LSDisplacer 0.1.0

Imran Lokhat

1 Introduction

Le problème considéré est de déplacer/déformer un ensemble de lignes (des talus), par rapport à un réseau d'autres lignes (des routes) pour qu'à une échelle donnée, leurs rendus ne se superposent pas.
Un talus est une polyligne, i.e. une suite de points 2D qui forment des segments consécutifs (Figure 1).

FIGURE 1 – Points et segments d'un talus

L'échelle est donnée, on sait donc de combien de mètres on cherche à déplacer les talus pour qu'ils soient suffisamment éloignés de l'ensemble des routes en présence.
Le processus est local, on est au niveau d'une petite zone correspondant à quelques routes et talus qui forment une unité logique, i.e. un îlot urbain ou un morceau d'îlot (Figure 2).
À la fin, on obtient de nouvelles coordonnées pour les points dont sont

constitués les talus.

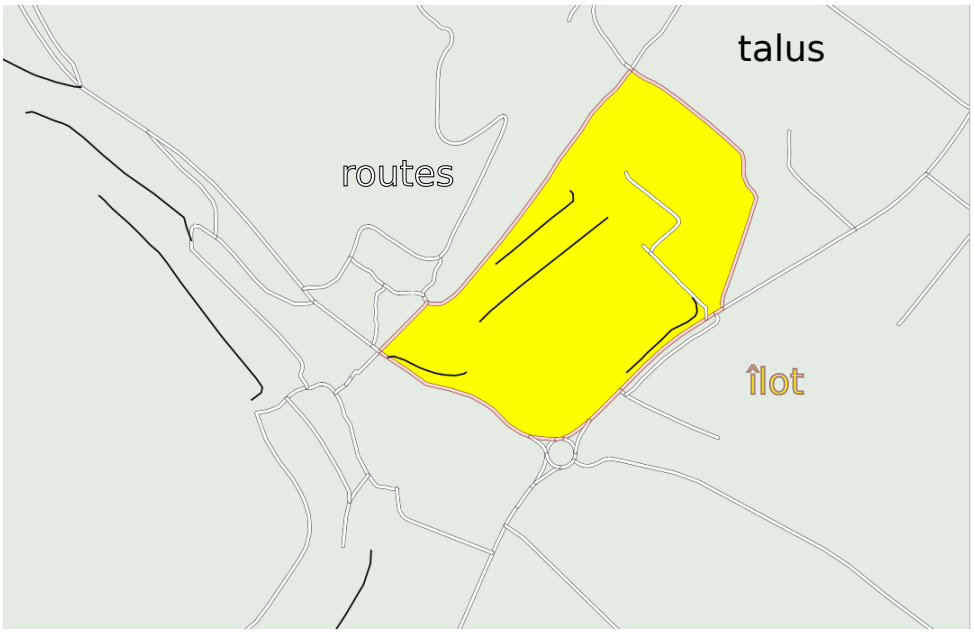


FIGURE 2 – Îlot, routes et talus

Afin d'essayer de minimiser les déformations spatiales des talus, un certain nombre de contraintes sont exprimées sous forme d'équations portant sur les coordonnées des points, et le système est résolu par moindres carrés.

2 Moindres carrés non linéaires

Le procédé est itératif, à chaque étape m un vecteur d'incrément dX^m est calculé, qu'on ajoute aux coordonnées courantes X^m pour obtenir la nouvelle solution $X^{m+1} = dX^m + X^m$.
On réitère jusqu'à ce que le critère d'arrêt soit satisfait :

- ou bien on a atteint le nombre maximal d'itérations fixé
- ou alors $\|dX^m\|_\infty$ est inférieure à un seuil donné, i.e. le déplacement est minime suivant toutes les directions

Le formalisme utilisé est le suivant :

X^m vecteur des coordonnées $(x_0, y_0, \ldots, x_n, y_n)$ des points de l'ensemble des talus à l'itération m (les coordonnées initiales sont donc X^0)

dX^m vecteur d'incrément obtenu à la fin de l'itération m

P matrice des poids des contraintes

Y vecteur des observations, il contient les valeurs initiales pour les coordonnées des points, les produits vectoriels des angles, les longueurs de segments et autant de 0 que de routes

$S(X^m)$ fonction du modèle, calcule les contraintes pour la solution courante à l'itération m

A matrice jacobienne du système, i.e. les valeurs des dérivées partielles pour chaque contrainte à l'itération m

Avec $B = Y - S(X^m)$, les équations normales du système à l'itération m sont :

$$(A^T PA)dX^m = A^T PB$$

et

$$X^{m+1} = X^m + (A^T PA)^{-1} A^T PB$$

3 Contraintes

Les contraintes, dont la formulation mathématique est $S(X)$, s'appliquent sur les coordonnées des points des talus. Elles cherchent à éloigner les talus des routes de la distance voulue, à minimiser leur déformation (angles et segments internes), et à préserver la figure que forme l'ensemble des talus.

