TIPE : Minimisation de l'énergie de Möbius pour le dénouage de nœuds

Lokmane Mohamed-Yassine

n°17629

Session 2025

Illustration introductive

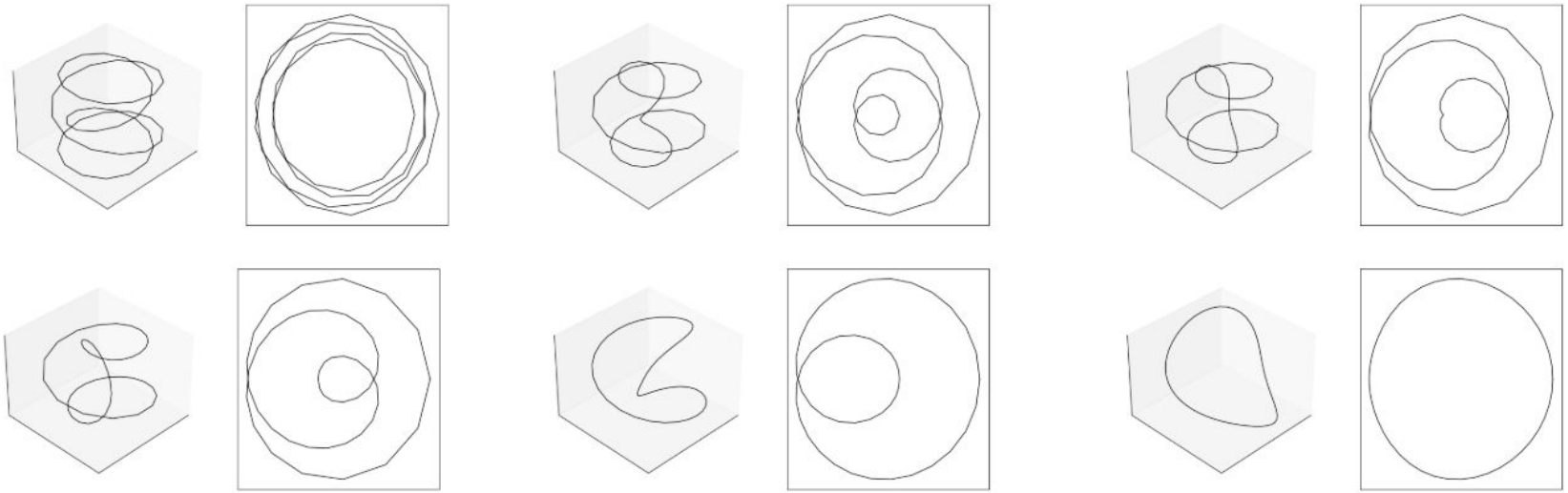
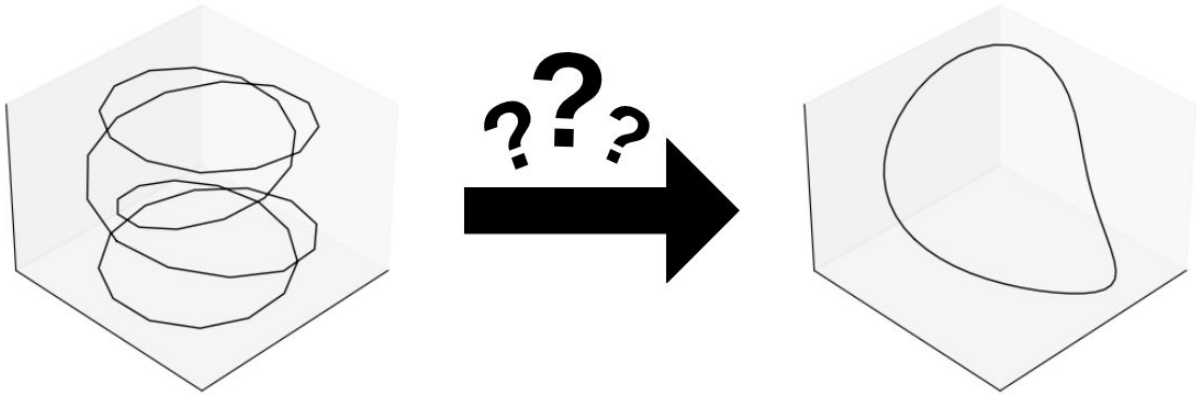


