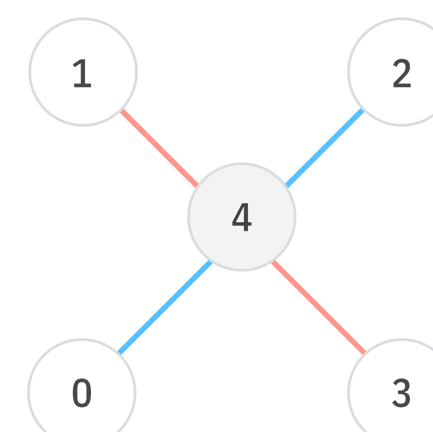
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	1	1	1	0

distances_routes =

0.	0.	0.	0.	1.
0.	0.	0.	0.	1.
0.	0.	0.	0.	1.
0.	0.	0.	0.	1.
1.	1.	1.	1.	1.

flux_routes =

0	0	0	0	5
0	0	0	0	9
0	0	0	0	5
0	0	0	0	9
5	9	5	9	0



Remarque : L'algorithme de **Floyd-Warshall** est utilisé dans `det_flux_routes()` pour déterminer les chemins les plus courts dans le réseau de routes aériennes.

Choix de k (flux d'un secteur)

On suppose que toutes les **trajectoires des avions** correspondent aux **chemins les plus courts** dans le réseau de routes aériennes.

`demande_trafic_aerien` → `reseau_routes` → `det_dist_routes()` → `det_flux_routes()` → `det_flux_secteurs()`

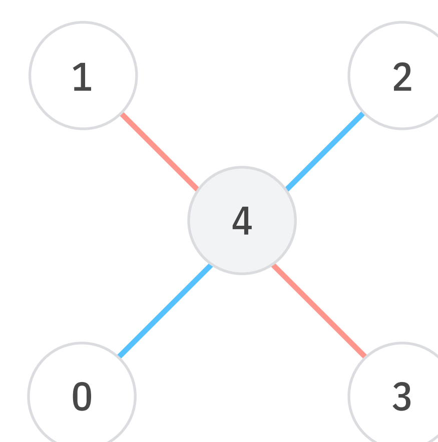
demande_trafic aérien = $\begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \\ 5 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}$

reseau_routes = $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$

distances_routes = $\begin{bmatrix} 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. \\ 1. & 1. & 1. & 1. & 1. \end{bmatrix}$

flux_routes = $\begin{bmatrix} 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 9 \\ 5 & 9 & 5 & 9 & 0 \end{bmatrix}$

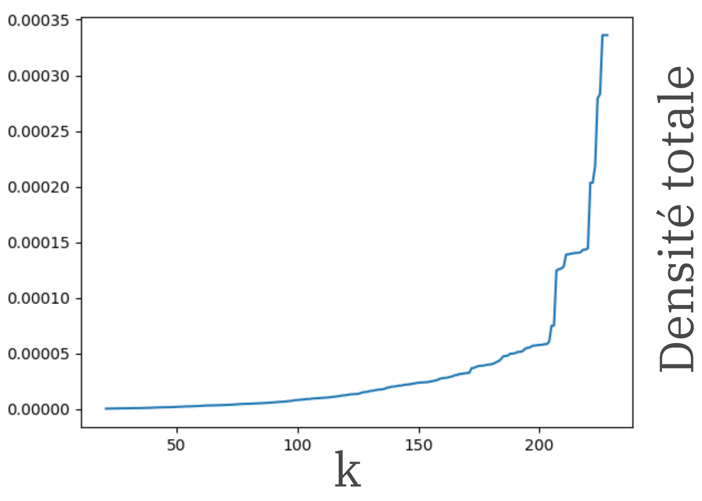
flux_secteurs = [5 + 9 + 5 + 9] = [28]



Remarque : L'algorithme de **Floyd-Warshall** est utilisé dans `det_flux_routes()` pour déterminer les chemins les plus courts dans le réseau de routes aériennes.

Choix de k (Résultat)

Considérons la **densité totale** (somme des densités) en **fonction de k** :

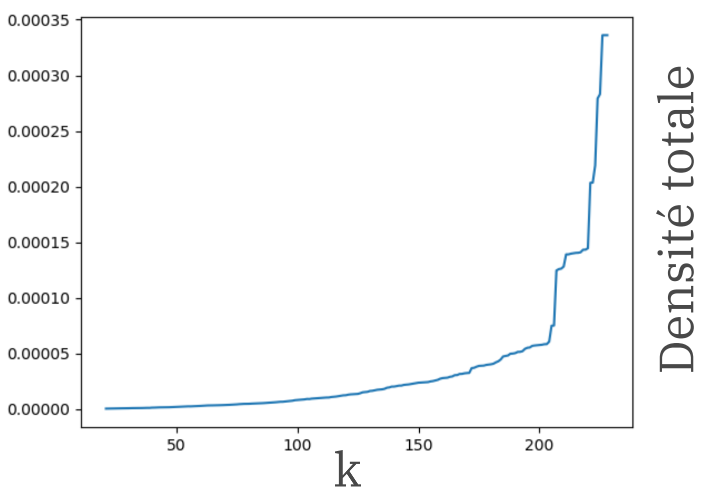


- k↘ ⇒ densité totale↘ strictement

↳ Regroupement d’intersections ⇒ amélioration du réseau de routes aériennes.

Choix de k (Résultat)

Considérons la **densité totale** (somme des densités) en **fonction de k** :



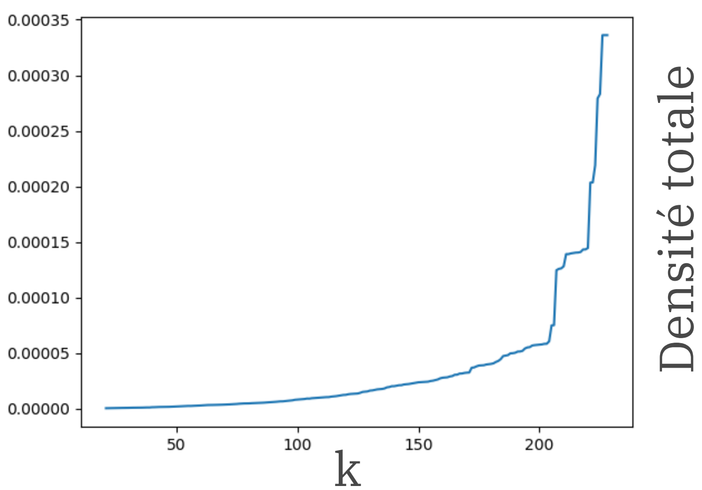
- k↘ ⇒ densité totale↘ strictement

↳ Regroupement d'intersections ⇒ amélioration du réseau de routes aériennes.

Mais, le réseau de routes avec **1 seule intersection insatisfaisante** (un contrôleur humain ne peut pas s'occuper d'un grand nombre d'avions simultanément).
↳ La grandeur densité **insuffisante pour choisir k**.

Choix de k (Résultat)

Considérons la **densité totale** (somme des densités) en **fonction de k** :



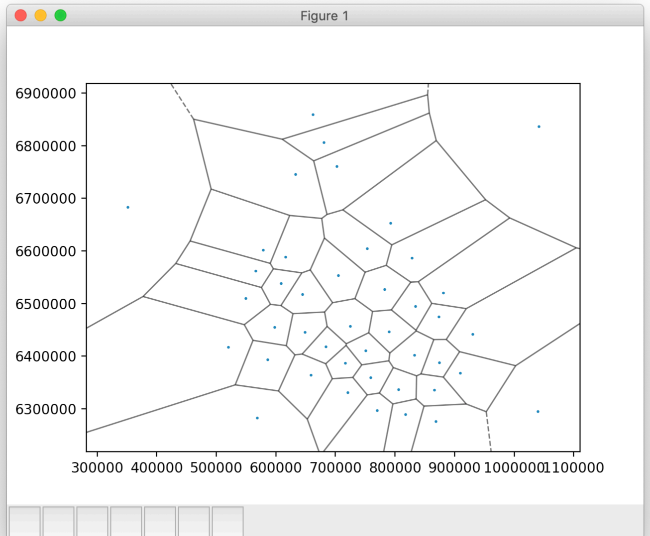
- $k \searrow \Rightarrow$ densité totale $\searrow$ strictement

↳ Regroupement d’intersections ⇒ amélioration du réseau de routes aériennes.

Mais, le réseau de routes avec **1 seule intersection insatisfaisante** (un contrôleur humain ne peut pas s’occuper d’un grand nombre d’avions simultanément).
↳ La grandeur densité **insuffisante pour choisir k.**

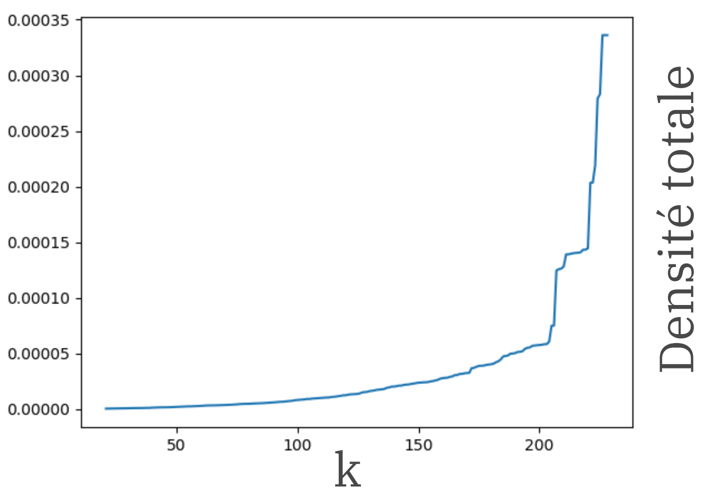
En choisissant `seuil = 800`, on obtient à l’aide de `det_nbr_inter_optimal()` :

```
k = 43
```



Choix de k (Résultat)

Considérons la **densité totale** (somme des densités) en **fonction de k** :



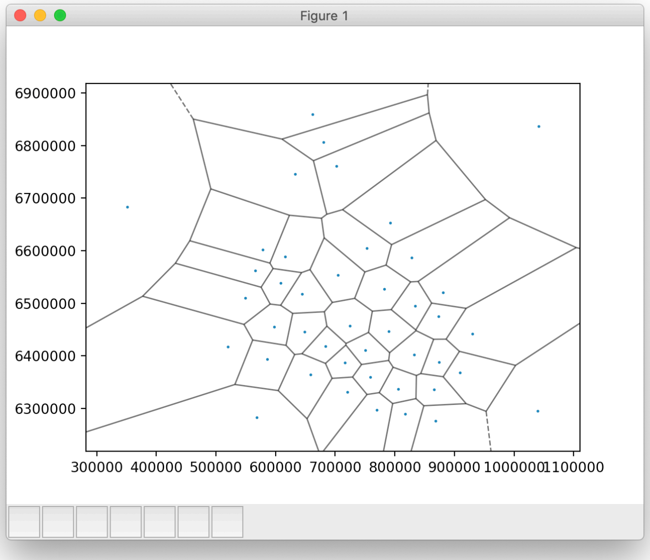
- k↘ ⇒ densité totale↘ strictement

↳ Regroupement d'intersections ⇒ amélioration du réseau de routes aériennes.