3.1 Mouvement

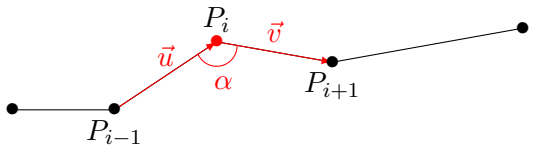
Les points ne doivent pas trop s'éloigner de leur position d'origine. La contrainte s'exprime trivialement comme :

$$c_i = c_i \quad \forall c_i \in \{x_0, y_0, \ldots, x_n, y_n\} \tag{1}$$

Il y a autant de contraintes que de coordonnées, soit deux fois le nombre de points.

3.2 Angles

Pour un talus donné, on essaie de conserver les angles que forment les segments entre eux.



En chaque point P_i qui n'est pas une des extrémités du talus, sommet d'un angle α, on considère les deux points P_{i-1} et P_{i+1} qui l'entourent, et on calcule le produit vectoriel des deux vecteurs (normés) qui se suivent.

$$C(\alpha) = C(P_{i-1}, P_i, P_{i+1}) = \frac{\vec{u}}{\|\vec{u}\|} \times \frac{\vec{v}}{\|\vec{v}\|} \tag{2}$$

Il y a autant de contraintes qu'il y a d'angles, i.e. le nombre total de points de talus moins deux fois le nombre de talus (on enlève les extrémités de chaque talus).

3.3 Longueur des segments

À l'instar des contraintes d'angles, on essaie de conserver les longueurs des segments des talus pour ne pas trop les déformer. On cherche aussi à ne pas trop changer la figure globale formée par l'ensemble des talus.
Pour se faire, on va générer une triangulation de Delaunay contrainte (voir figure 3) entre tous les points de tous les talus, et calculer la longueur de ces segments, i.e. pour deux points de coordonnées (x_1, y_1) et (x_2, y_2) définissant un segment e :

$$\mathrm{L(e)} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3}$$

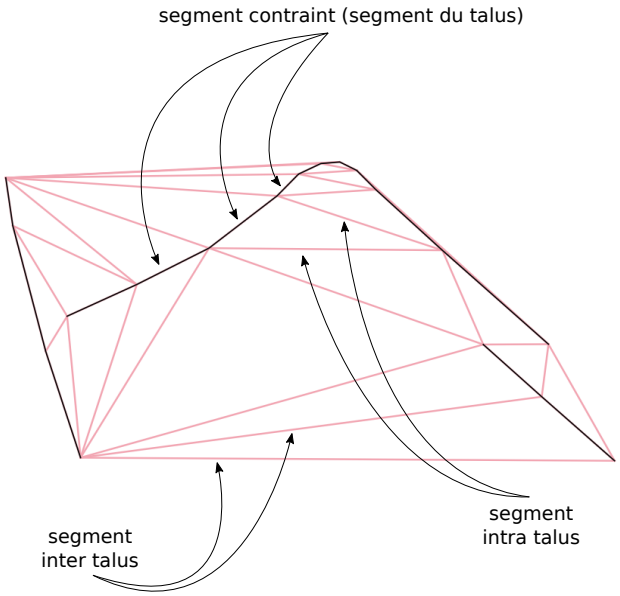


FIGURE 3 – Triangulation contrainte entre tous les points de talus

Il y a autant de contraintes que de segments issus de la triangulation (Θ(nombre de points), empiriquement à peu près deux fois le nombre de points).

3.4 Distance minimale aux routes

On impose aux points des talus d'être à une distance minimale fixée, disons $buffer_k$, d'une route r_k donnée.
Appelons l'ensembles des talus $\mathbf{T}$, et $dmin_k$ la distance de l'ensemble des polylignes $\mathbf{T}$ à la polyligne r_k.
On définit alors une fonction $\mathrm{d}_T(r_k)$:

$$\mathrm{d}_T(r_k) = \begin{cases} \mathrm{buffer}_k - \mathrm{dmin}_k & \text{si } \mathrm{dmin}_k < \mathrm{buffer}_k \\ 0 & \text{sinon} \end{cases} \tag{4}$$

Le nombre de ces contraintes est égal au nombre de routes.