Illustration des étapes de dénouage d'un nœud complexe

Problématique

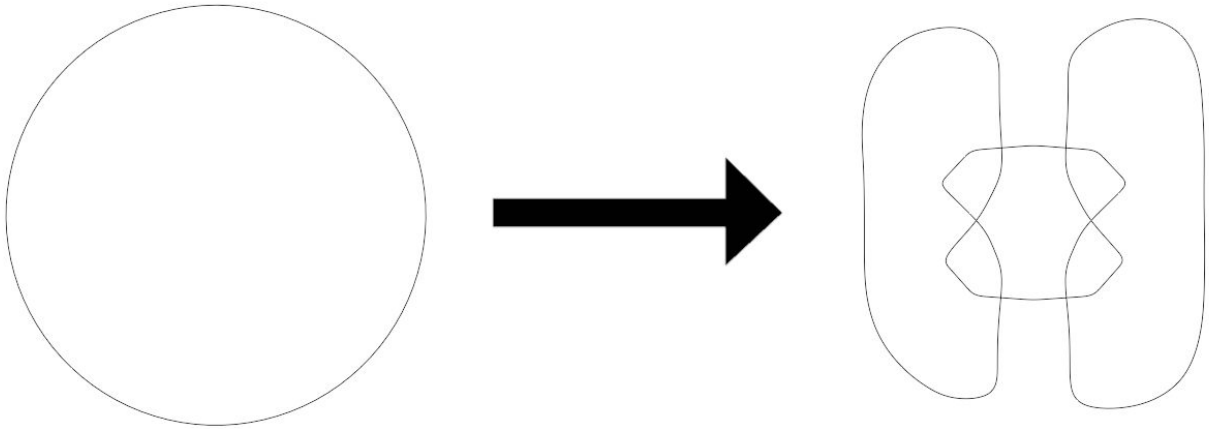
Question centrale

Comment concevoir un algorithme efficace pour dénouer un nœud ?



Plan de la présentation

Exemple de configuration géométrique

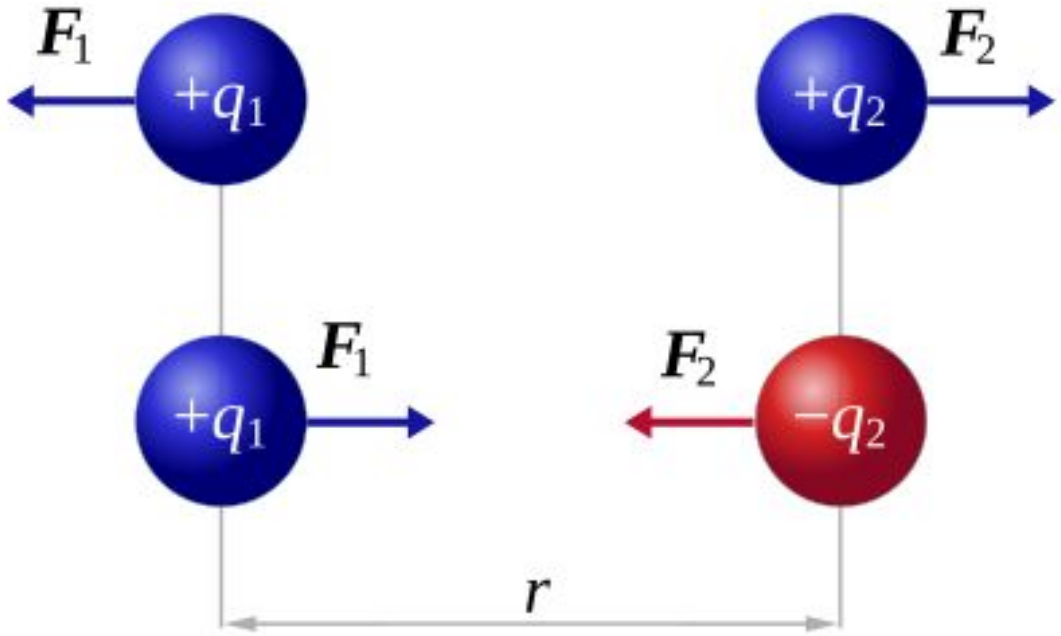


$$\gamma : \mathbb{S}^1 \to \mathbb{R}^3$$

Propriété

γ est une application continue et injective : elle définit un nœud fermé sans auto-intersection.