Mais, le réseau de routes avec **1 seule intersection insatisfaisante** (un contrôleur humain ne peut pas s'occuper d'un grand nombre d'avions simultanément).
↳ La grandeur densité **insuffisante pour choisir k.**

En choisissant `seuil = 800`, on obtient à l'aide de `det_nbr_inter_optimal()` :

```
k = 43
```



Enfin, on en déduit le nouveau réseau de routes aériennes à l'aide de la fonction `det_res_rout_réduit()`.

III.Optimisation du réseau de routes aériennes

Ajustement des positions d'intersections, minimisant le coût des compagnies aériennes et la dangérosité du réseau de routes.

Coût total des compagnies aériennes

Les compagnies aériennes cherchent à **minimiser les coûts**, donc la distance totale parcourue.

Coût total des compagnies aériennes

Les compagnies aériennes cherchent à **minimiser les coûts**, donc la distance totale parcourue.

Définition (coût total des compagnies aériennes)

Il est défini comme :

$$C = \sum_{i=0}^{n-1} \sum_{j=0, i \neq j}^{n-1} f_{ij} \cdot d_{ij} \qquad (1)$$

- n : nombre de sommets.
- f_{ij} : flux d'avions associé à la route (i, j)
- d_{ij} : distance associée à la route (i, j).

Coût total des compagnies aériennes

Les compagnies aériennes cherchent à **minimiser les coûts**, donc la distance totale parcourue.

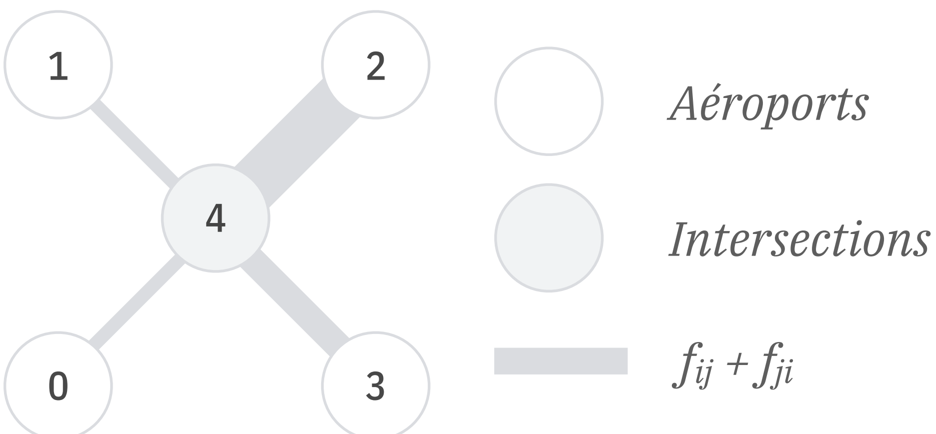
Définition (coût total des compagnies aériennes)

Il est défini comme :

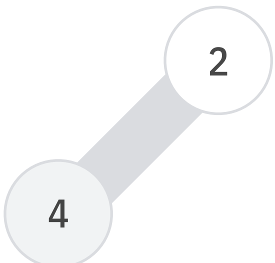
$$C = \sum_{i=0}^{n-1} \sum_{j=0, i \neq j}^{n-1} f_{ij} \cdot d_{ij} \quad (1)$$

- n : nombre de sommets.
- f_{ij} : flux d'avions associé à la route (i, j)
- d_{ij} : distance associée à la route (i, j).

Exemple



Aux routes $(4, 2)$ et $(2, 4)$ représentées par :



On associe le coût: $(f_{42} + f_{24}) \times d_{42}$.

Par sommation, on obtient le **coût total des compagnies aériennes**.

Coût total des compagnies aériennes

Les compagnies aériennes cherchent à **minimiser les coûts**, donc la distance totale parcourue.

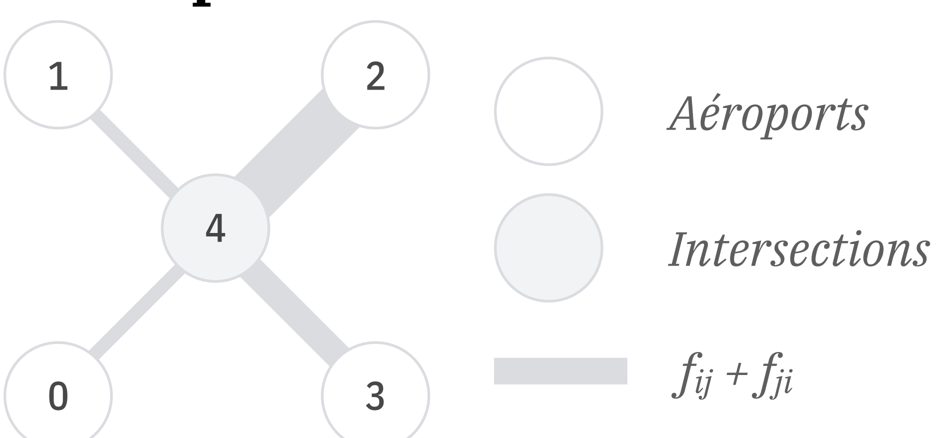
Définition (coût total des compagnies aériennes)

Il est défini comme :

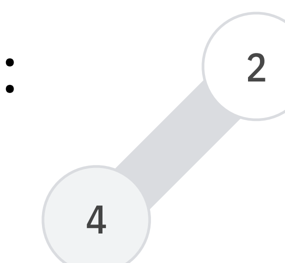
$$C = \sum_{i=0}^{n-1} \sum_{j=0, i \neq j}^{n-1} f_{ij} \cdot d_{ij} \quad (1)$$

- n : nombre de sommets.
- f_{ij} : flux d'avions associé à la route (i, j)
- d_{ij} : distance associée à la route (i, j).

Exemple



Aux routes $(4, 2)$ et $(2, 4)$ représentées par :



On associe le coût: $(f_{42} + f_{24}) \times d_{42}$.

Par sommation, on obtient le **coût total des compagnies aériennes**.

Remarque (Coût du réseau initial)

Le réseau de routes aériennes initial a été construit **à partir de trajectoires directes**.

↳ Il **minimise** le coût total des compagnies aériennes.

Dangérosité du réseau de routes

Le contrôle du trafic aérien cherche à **maximiser la sécurité**, donc à éviter les situations complexes à gérer pour le contrôleur du trafic aérien :

Dangérosité du réseau de routes

Le contrôle du trafic aérien cherche à **maximiser la sécurité**, donc à éviter les situations complexes à gérer pour le contrôleur du trafic aérien :

Définition (dangérosité du réseau de routes)

Il est défini comme :

$$D = \sum_{k=0}^{p-1} \sum_{i \in V_k} \sum_{j \in V_k,\ i \neq j} \frac{f_{ki} \cdot f_{kj}}{\cos \frac{\alpha_{ij}^k}{2}} \qquad (2)$$

- p : nombre d'intersections.
- V_k : ensemble des voisins de k.
- f_{ki} : flux d'avions de (k, i).
- α_{ij} : angle intérieur entre (i, k) et (j, k)

Dangérosité du réseau de routes

Le contrôle du trafic aérien cherche à **maximiser la sécurité**, donc à éviter les situations complexes à gérer pour le contrôleur du trafic aérien :

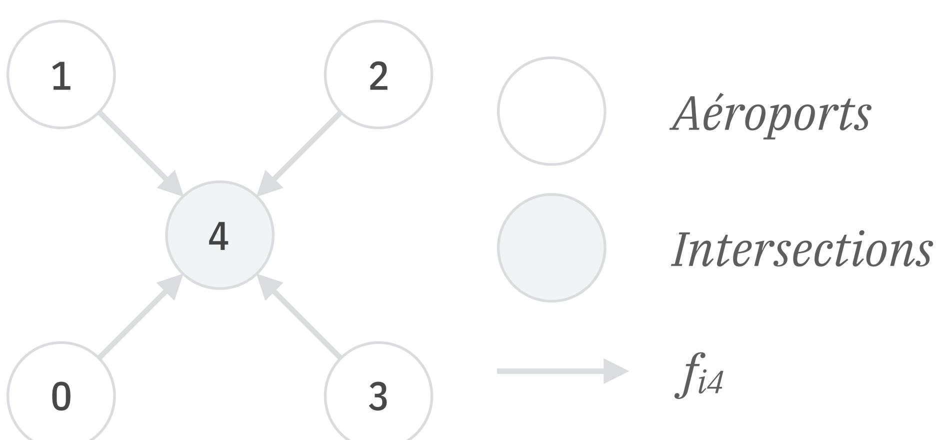
Définition (dangérosité du réseau de routes)