3.5 Mise en forme matricielle

3.5.1 Modèle

Avec les fonctions qui viennent d'être définies, on note $\_^m$ une donnée à l'itération m, et $\_^o$ une donnée initiale.
Les α_i sont les sommets de chaque angle d'un talus, les e_i les segments issus de la triangulation, et les r_i les routes.
Nous écrivons alors la matrice B comme :

$$B = Y - S(X) = \begin{pmatrix} x_0^o - x_0^m \\ y_0^o - y_0^m \\ \vdots \\ x_n^o - x_n^m \\ y_n^o - y_n^m \\ C(\alpha_0^o) - C(\alpha_0^m) \\ \vdots \\ C(\alpha_j^o) - C(\alpha_j^m) \\ L(e_0^o) - L(e_0^m) \\ \vdots \\ L(e_k^o) - L(e_k^m) \\ 0 - d_T(r_0)^m \\ \vdots \\ 0 - d_T(r_l)^m \end{pmatrix} \begin{matrix} \left.\vphantom{\begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix}}\right\} \text{contrainte de mouvement (1)} \\ \left.\vphantom{\begin{matrix} a \\ b \\ c \end{matrix}}\right\} \text{angles (2)} \\ \left.\vphantom{\begin{matrix} a \\ b \\ c \end{matrix}}\right\} \text{longueurs (3)} \\ \left.\vphantom{\begin{matrix} a \\ b \\ c \end{matrix}}\right\} \text{distances aux routes (4)} \end{matrix}$$

3.5.2 Matrice jacobienne

La matrice jacobienne du système est obtenue en calculant les valeurs des dérivées partielles des contraintes, équations (1) à (4), pour les variables concernées .
On notera que comme la fonction de distance à une route (équation (4)) n'a pas d'expression analytique simple, on a recours à une dérivation numérique, i.e. en chaque coordonnée c_i on calcule :

$$\frac{\partial dr_k}{\partial c_i} = \frac{d_r(X + h_i) - d_r(X - h_i)}{2h_i}$$

La matrice A s'écrit :

$$
A = \left(\begin{array}{cccccccccc}
1 & 0 & \dots & & & & & & \dots & 0 \\
0 & 1 & 0 & \dots & & & & & & \vdots \\
\vdots & & & & & & & & & \vdots \\
\vdots & & & & & & & & 1 & 0 \\
0 & \dots & & & & & & \dots & 0 & 1 \\
0 & \dots & 0 & \frac{\partial C}{\partial x_{i-1}} & \frac{\partial C}{\partial y_{i-1}} & \dots & \frac{\partial C}{\partial y_{i+1}} & 0 & \dots & 0 \\
\vdots & & & & & & & & & \vdots \\
0 & \dots & \frac{\partial L}{\partial x_{j-1}} & \frac{\partial L}{\partial y_{j-1}} & \dots & \frac{\partial L}{\partial y_{j+1}} & 0 & \dots & & 0 \\
\vdots & & & & & & & & & \vdots \\
\frac{\partial dr_0}{\partial x_0} & \dots & & & & & & & \dots & \frac{\partial dr_0}{\partial y_n} \\
\vdots & & & & & & & & & \vdots \\
\frac{\partial dr_l}{\partial x_0} & \dots & & & & & & & \dots & \frac{\partial dr_l}{\partial y_n}
\end{array}\right)
\begin{array}{l}
\left.\begin{array}{l}\\ \\ \\ \\ \\ \end{array}\right\} I_{2(n+1)} \\
\left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{angles} \\
\left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{longueurs} \\
\left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{distances routes}
\end{array}
$$

4 Implémentation

Elle [1] est faite en Python, avec les librairies suivantes :
— NumPy pour le calcul matriciel et la résolution par moindres carrés
— Triangle pour la triangulation de Delaunay contrainte
— Shapely et PyGEOS pour les calculs géométriques
— Fiona pour l'import des données

4.1 Classe LSDisplacer

Définie dans le fichier `displacer.py`, c'est le cœur du programme.
La signature de son constructeur est :

```
def __init__(self, points_talus, roads_wkts, talus_lengths, edges,
             edges_dist_min=10, edges_dist_max=30)
```