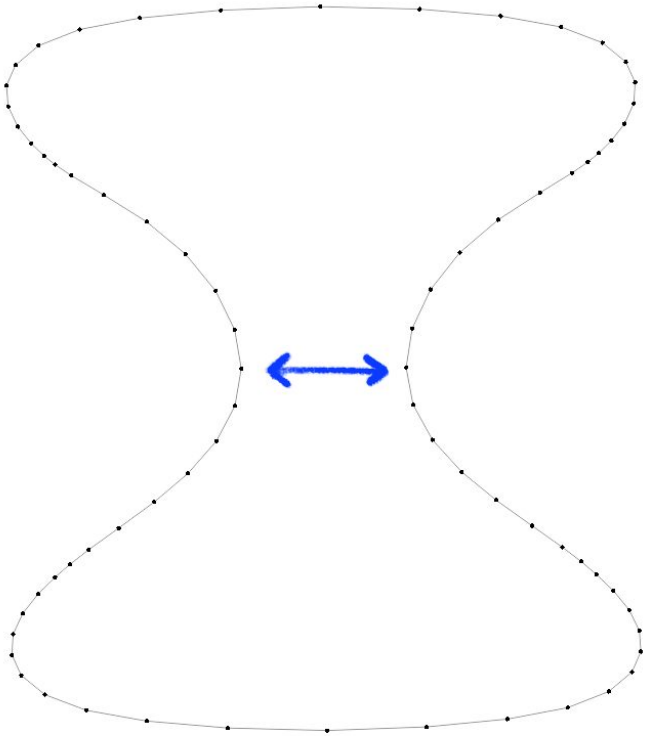
Motivation géométrique



$$\left|F_1\right| = \left|F_2\right| = k_e \frac{|q_1 \times q_2|}{r^2}$$

Répulsion par la force électrostatique.

Expression de l'énergie d'une paire

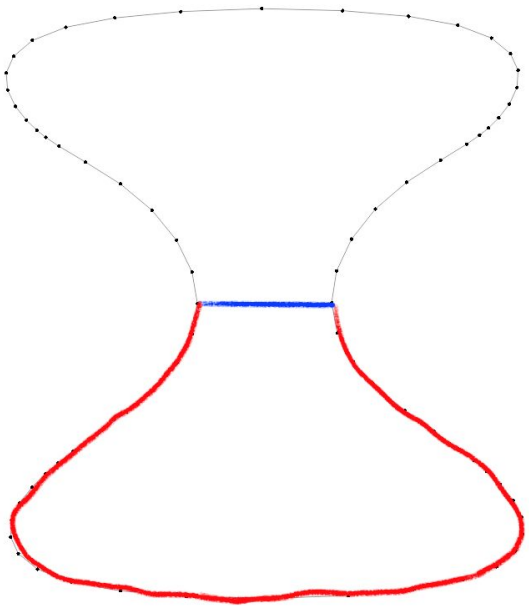


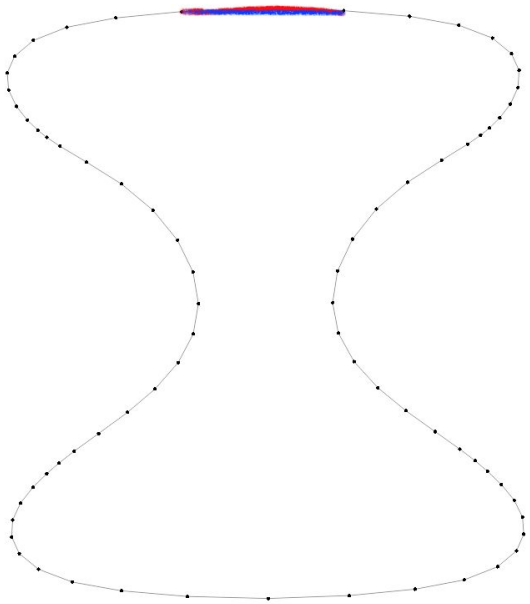
Répulsion entre points proches.