Il est défini comme :

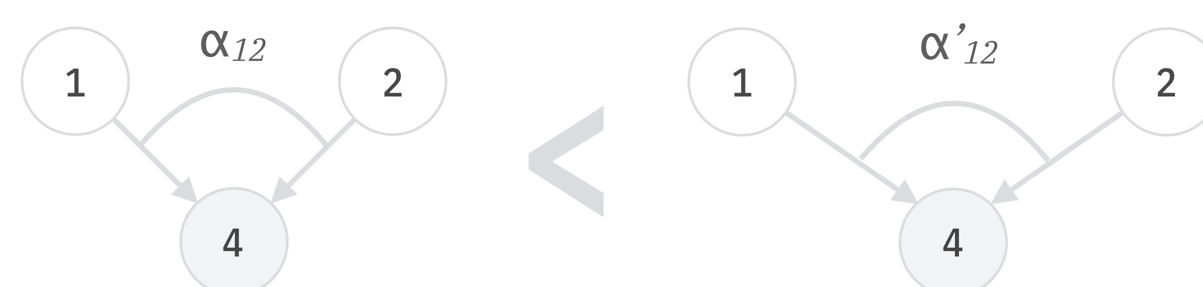
$$D = \sum_{k=0}^{p-1} \sum_{i \in V_k} \sum_{j \in V_k,\ i \neq j} \frac{f_{ki} \cdot f_{kj}}{\cos \frac{\alpha_{ij}^k}{2}} \quad (2)$$

- p : nombre d'intersections.
- V_k : ensemble des voisins de k.
- f_{ki} : flux d'avions de (k, i).
- α_{ij} : angle intérieur entre (i, k) et (j, k)

Exemple



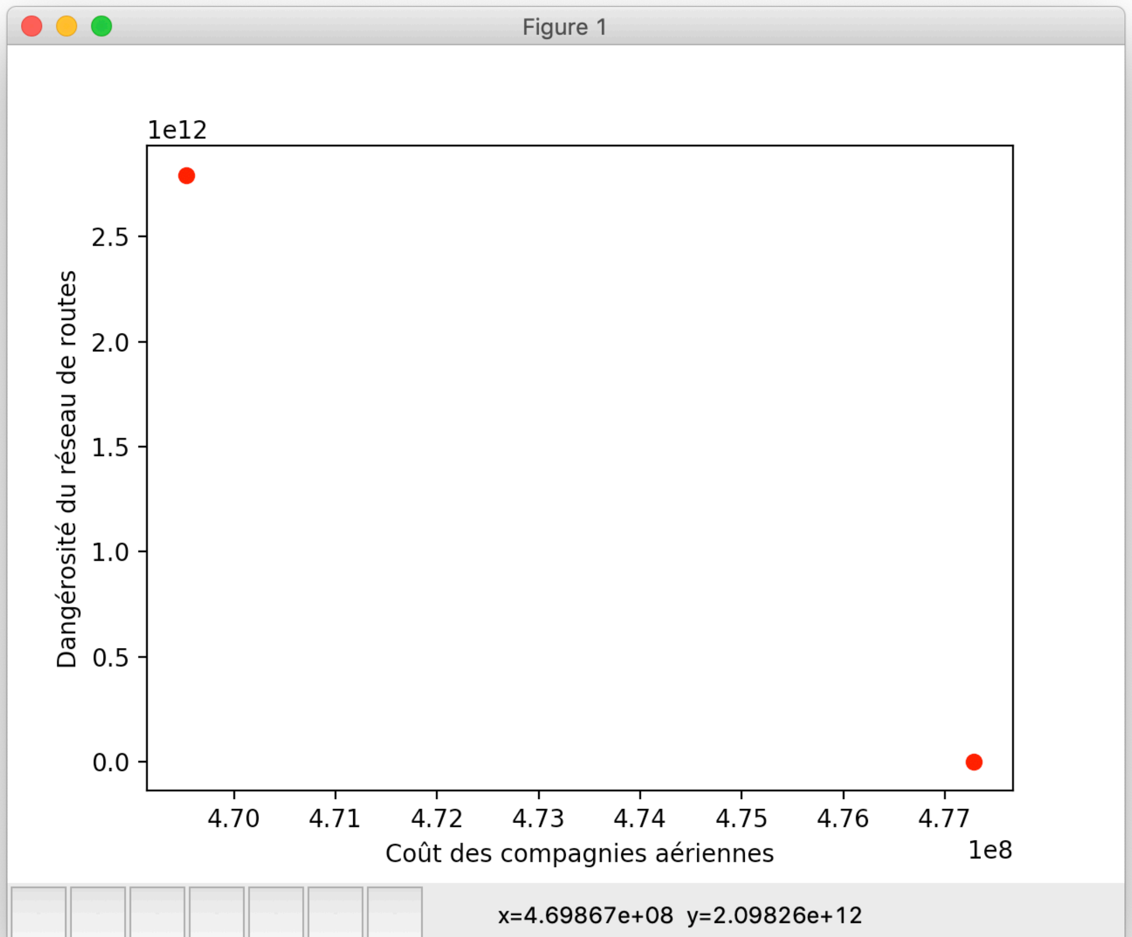
On cherche : angle intérieur ↗ ⇒ dangérosité ↗



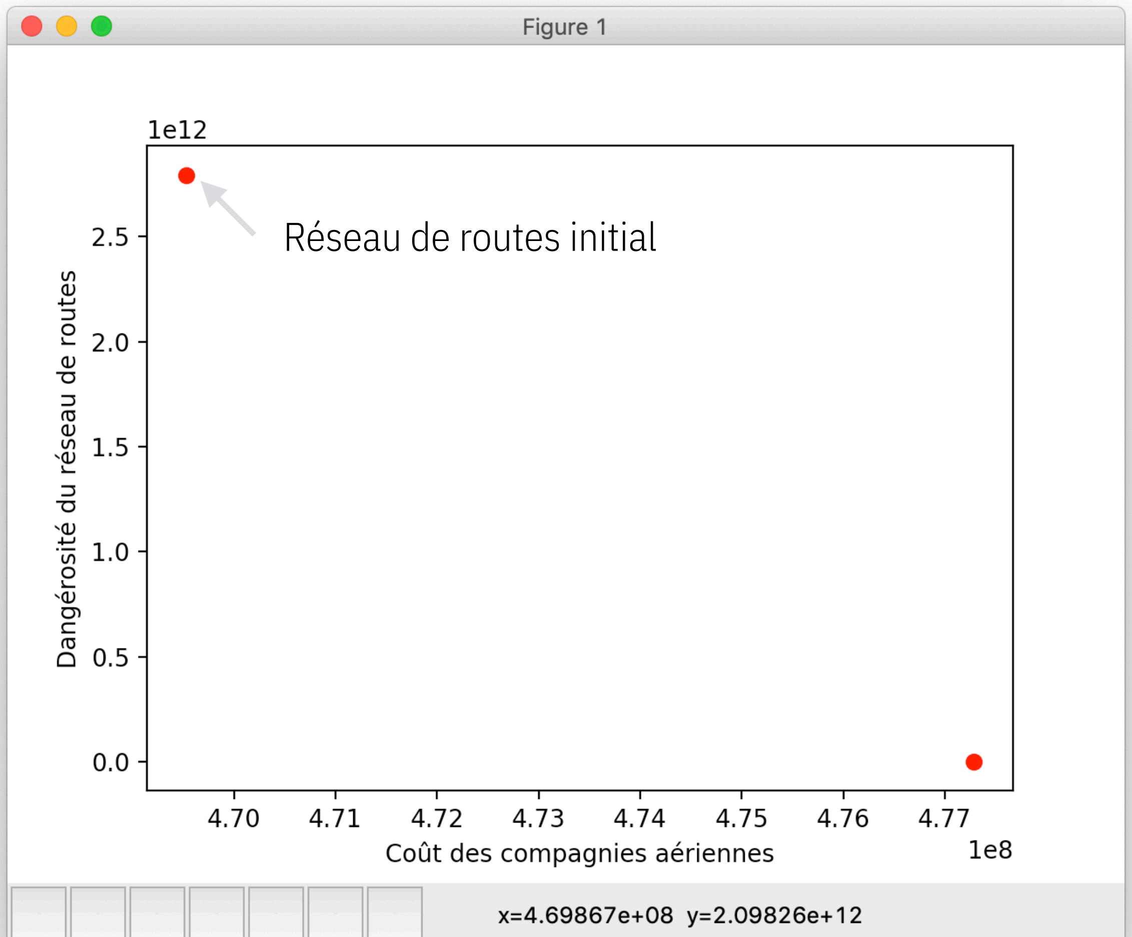
On définit alors pour **chaque paire de routes,** $\dfrac{f_{ki} \cdot f_{kj}}{\cos \frac{\alpha_{ij}^k}{2}}$

Enfin, on en déduit **en sommant** la dangérosité du réseau de routes aériennes.