Paramètres :
— **points_talus** : tableau numpy de l'ensemble des coordonnées des points de tous les talus en présence $[x_0, y_0, \dots, x_n, y_n]$
— **roads_wkts_and_buffers** : liste de couples de routes, au format WKT, et de la distance minimale à laquelle un talus doit être de cette route [('LINESTRING..', 15), ('LINESTRING..', 10), ('LINESTRING..', 15)]
— **talus_lengths** : liste du nombre de points de chaque talus, dans l'ordre des coordonnées passées dans *points_talus*
— **edges** : liste des segments issus de la triangulation, sous forme de couples $(\text{idx}(p_1), \text{idx}(p_2))$, avec $\text{idx}(p_i)$ se référant à l'index de l'extrémité p_i du segment parmi les points

1. `https://github.com/ilokhat/embankment_lsq`

— **edges_dist_min** : longueur minimale d'un segment liant deux talus
— **edges_dist_max** : distance au dessus de laquelle le poids pour la contrainte de conservation de longueur d'un segment liant deux talus devient négligeable

On peut ajuster les paramètres du calcul à l'aide de la méthode de classe :

```
def set_params(MAX_ITER=250, NORM_DX=0.3, H=2.0, DIST='MIN', KKT=False,
       ID_CONST=True, ANGLES_CONST=True, EDGES_CONST=True, DIST_CONST=True,
       PEdges_int=10, PEdges_ext=2, Pedges_ext_far=0, PEdges_int_non_seg=5,
       PFix=1., PAngles=50, PDistRoads=1000)
```

Tous les paramètres ont des valeurs par défaut, il suffit donc de changer seulement ceux que l'on veut :

— **MAX_ITER** : nombre max d'itérations
— **NORM_DX** : seuil pour le critère d'arrêt sur la norme de dX
— **H** : valeur utilisée pour le calcul de la dérivée numérique de la fonction de distance aux routes (équation (4))
— **ID_CONST, ANGLES_CONST, EDGES_CONST, DIST_CONST** : permettent d'activer/désactiver les contraintes de mouvement, angles, longueurs de segments et distances aux routes
— **PFix, PAngles, PDistRoads** : poids sur les contraintes de mouvement, d'angles et de distance aux routes
— **PEdges_int, PEdges_int_non_seg** : poids sur la contrainte de longueur des segments intra-talus issues de la triangulation (segments liant des points consécutifs et non consécutifs)
— **PEdges_ext, Pedges_ext_far** : poids sur la longueur des arêtes inter-talus (longueur inférieure ou supérieure à *edges_dist_max*)
— DIST : **obsolète**. Paramètre qui a servi à tester le type de distance calculée pour la contrainte de distance au routes. Si on met autre chose que 'MIN', au lieu de prendre la distance minimale de l'ensemble des talus à une route, on en prend la moyenne. **Laisser à 'MIN'**
— KKT : **obsolète**. Paramètre qui a servi à tester un moindre carrés contraint [2] (la contrainte de distance aux routes doit être exactement respectée). N'apporte rien. **Laisser à False**

La méthode *square* effectue l'ajustement par moindres carrés. On peut ensuite récupérer les nouveaux talus en appelant la méthode *get_linestrings_wkts*, qui renvoie une liste (un tableau numpy en réalité) des WKT des talus.

Ainsi, lancer de manière minimale le programme consiste, une fois les données en entrée récupérées, à changer les paramètres souhaités, instancier *LSDisplacer*, puis faire appel à sa méthode *square* :

2. Voir par exemple la slide 16 de `https://stanford.edu/class/engr108/lectures/constrained-least-squares_slides.pdf`

```
# ...
# on a récupéré les différentes entrées, point_talus, routes, etc..
# ...

LSDisplacer.set_params(MAX_ITER=300, NORM_DX=0.3,
                       PFix=8.0, PDistRoads=200, PAngles=8,
                       PEdges_ext=15, Pedges_ext_far=0.5,
                       PEdges_int=1, PEdges_int_non_seg=1)

displacer = LSDisplacer(points_talus, roads, talus_lengths, edges)
displacer.square()
for talus in displacer.get_linestrings_wkts():
    print(talus)
```

4.2 Détails d'implémentation

Le code suit assez fidèlement le principe décrit à la section 2 page 2 .

4.2.1 Structures des données

La structure la plus importante de ce code est le tableau numpy, i.e. le *ndarray*[3], qui est utilisé pour les diverses matrices et vecteurs de la classe.