$$E(s,t) = \frac{1}{|\gamma(s) - \gamma(t)|^2}$$

Distance euclidienne et distance sur la courbe

$d_\gamma(s,t)$ $\|\gamma(s) - \gamma(t)\|$





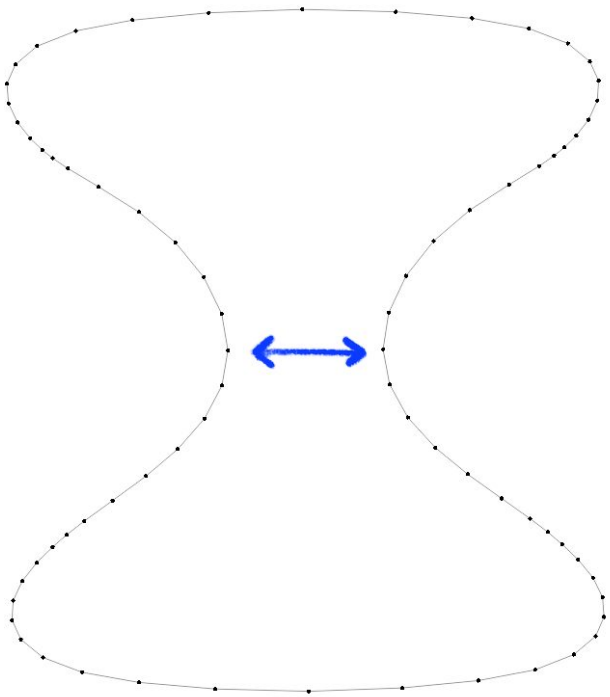
$d_\gamma(s,t) = 5{,}507$

$\|\gamma(s) - \gamma(t)\| = 0{,}301$

$d_\gamma(s,t) = 0{,}361$

$\|\gamma(s) - \gamma(t)\| = 0{,}361$

Correction par la distance intrinsèque



Répulsion entre points proches, corrigée avec distance intrinsèque

$$E(s,t) = \frac{1}{\|\gamma(s) - \gamma(t)\|^2} - \frac{1}{d_\gamma(s,t)^2}$$

Énergie de Möbius

Formule finale de l'énergie de Möbius

$$E(\gamma) = \iint_{\mathbb{S}^1 \times \mathbb{S}^1} \left(\frac{1}{\|\gamma(s) - \gamma(t)\|^2} - \frac{1}{d_\gamma(s,t)^2} \right) ds\, dt$$

- $\|\gamma(s) - \gamma(t)\|$: distance euclidienne entre deux points de la courbe,
- $d_\gamma(s,t)$: distance le long de la courbe,
- L'intégrale exprime une énergie répulsive globale entre tous les couples de points.

Propriétés fondamentales de l'énergie de Möbius

L'énergie de Möbius est conçue pour ne dépendre que de la **forme intrinsèque** du nœud.