Comparaison des réseaux de routes

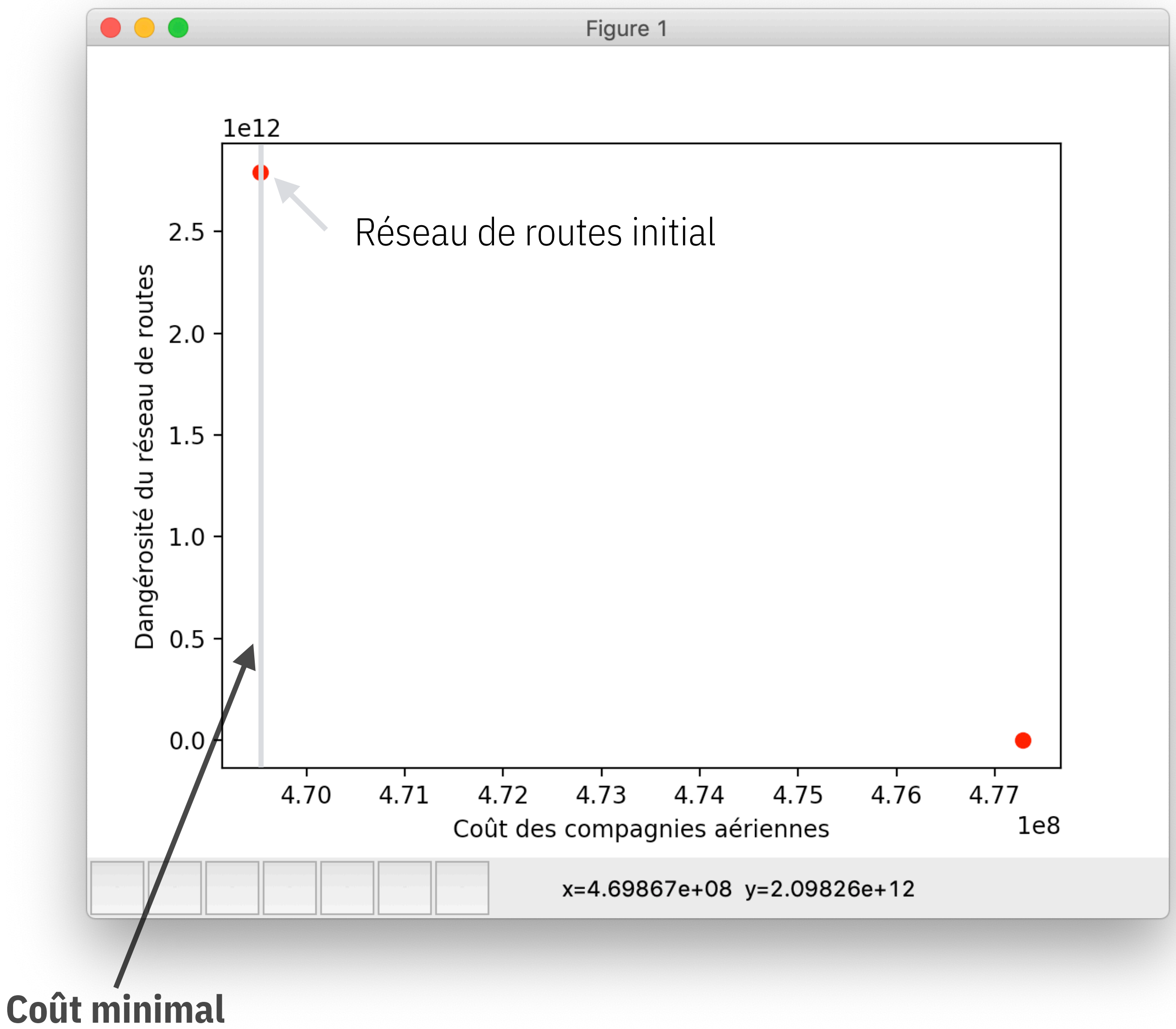


Comparaison des réseaux de routes



Réseau de routes **initial** construit à partir **de trajectoires directes** (I).

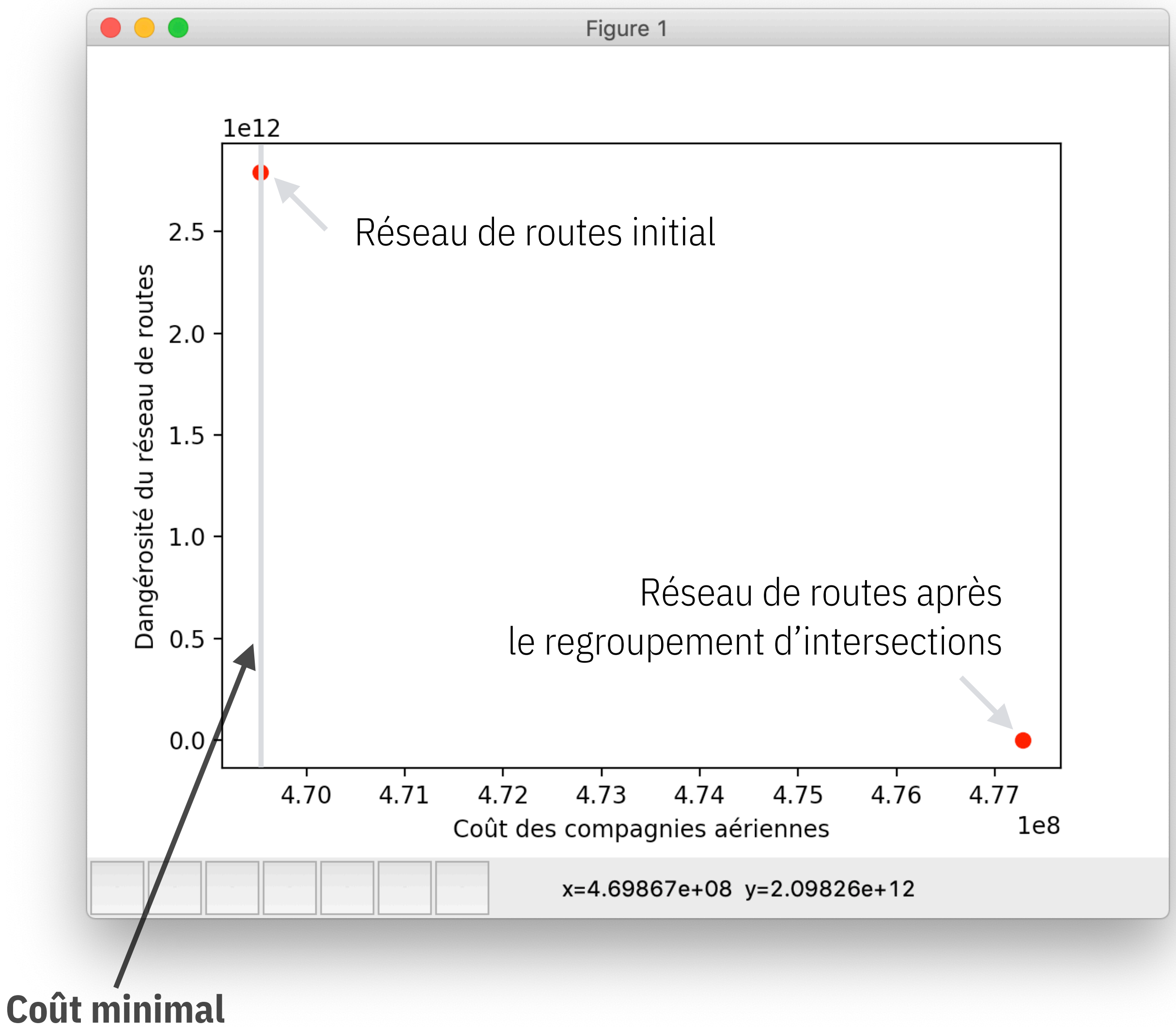
Comparaison des réseaux de routes



Réseau de routes **initial** construit à partir **de trajectoires directes** (I).

↳ Il **minimise** le coût des compagnies aériennes.

Comparaison des réseaux de routes



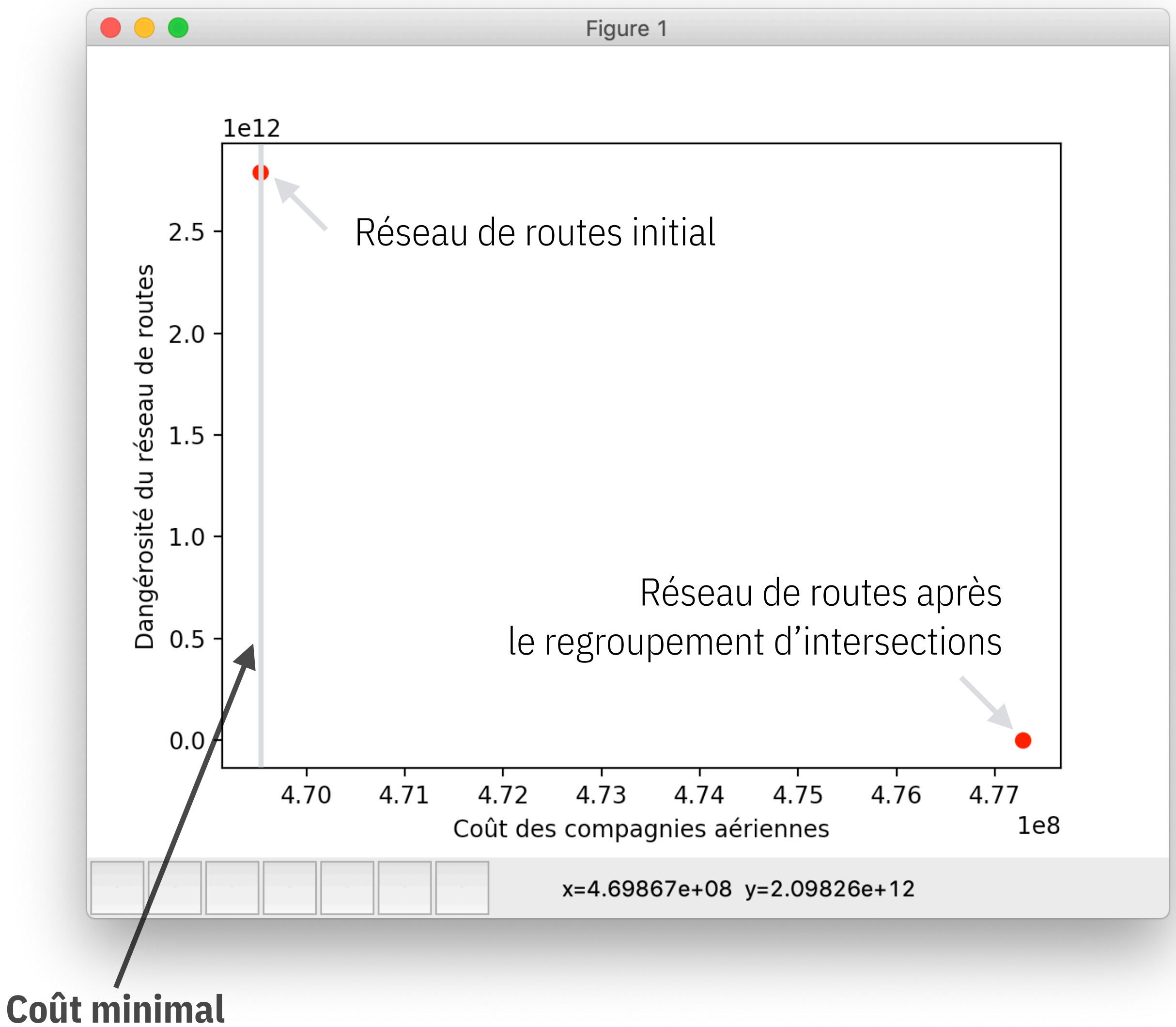
Réseau de routes **initial** construit à partir **de trajectoires directes** (I).