Les coordonnées des, disons, $n+1$ points de l'ensemble des talus sont mis bout à bout et à plat dans un même vecteur *x_courant* que l'on note ici X. Pour pouvoir les différencier on utilise la liste *talus_lengths* qui contient le nombre de points de chaque talus.
Si on appelle $t_0, \ \ldots, \ t_m$ les talus, on a :

$$X = (x_0, y_0, \ldots, x_n, y_n) = \left[\underbrace{x_0 \ y_0 \ \ldots \ x_k \ y_k}_{t_0} \quad \cdots \quad \underbrace{x_0 \ y_0 \ \ldots \ x_j \ y_j}_{t_m}\right]$$

avec talus_lengths = [nombre_points(t_0) $\ \ldots \ $ nombre_points(t_m)]

et $\sum_{i=0}^{m}$ nombre_points(t_i) $= n+1$.

Les segments issus de la triangulation, i.e. l'argument *edges* du constructeur, sont représentés par un tableau numpy de taille $(N, 2)$ où chaque élément est un couple d'index des points définissant ses extrémités.
Ainsi, avec le vecteur des points X tel que décrit plus haut, un index i de *edges* se réfère donc aux coordonnées x_i et y_i d'index $2*i$ et $2*i+1$ dans X.
Par exemple le tableau [[0 2] [1 3] [4 5]] indique qu'il y a 3 segments. Le second, défini par [1 3], lie les points d'index 1 et 3 dans la liste de tous les points de talus et c'est donc le segment dont les extrémités ont pour coordonnées $(X[2], X[3])$ et $(X[6], X[7])$.

3. https://numpy.org/doc/stable/reference/arrays.ndarray.html

Le reste de cette partie est consacrée à balayer de manière descendante mais non exhaustive, le code de *LSDisplacer*.

4.2.2 square

La méthode *square* consiste essentiellement en une boucle qui calcule dX itérativement et met à jour les coordonnées courantes *x_courant* :

```
def square(self):
        # ...
        alpha = 0.1
        ro = 0.1
        min_dx = np.inf
        norm_float = LSDisplacer.NORM_DX
        for i in range(LSDisplacer.MAX_ITER):
                dx = self.compute_dx()
                self.x_courant += alpha * dx[0]
                normdx = np.linalg.norm(dx[0], ord=np.inf)
                alpha =  (LSDisplacer.H * ro) / (2**0.5 * normdx) if normdx != 0 else 0.1
                min_dx = normdx if normdx < min_dx else min_dx
                if normdx < norm_float : #NORM_DX :
                        break
                if LSDisplacer.FLOATING_NORM:
                        norm_float = LSDisplacer.NORM_DX if i < 100 else (LSDisplacer.NORM_DX + 2 * min_dx) / 3
        # ...
        return self.x_courant
```

Deux éléments notables :

— pour limiter la divergence, on multiplie en réalité dX par un facteur d'amortissement [4] *alpha*

— au dessus de 100 itérations, on augmente la valeur du seuil pour le critère d'arrêt sur la norme de dX. On fait une moyenne pondérée entre le minimum atteint jusqu'à présent ($\frac{2}{3}$) et la valeur originale de *NORM_DX* ($\frac{1}{3}$).
Ce comportement est désactivable en passant la valeur de *LSDisplacer.FLOATING_NORM* à *False*

4.2.3 compute_dx

Rien de particulier, on construit les matrices A et B à chaque itération, et le système est résolu à l'aide de NumPy :

```
import numpy as np
# ...
def compute_dx(self):
        # if LSDisplacer.KKT:
                # obsolète ...
        A = self.get_A()
        B = self.get_B()
        atp = A.T @ self.P
```

4. https://en.wikipedia.org/wiki/Non-linear_least_squares#Shift-cutting

```
        atpa = atp @ A
        atpb = atp @ B
        dx = np.linalg.lstsq(atpa, atpb, rcond=None)
        return dx
```