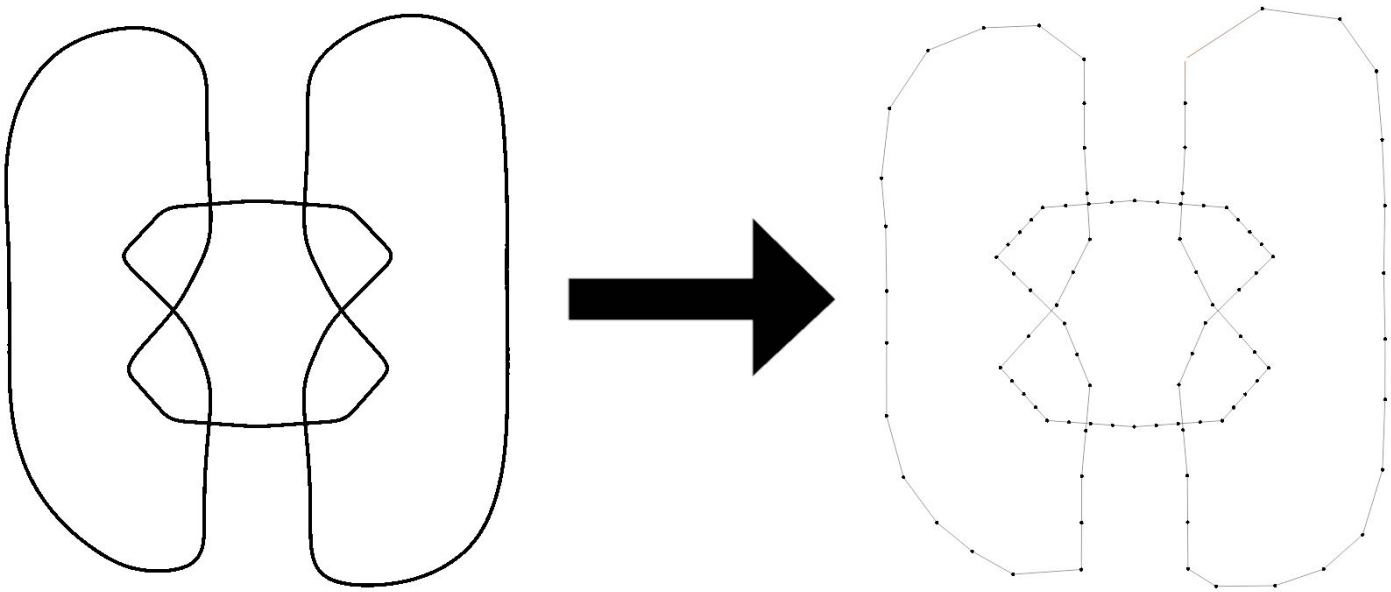
Remarque importante

L'énergie est invariante par :

- Translation,
- Rotation,
- Homothétie.