↳ Il **minimise** le coût des compagnies aériennes.

Réseau de routes proposé après le regroupement d'intersections (II):

- **satisfait mieux** le contrôle du trafic aérien (éloignement des intersections).
- **bien moins intéressant** pour les compagnies aériennes.

Comparaison des réseaux de routes



Réseau de routes **initial** construit à partir **de trajectoires directes** (I).

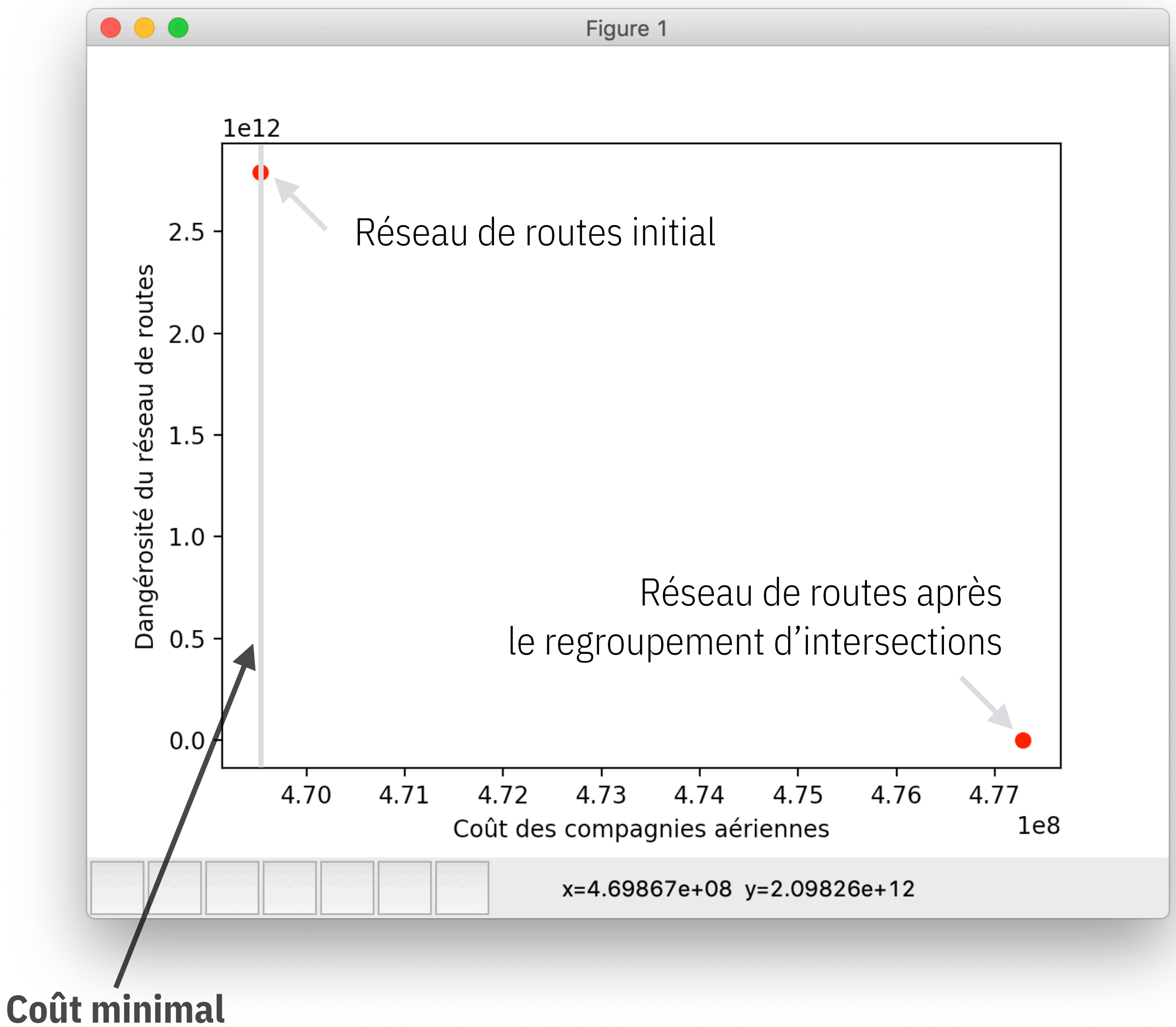
↳ Il **minimise** le coût des compagnies aériennes.

Réseau de routes proposé après le regroupement d'intersections (II):

- **satisfait mieux** le contrôle du trafic aérien (éloignement des intersections).
- **bien moins intéressant** pour les compagnies aériennes.

Remarquons que **minimiser** la dangérosité d'un réseau de routes peut éventuellement **faire augmenter** le coût des compagnies aériennes qui lui est associé.

Comparaison des réseaux de routes



Réseau de routes **initial** construit à partir **de trajectoires directes** (I).

↳ Il **minimise** le coût des compagnies aériennes.

Réseau de routes proposé après le regroupement d'intersections (II):

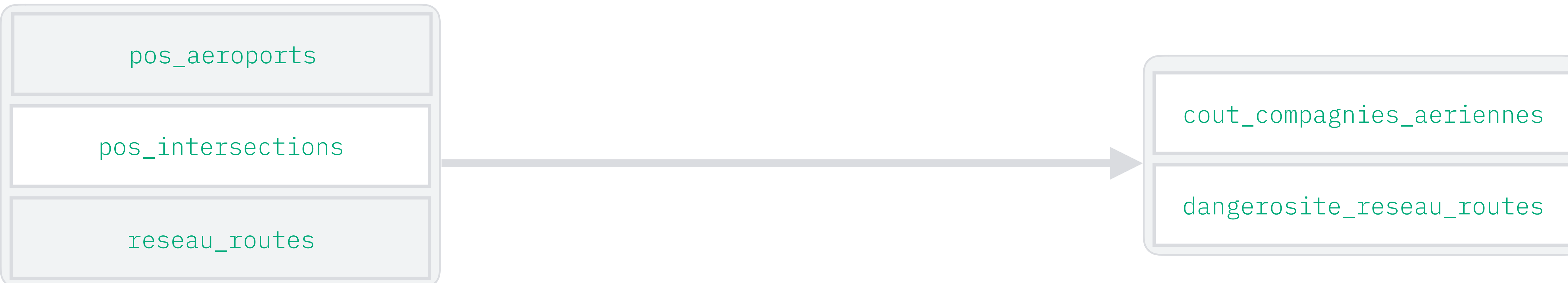
- **satisfait mieux** le contrôle du trafic aérien (éloignement des intersections).
- **bien moins intéressant** pour les compagnies aériennes.

Remarquons que **minimiser** la dangérosité d'un réseau de routes peut éventuellement **faire augmenter** le coût des compagnies aériennes qui lui est associé.

Rappel : par construction, le réseau de routes proposé en (II) est **de bonne qualité**.

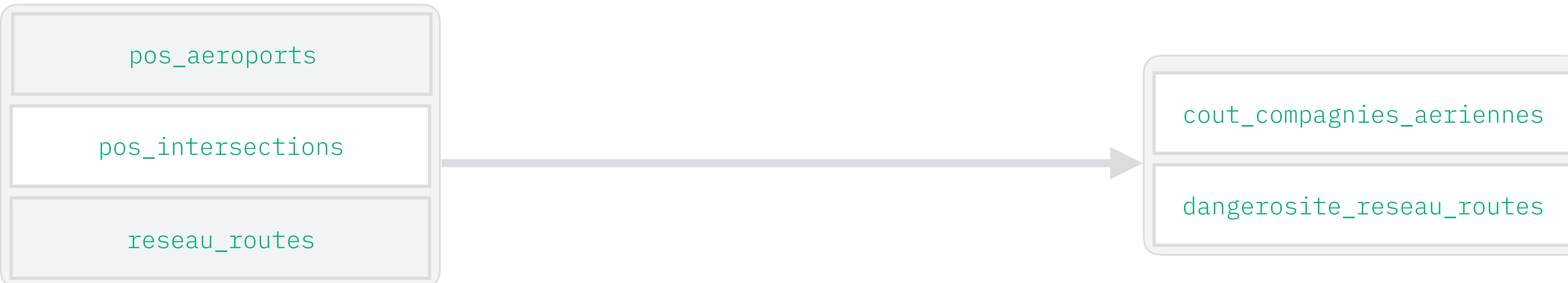
Ajustement des positions d’intersections

Déterminer le **réseau de routes optimal** revient à **minimiser la fonction suivante** :



Ajustement des positions d'intersections

Déterminer le **réseau de routes optimal** revient à **minimiser la fonction suivante** :