4.2.4 Matrices P, A et B

Ces matrices sont construites à l'aide des méthodes *get_X* ($X = A, B$ ou P) avec la même logique, en empilant les parties liées à chaque type de contrainte. P est créée une fois pour toute lors de l'instanciation alors que les matrices A et B sont elles régénérées à chaque itération, puisqu'elles dépendent de la solution courante.

```
def get_X(self):
        X = None
        if LSDisplacer.ID_CONST:
                # construire bloc contrainte mouvement
        if LSDisplacer.ANGLES_CONST:
                # construire bloc contrainte angles
                # l'ajouter au bloc précédent
        if LSDisplacer.EDGES_CONST:
                # construire bloc contrainte longueurs segments
                # l'ajouter au bloc précédent
        if LSDisplacer.DIST_CONST:
                # construire bloc contrainte longueurs segments
                # l'ajouter au bloc précédent
        return X # A ou B..
```

P

Matrice diagonale, créée par la méthode *get_P* dont la partie notable est celle où l'on ajoute les poids pour les contraintes sur les segments issus de la triangulation : pour les segments intra-talus on sépare les segments contraints des autres, et pour les segments inter-talus on sépare ceux qui sont proches ou lointains (longueur $>$ *edges_dist_max*) :

```
#...
if LSDisplacer.EDGES_CONST:
        wEdges = []
        for i, e in enumerate(self.edges):
                same_talus = num_talus(e[0], self.talus_lengths) == num_talus(e[1], self.talus_lengths)
                non_consecutive_points = abs(e[0] - e[1]) != 1
                if same_talus:
                        if non_consecutive_points:
                                wEdges.append(LSDisplacer.PEdges_int_non_seg)
                        else:
                                wEdges.append(LSDisplacer.PEdges_int)
                else:
                        if edge_length(e, self.points_talus) >= self.edges_dist_max:
                                wEdges.append(LSDisplacer.Pedges_ext_far)
                        else:
                                wEdges.append(LSDisplacer.PEdges_ext)
```

```
        wEdges = np.array(wEdges)
        weights.append(wEdges)
# ...
```

B

Matrice colonne, construite par *get_B* en soustrayant des données originales (coordonnées, produits vectoriels normés des angles, ...) celles qui sont recalculées pour la solution courante, à l'aide :

— de la méthode *angles_crossprod* pour les produits vectoriels normés
— de la fonction *edge_length* pour les longueurs de segments
— de la méthode *dist_F_vectorized* pour les distances aux routes

edge_length Calcule la longueur d'un segment défini par le couple de ses extrémités dans *edges* (voir 4.2.1).

angles_crossprod Rien de particulier, on parcourt la liste des coordonnées en la séparant en talus puis on calcule le produit vectoriel normé des vecteurs entourant chaque point non extrême.

dist_F_vectorized Une subtilité pour la méthode *dist_F_vectorized*, est qu'elle attend en entrée (en sus d'une route) une matrice de points de talus au lieu d'un seule ligne de points. Cela est dû au besoin de performance pour le calcul de la dérivée numérique comme on le verra à la partie suivante (on calcule la distance pour quatre fois le nombre de points, puisqu'on ajoute/retranche un petit h à chaque coordonnée).
Aussi, on ajoute une dimension au tableau numpy des points de talus avant d'appeler le calcul de distance.
L'indice i est l'index de la route dans le tableau de routes/distances passé au constructeur : il permet d'associer la bonne distance à la route traitée :

```
def get_B(self):
        #...
        # distance from roads
        if LSDisplacer.DIST_CONST:
                r_dists = []
                for i, r in enumerate(self.roads_shapes):
                        fk = - self.dist_F_vectorized(r, i, self.x_courant[np.newaxis,:])
                        r_dists.append(fk.item())
                # ...
        return b
```

La méthode *dist_F_vectorized* construit une multiligne par ligne de points de talus et calcule la distance à la route donnée pour chacune de ces multilignes, comme définie en (4) :

```
def dist_F_vectorized(self, road, i, points_array):
        ml = []
        for c in points_array:
```

```
            m = LSDisplacer._multiline_from_points(c, self.talus_lengths)
            ml.append(m)
    ml = np.array(ml)
    dists = pygeos.distance(road, ml)
    dists = np.where(dists > self.buffers[i], 0., self.buffers[i] - dists)
    return dists
```

A

Matrice des dérivées partielles, elle est construite par *get_A*.
Le sous-bloc correspondant aux dérivées de la contrainte de mouvement est la matrice identité $I_{2(n+1)}$ pour $(n+1)$ points. Les autres sous-blocs sont générés par :

— *cross_norm_diff* pour le produit vectoriel normé
— *edge_length_diff* pour la longueur des segments
— *dist_F_diff* pour la dérivée numérique de la distance à une route

edge_length_diff Les dérivées partielles pour la longueur d'un segment (équation (3)), se calculent par rapport aux coordonnées des sommets du segment, disons $A(x_a, y_a)$ et $B(x_b, y_b)$ et le code de *edge_length_diff* n'est qu'une transcription de ces équations :

$$\frac{\partial L}{\partial x_a} = \frac{x_a - x_b}{|AB|}$$
$$\frac{\partial L}{\partial y_a} = \frac{y_a - y_b}{|AB|}$$
$$\frac{\partial L}{\partial x_b} = \frac{-(x_a - x_b)}{|AB|}$$
$$\frac{\partial L}{\partial y_b} = \frac{-(y_a - y_b)}{|AB|}$$

cross_norm_diff De même, les dérivées partielles pour le produit vectoriel normé (équation (2)) se calculent par rapport aux coordonnées des 3 points qui définissent l'angle. Le calcul étant plus long et un peu technique, on a fait de la dérivation symbolique avec SymPy pour obtenir les équations implémentées dans *cross_norm_diff*.
On pourra se référer au fichier `sympy_differentials.py`.

dist_F_diff Pour calculer la dérivée numérique de manière pas trop lente, on va ≪vectoriser≫ les calculs de distance : on crée une matrice où chaque ligne i correspond aux coordonnées des points auxquelles on a ajouté (ou retranché) un h à la $i^{\text{ème}}$ coordonnée.
Cela se fait de manière assez élégante en utilisant le *broadcasting* de NumPy. On ajoute (ou retranche) au vecteur de coordonnées courantes une matrice diagonale de h :

$$\begin{pmatrix} x_0 \\ y_0 \\ \vdots \\ x_n \\ y_n \end{pmatrix} \pm \begin{pmatrix} h & 0 & \dots & & \dots & 0 \\ \vdots & & & & & \vdots \\ \vdots & & & & & \vdots \\ 0 & \dots & & \dots & 0 & h \end{pmatrix} = \begin{pmatrix} x_0 \pm h & y_0 & \dots & & \dots & y_n \\ x_0 & y_0 \pm h & x_1 & \dots & & \vdots \\ \vdots & & & & & \vdots \\ x_0 & \dots & & & \dots & y_n \pm h \end{pmatrix}$$

Ce qui permet de calculer $d_r(X \pm h_i)$ en une fois à l'aide de la vectorisation de *pygeos.distance* utilisée dans la méthode *dist_F_vectorized* :

```
def dist_F_diff(self, road, i):
        # diagonal matrix with H on diagonal
        h = np.eye(self.nb_vars) * self.H
        coords_plus_H = self.x_courant + h
        coords_minus_H = self.x_courant - h
        # seems a bit faster to have 2 np arrays instead of the same one splitted
        d_plus = self.dist_F_vectorized(road, i, coords_plus_H)
        d_min = self.dist_F_vectorized(road, i, coords_minus_H)
        ds = (d_plus - d_min) / (2 * self.H)
        return ds
```

4.3 Triangulation

Les fonctions qui gèrent la triangulation et les segments sont regroupées dans le module `triangulation.py`.
LSDisplacer attend en entrée des segments issus d'une triangulation de Delaunay contrainte effectuée sur l'ensembles des points de tous les talus.
Le calcul de cette triangulation est délégué à la librairie *Triangle*, qui n'est qu'un wrapper Python autour de la (rapide) librairie C du même nom de Jonathan Richard Shewchuk. [5]

Pour générer les segments, on appelle la fonction :

```
def get_edges_from_triangulation(points_talus, talus_lengths, decimate=False)
```

Elle attend en entrée un tableau numpy des points de talus, et la liste du nombre de points pour chaque talus, au format présenté dans la section précédente pour *LSDisplacer*.
En sortie on obtient une liste de segments, sous la forme décrite précédemment, i.e. une liste de couples d'index des extrémités des points.

Le paramètre *decimate* a été utilisé pour tester l'impact de la décimation de certaines arêtes de la triangulation, afin d'en mesurer l'impact sur la vitesse/qualité de l'ajustement.
On a essayé d'éliminer les segments inter-talus ayant pour origine le même point et formant un angle assez petit. Cependant, aucun effet intéressant n'ayant été observé, ce paramètre est caduc, et laissé à *False*. On garde donc l'ensemble des segments.