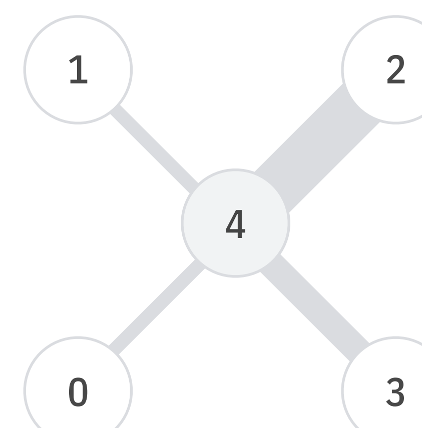
Qui se réécrit : $\begin{array}{rcl}\mathbb{R}^{2n} & \to & \mathbb{R}^2 \\ X & \mapsto & (C(X), D(X))\end{array}$ où $X = (x_0, y_0, ..., x_n, y_n)$: `pos_intersections`

Ajustement des positions d'intersections

Déterminer le **réseau de routes optimal** revient à **minimiser la fonction suivante** :

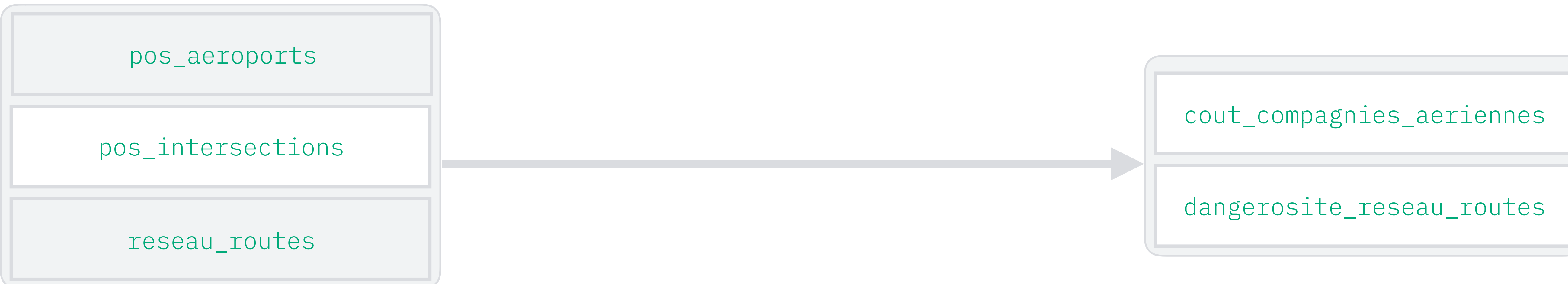
Qui se réécrit : $\begin{aligned} \mathbb{R}^{2n} &\to \mathbb{R}^2 \\ X &\mapsto (C(X), D(X)) \end{aligned}$ où $X = (x_0, y_0, \ldots, x_n, y_n)$: pos_intersections

Pour réaliser ceci, on peut par exemple procéder itérativement :



Ajustement des positions d'intersections

Déterminer le **réseau de routes optimal** revient à **minimiser la fonction suivante** :

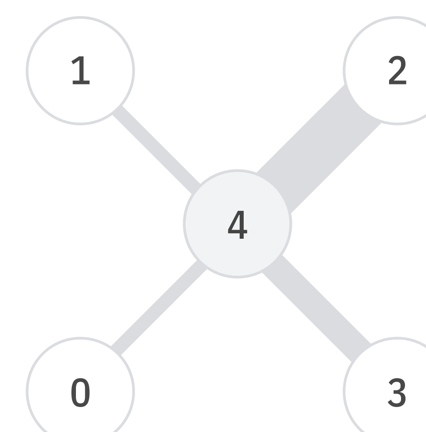


Qui se réécrit : $\begin{array}{rcl} \mathbb{R}^{2n} & \to & \mathbb{R}^2 \\ X & \mapsto & (C(X), D(X)) \end{array}$ où $X = (x_0, y_0, \ldots, x_n, y_n)$: `pos_intersections`

Pour réaliser ceci, on peut par exemple procéder itérativement :

- **Rapprocher** « d'une petite distance » chaque intersection $\boldsymbol{k}$ de son voisin **i** pour lequel $f_{ik} + f_{ki}$ est le plus grand.

↳ Diminuer le coût des compagnies aériennes.

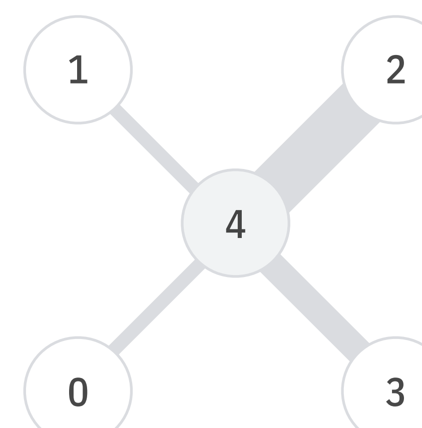


Ajustement des positions d’intersections

Déterminer le **réseau de routes optimal** revient à **minimiser la fonction suivante** :

Qui se réécrit : $\begin{array}{l}\mathbb{R}^{2n} \to \mathbb{R}^{2} \\ \quad X \mapsto (C(X), D(X))\end{array}$ où $X = (x_0, y_0, ..., x_n, y_n)$: `pos_intersections`

Pour réaliser ceci, on peut par exemple procéder itérativement :



- **Rapprocher** « d’une petite distance » chaque intersection $\boldsymbol{k}$ de son voisin **i** pour lequel $f_{ik} + f_{ki}$ est le plus grand.

↳ Diminuer le coût des compagnies aériennes.

- **Repousser** « d’une petite distance » chaque intersection $\boldsymbol{k}$ de la **paire de voisins** pour laquelle $f_{ik} \times f_{jk}$ est le plus grand.

↳ Diminuer la dangérosité du réseau de routes.

Merci de m'avoir écouté !

I.

```python
def det_trajectoires(demande, m):
    """
    Déterminer la liste des trajectoires [i, j] en fonction de la demande.
    :param demande: matrice de la demande de trafic aérien (Numpy.ndarray).
    :param m: nombre d'aéroports (Int).
    :return: trajectoires (List of list of int).
    """

    trajectoires = []   # liste des trajectoires

    for i in range(m):
        for j in range(i):  # matrice triangulaire strictement inférieure

            if demande[i, j] != 0:
                trajectoires.append([i, j])

    return trajectoires
```

I.

```python
def cal_inter(s1, s2, pos):
    """
    Calculer l'intersection entre les deux segments, s'il en existe.
    :param s1: le segment d'extrémités s_1[0] et s_1[1] (List of int).
    :param s2: le segment d'extrémités s_2[0] et s_2[1] (List of int).
    :param pos: liste des positions (List of list of float).
    :return: position de l'intersection (List of float) ou None
    """

    s_1 = [pos[s1[0]], pos[s1[1]]]
    s_2 = [pos[s2[0]], pos[s2[1]]]

    if s_1[1][0] != s_1[0][0] and s_2[1][0] != s_2[0][0]:   # droites non verticales

        c_1 = (s_1[1][1] - s_1[0][1]) / (s_1[1][0] - s_1[0][0])     # coefficient directeur de s_1
        c_2 = (s_2[1][1] - s_2[0][1]) / (s_2[1][0] - s_2[0][0])  # coefficient directeur de s_2

        o_1 = s_1[1][1] - c_1 * s_1[1][0]   # ordonnée à l'origine de s_1
        o_2 = s_2[1][1] - c_2 * s_2[1][0]  # ordonnée à l'origine de s_1

        if c_1 != c_2:  # droites sont sécantes
            x_inter = (o_2 - o_1) / (c_1 - c_2)
            y_inter = c_1 * x_inter + o_1

            if s_1[0][0] <= s_1[1][0]:
                if s_2[0][0] <= s_2[1][0]:

                    if s_1[0][0] < x_inter < s_1[1][0] and s_2[0][0] < x_inter < s_2[1][0]:
                        return [x_inter, y_inter]

                else:

                    if s_1[0][0] < x_inter < s_1[1][0] and s_2[1][0] < x_inter < s_2[0][0]:
                        return [x_inter, y_inter]

            else:   # s_1[0][0] <= s_1[1][0]
                if s_2[0][0] <= s_2[1][0]:

                    if s_1[1][0] < x_inter < s_1[0][0] and s_2[0][0] < x_inter < s_2[1][0]:
                        return [x_inter, y_inter]

                else:

                    if s_1[1][0] < x_inter < s_1[0][0] and s_2[1][0] < x_inter < s_2[0][0]:
                        return [x_inter, y_inter]
        else :
            return None
    else:   # droites verticales
        print("cas à traiter !")
        return None
```

I.

```python
def det_inter_routes(pos_a, traj):
    """
    Déterminer la liste des positions des intersections des trajectoires et celle des rts.
    :param pos_a: liste des positions des aéroports (List of list of float).
    :param traj: listes des trajectoires (List of list of int).
    :return: pos_i (List of list of float), rts (List of list of int).
    """

    routes_potentielles = copy.deepcopy(traj)

    pos = copy.deepcopy(pos_a)   # liste des positions
    rts = []     # liste des rts

    while routes_potentielles:  # liste non vide

        route_p = routes_potentielles.pop()

        if not routes_potentielles: # liste vide
            rts.append(route_p)  # dernier élément de la liste

        else:   # liste non vide

            compt = 0   # compteur (on commence par accéder le 1er élément de la liste)

            while compt < len(routes_potentielles) and not cal_inter(route_p, routes_potentielles[compt], pos):
                compt += 1  # incrémenter tant que pas d'intersection trouvée

            if compt >= len(routes_potentielles):   # aucune intersection avec les autres ( graphe planaire recherché )
                rts.append(route_p)

            else:   # une intersection trouvée entre route_p et et routes_potentielles[compt]

                candidat_inter = routes_potentielles.pop(compt) # retiré et stocké

                pos.append(cal_inter(route_p, candidat_inter, pos))  # ajout de la nouvelle intersection

                extremites = [route_p[0], route_p[1], candidat_inter[0], candidat_inter[1]]

                for i in extremites:    # ajout de nouvelles rts potentielles
                    routes_potentielles.append([i, len(pos) - 1])

                # Remarque : [i,j] = [j,i]
    return pos, rts
```

I.

```python
def rep_reseau(pos, rts):
    """
    Représenter matriciellement le réseau de routes : matrice d'adjacence.
    :param pos: liste des positions (List of list of float).
    :param rts: liste de routes (List of list of int).
    :return: matrice d'adjacence représentant les arêtes (List of list of int).
    """
    reseau = [[0 for j in range(len(pos))] for i in range(len(pos))]

    for k in rts:
        reseau[k[0]][k[1]], reseau[k[1]][k[0]] = 1, 1

    return reseau


def tra_routes(pos, rts):
    """
    Tracer les routes avec matplotlib.
    :param pos: liste des positions des aéroports (List of list of float).
    :param rts: liste des routes (List of list of int).
    :return: None.
    """
    nbr = len(rts)

    for i in range(nbr):

        plt.plot([pos[rts[i][0]][0], pos[rts[i][1]][0]], [pos[rts[i][0]][1], pos[rts[i][1]][1]], color='black')

    plt.scatter((100000, 1200000), (6000000, 7100000), color='white')
    plt.show()
```

II.

```python
def det_diag_voronoi(pos_i):
    """
    Déterminer le diagramme de Voronoï correspondant à l'ensemble des positions des intersections.
    :param pos_i: liste des positions des intersections (List of list of floats) ou (Ndarray of float).
    :return: diagramme de Voronoï (Voronoi).
    """
    p_i = np.asarray(pos_i)

    vor = Voronoi(p_i)  # création du diagramme de Voronoï

    return vor


def rep_diag_voronoi(vor):
    """
    Représenter le diagramme de Voronoï.
    :param vor: diagramme de Voronoï (Voronoi).
    :return: None
    """
    fig = voronoi_plot_2d(vor, show_vertices=False, line_colors='black', line_width=1, line_alpha=0.6, point_size=2)
    plt.show()
    return None


def det_som_sect_eti(vor):
    """
    Déterminer les positions des sommets du diagramme de Voronoï, la liste des secteurs,
    et une liste reliant chaque intersection à son secteur.
    :param vor: diagramme de Voronoi (Voronoi).
    :return: positions des sommets (Ndarray of float),
             liste des secteurs (List of list of ints) et
             pour chaque intersection, indice du secteur correspondant à l'intersection (List of ints).
    """

    return vor.vertices, vor.regions, vor.point_region
```

II.

```python
def det_voisins(k, reseau):
    """
    Déterminer les voisins du sommet k dans le réseau de routes.
    :param k: indice correspondant au sommet (Int).
    :param reseau: matrice d'adjacence, représentant les arêtes (Ndarray of int).
    :return: liste des indices des voisins de k (List of ints).
    """

    voisins = []

    for i in range(len(reseau)):
        if reseau[k, i] == 1:
            voisins.append(i)

    return voisins


def det_flux_secteurs(nbr_a, nbr_i, reseau, f_routes):
    """
    Déterminer le flux d'avions des secteurs.
    :param nbr_a: nombre d'aéroports (Int).
    :param nbr_i: nombre d'intersections (Int).
    :param reseau: matrice d'adjacence (Nd).
    :param f_routes: matrices des flux associés aux routes (Ndarray of int).
    :return: matrice des flux associés aux secteurs (Ndarray of int).
    """

    f_secteurs = np.zeros(nbr_i, dtype=int)   # matrice des flux des secteurs

    for i in range(nbr_i):

        j = i + nbr_a   # indice de l'intersection dans le réseau de routes

        voisins_j = det_voisins(j, reseau)

        for v in voisins_j:
            f_secteurs[i] += f_routes[j, v] + f_routes[v, j]

    return f_secteurs
```

II.

```python
def cal_angle_distance(p_s):    # un seul paramètre en entrée (sorted...)
    """
    Calculer l'angle que fait p_s avec v_ref (sens horaire) et la distance par rapport à l'origine.
    :param p_s: position du sommet (List of floats).
    :return: angle et la distance (float, float).
    """

    v = [p_s[0] - p_origine[0], p_s[1] - p_origine[1]]  # vecteur associé à la position du sommet
    n_v = math.hypot(v[0], v[1])    # norme de ce vecteur

    if n_v == 0:    # angle non défini ?
        return -math.pi, 0

    v_n = [v[0] / n_v, v[1] / n_v]  # vecteur normalisé

    ps = v_n[0] * v_ref[0] + v_n[1] * v_ref[1]     # x1*x2 + y1*y2 (produit scalaire)
    pv = v_n[0] * v_ref[1] - v_n[1] * v_ref[0]   # x1*y2 - x2*y1 (z du produit vectoriel R^3)

    angle = math.atan2(pv, ps)

    if angle < 0:   # angles négatifs dans le sens anti-horaire -> soustraire de 2*pi
        return (2 * math.pi + angle), n_v

    return angle, n_v   # angle (1er critère) -> si égalité : distance (2ème critère)


def tri_somm_secteur(pos_s, p_i):
    """
    Trier les sommets du secteur dans le sens horaire.
    :param pos_s: liste des positions des sommets du secteur (List of list of floats).
    :param p_i: position de l'intersection correspondant au secteur, l'origine (List of floats).
    :return: liste des positions des sommets du secteur triées (List of list of floats).
    """

    global p_origine
    p_origine = p_i  # modification de l'origine : position de l'intersection -> origine

    return sorted(pos_s, key=cal_angle_distance)
```

II.

```python
def cal_aire_secteur(pos_s, p_i):
    """
    Calculer l'aire du secteur pos_s dont l'intersection correspondante est p_i.
    :param pos_s: liste des positions des sommets du secteur (List of list of floats).
    :param p_i: position de l'intersection correspondant au secteur (List of floats).
    :return: l'aire correspondant au secteur (Float).
    """

    pos_s = tri_somm_secteur(pos_s, p_i)     # les positions des sommets triées pour calculer l'aire

    aire, nbr_s = 0, len(pos_s)    # initialiser l'aire et nombre de sommets formant le secteur

    for i in range(nbr_s-1):
        aire += pos_s[i][0] * pos_s[i+1][1]     # + a_1*b_2 + .... + a_{n-1}*b_n
        aire -= pos_s[i][1] * pos_s[i+1][0]     # - b_1*a_2 - .... - b_{n-1}*a_n

    aire += pos_s[nbr_s-1][0] * pos_s[0][1]   # + a_n*b_1
    aire -= pos_s[nbr_s-1][1] * pos_s[0][0]   # - b_n*a_1

    aire = math.fabs(aire)  # valeur absolue
    aire /= 2.

    return aire


def det_aire_secteurs(pos_i, nbr_i, pos_s, sect, eti):
    """
    Déterminer les aires associées aux secteurs. (-1 si l'aire est infini...)
    :param pos_i: positions des intersections (Numpy of float).
    :param nbr_i: nombre d'intersections (Int).
    :param pos_s: positions des sommets du diagramme (Numpy of float).
    :param sect: Liste des secteurs (List of list of ints).
    :param eti: liste des indices associant les intersections aux secteurs (List of ints).
    :return: aires des secteurs (Numpy of float)
    """
    a_secteurs = np.zeros(nbr_i)   # aires des secteurs

    for i in range(nbr_i):
        s_i = sect[eti[i]]  # secteur associé à l'intersection i

        if s_i != [] and (-1) not in s_i:
            a_secteurs[i] = cal_aire_secteur([[pos_s[s][0], pos_s[s][1]] for s in s_i], pos_i[i])
        else:
            a_secteurs[i] = -1.     # aire -> infini

    return a_secteurs
```

II.

```python
def eva_densites_secteurs(nbr_i, f_secteurs, a_secteurs):
    """
    Evaluer la densité des secteurs (-1 si non définie).
    :param nbr_i: nombre d'intersections (Int).
    :param f_secteurs: flux des secteurs (Ndarray of int).
    :param a_secteurs: aires des secteurs (Ndarray of float).
    :return: densités des secteurs (Ndarray of float).
    """
    d_secteurs = np.zeros(nbr_i)    # densités des secteurs

    for i in range(nbr_i):
        if a_secteurs[i] == -1.:
            d_secteurs[i] = 0.  # aire -> 0
        else:
            d_secteurs[i] = f_secteurs[i] / a_secteurs[i]

    return d_secteurs


def con_voronoi(nbr_i, pos_s, sect, eti):   # elle n'est plus utilisée
    """
    Convertir le diagramme de Voronoï pour le représenter.
    :param nbr_i: nombre d'intersections (Int).
    :param pos_s: positions des sommets du diagramme (Ndarray of float).
    :param sect: liste des secteurs (List of list of ints).
    :param eti: liste associant à chaque intersection son secteur (List of ints).
    :return: représentation du diagramme de Voronoï (List of list of list of floats).
    """
    r_vor = []

    for i in range(nbr_i):
        s_i = sect[eti[i]]  # secteur correspondant à l'intersection i
        if -1 not in s_i:
            r_vor.append([[pos_s[s][0], pos_s[s][1]] for s in s_i])
        else:
            r_vor.append([])

    return r_vor
```

II.

```python
def par_intersections(pos_i, k):
    """
    Partitionner les intersections en k groupes.
    :param pos_i: liste des positions des intersections (List of list of floats).
    :param k: le nombre de partitions (Int).
    :return: positions des barycentres (List of list of floats),
             matrice associant à chaque intersection son barycentre (Ndarray of int).
    """

    p_i = np.asarray(pos_i)   # conversion en matrice

    kmeans = KMeans(n_clusters=k, init='k-means++').fit(p_i)

    return kmeans.cluster_centers_, kmeans.labels_
```