5. `http://www.cs.cmu.edu/~quake/triangle.html`

4.4 Code utilitaire

Les fonctions se trouvant dans `shapes_and_geoms_stuff.py` servent aux divers scénarios d'exemples d'utilisation du code.
Elles sont essentiellement utilisées pour lire les données en entrée et les passer à *LSDisplacer* et sont donc fortement liées au format de celles-ci (shapefiles de lignes, etc.).
On notera l'utilisation de la bibliothèque Shapely ici, le besoin de vectoriser des calculs n'étant pas présent.

4.4.1 get_points_talus

```
def get_points_talus(shapely_lines)
```

Prend en entrée une liste de géométries Shapely de lignes et renvoie un numpy array de l'ensemble des points de ces lignes.

4.4.2 get_STRtrees

```
def get_STRtrees(network_file, talus_file)
```

Prend en entrée le chemin vers les shapefiles des routes et talus, et construit deux *STRtree*[6], qui vont permettre d'avoir un index spatial rapide sur les deux collections de lignes.

Ce qui va permettre d'accélérer la sélection de routes ou de talus, pour un îlot donné.

4.4.3 get_talus_inside_face

```
def get_talus_inside_face(f, ttree, merge=True, displace=True)
```

Prend en entrée un polygone f représentant un îlot urbain, le *STRtree* des talus, et renvoie les intersections de tous les talus et de cet îlot.
Si *merge* est à *True*, on essaie de fusionner les talus consécutifs en un seul talus.
Si *displace* est à *True*, on essaie de déplacer les talus qui sont sur la frontière de l'îlot à l'intérieur, soit en les bougeant vers le centroïde de la face si c'est possible, ou alors de manière aléatoire.

4.4.4 get_roads_for_face

```
def get_roads_for_face(f, ntree, merge=True)
```

Fonction symétrique à la précédente, mais pour les routes. On récupère toutes les routes composant l'îlot urbain formé par le polygone f, en essayant d'en fusionner le maximum si *merge* est à *True*. En particulier, la frontière extérieure de f mais aussi les frontières des éventuels trous à l'intérieur de f forment chacune une seule route.

6. https://shapely.readthedocs.io/en/stable/manual.html#str-packed-r-tree

4.5 Exemples d'utilisation

Le dépôt comporte quelques exemples d'utilisation de la classe *LSDisplacer*.

4.5.1 main_simple_example.py

Exemple basique d'utilisation : les routes et talus sont des listes de Linestrings au format WKT qu'on charge avec *Shapely*.
Le programme est explicite, et suffisamment court pour être reproduit ici :

```
from shapely.wkt import loads
from shapes_and_geoms_stuff import get_points_talus
from triangulation import get_edges_from_triangulation
from displacer import LSDisplacer

roads = [('LineString(...)', 10), ('LineString(...)', 20)]
talus = ['LineString(...)', 'LineString(...)', 'LineString(...)']

talus_shapes = [loads(t) for t in talus]
talus_lengths = [len(t.coords) for t in talus_shapes]

points_talus = get_points_talus(talus_shapes)
edges = get_edges_from_triangulation(points_talus, talus_lengths)

LSDisplacer.set_params(MAX_ITER=500, NORM_DX=0.3,
                       PFix=8.0, PDistRoads=200, PAngles=8,
                       PEdges_ext=15, Pedges_ext_far=0.5,
                       PEdges_int=1, PEdges_int_non_seg=1)

displacer = LSDisplacer(points_talus, roads, talus_lengths, edges)
print("P matrix size", displacer.P.shape)
displacer.square()
displacer.print_linestrings_wkts()
```

4.5.2 main_by_faces_example.py

Ici, on a un ensemble d'îlots sous forme d'un shapefile de polygones, et on utilise les fonctions utilitaires présentées en 4.4 pour récupérer routes et talus pour chaque îlot.
Le programme illustre aussi l'utilisation du *logger* de la classe basé sur le module *logging* de la librairie standard : les niveaux utilisés ici, par ordre croissant de sévérité, *DEBUG*, *INFO* et *WARNING*, sont par verbosité décroissante (*DEBUG* est le niveau le plus verbeux). Par défaut on est au niveau *INFO*.

4.5.3 multiproc_expe.py

Variation du programme précédent en version multi-cœurs. Chaque îlot est traité par une unité de calcul disponible.
De plus, on partitionne les îlots en groupes de talus proches (leurs distances sont inférieures ou égales à *edges_dist_max*) et on ne récupère que les routes

concernées afin de réduire la taille des matrices et le nombre de routes impliquées dans les calculs de distances.