II.

```python
def det_reseau_routes_reduit(pos_a, pos_b, eti, a_res):
    """
    Déterminer le réseau de routes réduit correspondant aux barycentres (nouvelles intersections).
    :param pos_a: liste des positions des aéroports (List of list of floats).
    :param pos_b: liste des positions des barycentres (Ndarray of float).
    :param eti: matrice associant à chaque intersection son barycentre (Ndarray of int).
    :param a_res: matrice d'adjacence, réseau de routes (Ndarray of int).
    :return: le nouveau réseau de routes construit (Ndarray of int).
    """

    a_n_pos = len(a_res)    # nombre de sommets dans l'ancien réseau
    n_a = len(pos_a)    # nombre d'aéroports

    pos = np.concatenate((np.array(pos_a), pos_b))     # liste des positions
    n_pos = len(pos)   # longueur de la liste des positions

    reseau = np.zeros((n_pos, n_pos), dtype=int)    # initialisation du nouveau réseau de routes

    for i in range(a_n_pos):    # on parcourt l'ancien réseau de routes
        for j in range(a_n_pos):

            if a_res[i, j] == 1:     # (i, j) est une route dans l'ancien réseau de routes ?

                if i >= n_a:    # extrémité gauche de la route : intersection
                    if j >= n_a:    # extrémité droite : intersection
                        reseau[eti[i - n_a] + n_a, eti[j - n_a] + n_a] = 1

                    else:   # extrémité droite : aéroport
                        reseau[eti[i - n_a] + n_a, j] = 1

                else:   # extrémité gauche de la route : aéroport
                    if j >= n_a:  # extrémité droite : intersection
                        reseau[i, eti[j - n_a] + n_a] = 1

                    else:  # extrémité droite : aéroport
                        reseau[i, j] = 1

    return reseau
```

II.

```python
def det_nbr_inter_optimal(pos_a, pos_i, reseau, seuil):
    """
    Déterminer le nombre d'intersections optimal.
    :param pos_a: liste des positions des aéroports (List of list of floats)
    :param pos_i: liste des positions des intersections (List of list of floats).
    :param reseau: matrice d'adjacence, représentant les arêtes (Ndarray of float).
    :param seuil : seuil sur le flux d'avions d'un secteur (Int).
    :return: k (le nombre d'intersections optimal)
    """

    pos_b = np.copy(np.array(pos_i))  # initialisation des positions des barycentres
    res = np.copy(reseau)   # réseau de routes aériennes

    # INITIALISATION
    compt = 100     # compteur initialisé à 100 car solution forcément plus petite.

    pos_b, eti_i_b = par_intersections(pos_b, compt)  # détermination des nouvelles intersections = barycentres
    res = det_reseau_routes_reduit(pos_a, pos_b, eti_i_b, res)    # nouveau reseau de routes
    nbr_b = len(pos_b)

    pos = np.concatenate((np.array(pos_a), pos_b))  # positions
    d_routes = c.det_distances_routes(pos, res)     # distances des routes
    f_routes = c.det_flux_routes(c.demande_trafic, c.trajectoires_reseau, d_routes)     # flux des routes
    f_secteurs = det_flux_secteurs(nbr_aeroports, nbr_b, res, f_routes)  # flux des secteurs

    while (np.all(f_secteurs < seuil)) and (compt > 5):     # flux des secteurs inférieurs au seuil

        print(compt)

        pos_b, eti_i_b = par_intersections(pos_b, compt)  # détermination des nouvelles intersections = barycentres
        res = det_reseau_routes_reduit(pos_a, pos_b, eti_i_b, res)  # nouveau reseau de routes
        nbr_b = len(pos_b)

        pos = np.concatenate((np.array(pos_a), pos_b))  # positions
        d_routes = c.det_distances_routes(pos, res)  # distances des routes
        f_routes = c.det_flux_routes(c.demande_trafic, c.trajectoires_reseau, d_routes)  # flux des routes
        f_secteurs = det_flux_secteurs(nbr_aeroports, nbr_b, res, f_routes)  # flux des secteurs

        compt -= 1

    vor = det_diag_voronoi(pos_b)
    pos_s, sect, eti_i_s = det_som_sect_eti(vor)

    rep_diag_voronoi(vor)

    return compt + 1
```

III.

```python
def cal_distance(p1, p2):
    """
    Calculer la distance euclidienne entre deux points.
    :param p1: 1er point (List of float).
    :param p2: 2ème point (List of float).
    :return: la distance euclidienne (Float).
    """
    return sqrt((p2[0]-p1[0])**2 + (p2[1]-p1[1])**2)


def det_distances_routes(pos, reseau):
    """
    Déterminer la distance entre deux sommets (aéroports ou intersections).
    :param pos: liste des positions (List of list of float).
    :param reseau: matrice représentant les routes (List of list of int).
    :return: matrice associant à chaque arête une distance (Numpy.ndarray).
    """

    nbr = len(pos)  # nombre de positions

    dist = np.zeros((nbr, nbr), dtype=float)    # initialisation

    for i in range(nbr):
        for j in range(nbr):

            if reseau[i][j] == 1:    # (i,j) est une route
                dist[i, j] = cal_distance(pos[i], pos[j])

            else:   # (i, j) n'est pas une route (en particulier, i = j)
                dist[i, j] = 0

    return dist
```

III.

```python
def det_flux_routes(dem, traj, dist):
    """
    Déterminer le flux d'avions aux routes.
    :param dem: matrice de la demande de trafic aérien (Numpy.ndarray).
    :param dist: matrice des distances associées aux routes (Numpy.ndarray).
    :param traj: liste des trajectoires (List of list of int).
    :return: matrice des flux associés aux routes (Numpy.ndarray).
    """

    nbr_pos = dist.shape[0]  # nombre de positions (aéroports et intersections)

    flux = np.zeros((nbr_pos, nbr_pos), dtype=int)  # initialisation

    dist_csr = csr_matrix(dist)    # matrice des distances compressée
    _, pre = floyd_warshall(dist_csr, directed=True, return_predecessors=True)    # matrice des prédecesseurs

    for t in traj:  # Attention : [i, j] = [j, i] comptée 1 seule fois mais dans non dans la demande

        i, j = t[0], t[1]   # extrémités du chemin
        f_ij, f_ji = dem[i, j], dem[j, i]     # flux d'avions associé à t (le chemin est valable dans les 2 sens)

        p = pre[i, j]   # initialisation du prédecessur

        while p != -9999:   # existence du prédecesseur, [k, j] est une route intervenant dans la trajectoire

            flux[p, j] += f_ij   # ajout au nombre d'avions empruntant cette route dans le sens (i,j)
            flux[j, p] += f_ji  # ajout du flux dans le sens contraire

            j = p   # nouvelle extrémité à droite
            p = pre[i, j]   # nouveau prédecesseur

    return flux
```

III.

```python
def eva_cout_compagnies(dist, flux):
    """
    Evaluer le coût des compagnies pour un réseau de routes donné.
    :param dist: matrice des distances associées aux routes (Numpy.ndarray).
    :param flux: matrice des flux associés aux routes (Numpy.ndarray).
    :return: le coût associé à ce réseau de routes (Float).
    """

    nbr_pos = dist.shape[0]

    c = 0   # coût des compagnies aériennes

    for i in range(nbr_pos):
        for j in range(nbr_pos):

            if i != j and dist[i, j] != 0.:
                c += dist[i, j] * flux[i, j]

    return c
```

III.

```python
def cal_angl_routes(pos, k, i, j):
    """
    Calculer l'angle intérieur entre les routes [i, k] et [j , k].
    :param k: l'intersection du mileu (au niveau de l'angle) (Int).
    :param i: 1er voisin de k (Int).
    :param j: 2ème voisin de k (Int).
    :param pos: positions des intersections (Ndarray of float).
    :return: l'angle intérieur en radians (Float).
    """

    # Détermination des vecteurs associés
    u = [pos[k][0] - pos[i][0], pos[k][1] - pos[i][1]]  # vecteur associé à [i, k]
    v = [pos[k][0] - pos[j][0], pos[k][1] - pos[j][1]]  # vecteur associé à [j, k]

    # Normalisation des vecteurs
    n_u, n_v = math.hypot(u[0], u[1]), math.hypot(v[0], v[1])   # normes de u et v

    if n_u == 0. or n_v == 0.:  # u ou v nuls
        raise ValueError("u ou v est nul !")

    u_n, v_n = [u[0] / n_u, u[1] / n_u], [v[0] / n_v, v[1] / n_v]  # vecteurs u et v normalisés

    # Produit scalaire des vecteurs normalisés
    ps_uv_n = u_n[0] * v_n[0] + u_n[1] * v_n[1]  # produit scalaire canonique de u_n et v_n

    if ps_uv_n > 1:     # correction des erreurs de calcul de Python
        ps_uv_n = 1
    if ps_uv_n < -1:
        ps_uv_n = -1

    # Détermination de l'angle
    return math.acos(ps_uv_n)
```

III.

```python
def eva_dang_reseau(pos, nbr_a, res, f_routes):
    """
    Evaluer la dangerosité du réseau de routes.
    :param pos: positions (Ndarray of float).
    :param nbr_a: nombre d'aéroports (Int).
    :param res: matrice d'adjacence, représenter les routes (Ndarray of Int).
    :param f_routes: matrice des flux routes (Ndarray of Int).
    :return: la valeur associée au réseau de routes (Float).
    """
    dang = 0.   # dangérosité du réseau de routes

    for k in range(nbr_a, len(pos)):

        voisins = r.det_voisins(k, res)   # voisins de k

        for i in voisins:
            for j in voisins:
                if i != j:

                    angle_ij = cal_angl_routes(pos, k, i, j)

                    if 0 <= angle_ij < math.pi:
                        dang += (f_routes[i, k] * f_routes[j, k]) / math.cos(angle_ij / 2)

                    else:   # angle_ij = pi
                        dang += f_routes[i, k] * f_routes[j, k]

    return dang
```