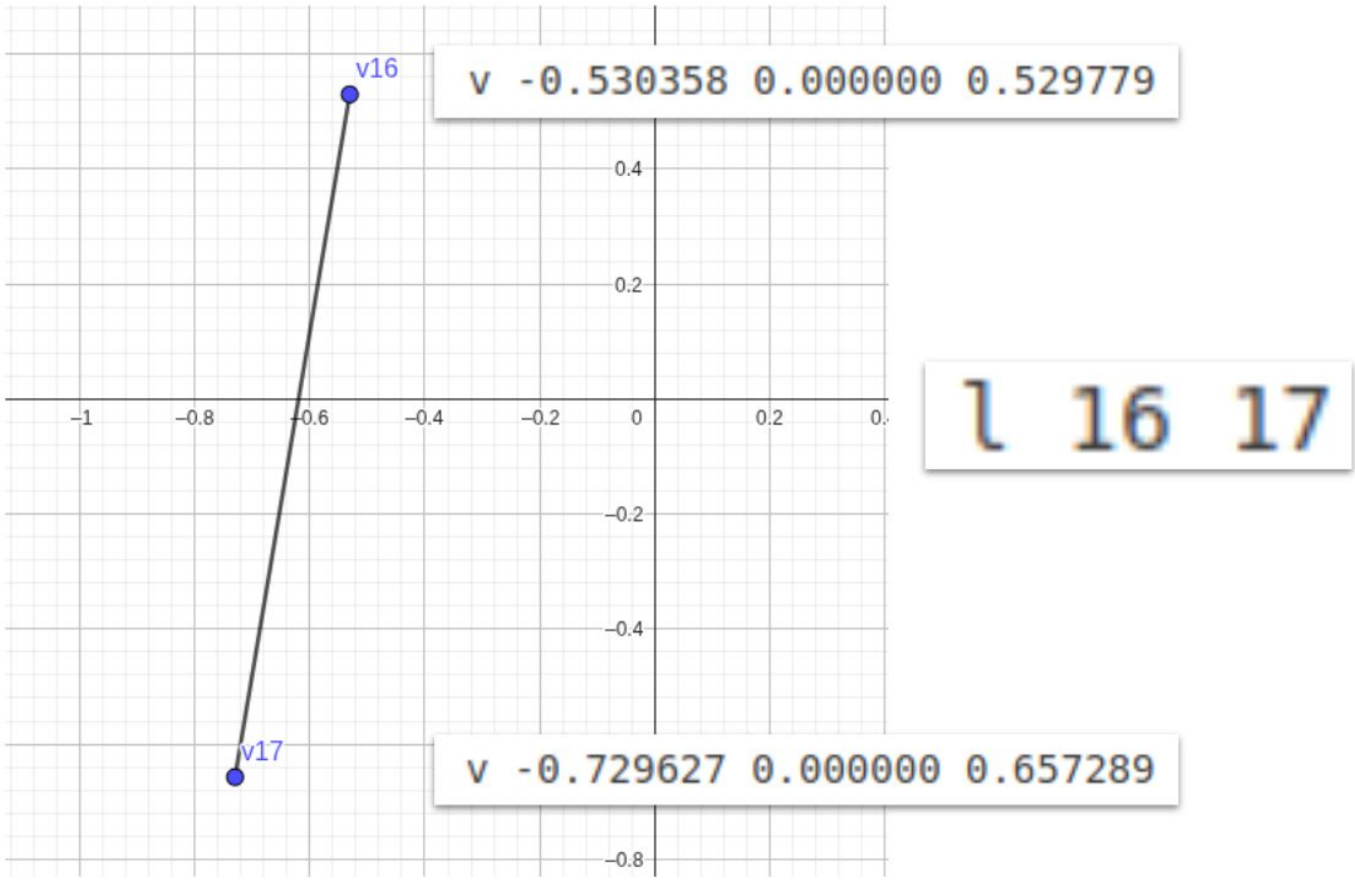
Algorithme de minimisation : stratégie et paramètres

Noeud fermé discrétisé en 95 sommets



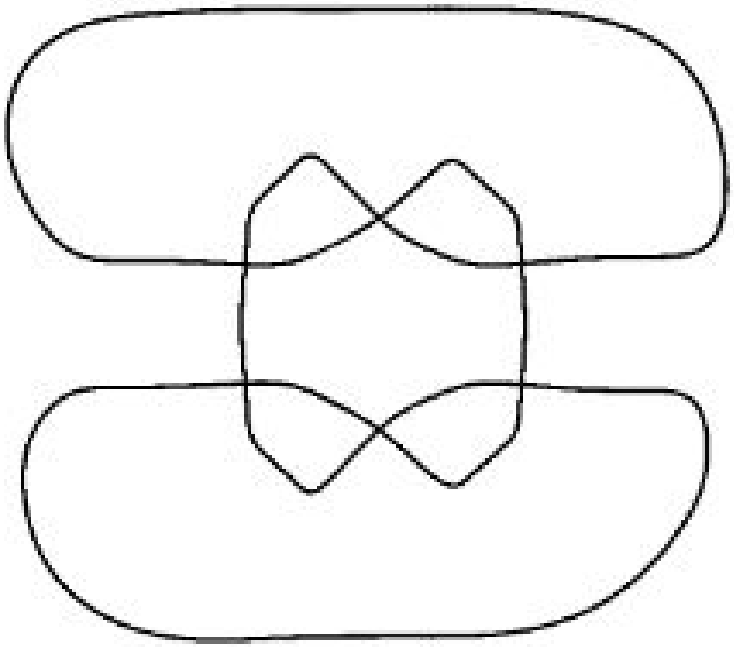
Le nœud initial est transformé en une courbe polygonale formée de 95 sommets : c'est la base de la discrétisation utilisée dans l'algorithme.

Représentation du fichier .obj



Exemple de représentation interne d'un nœud sous format `.obj`.

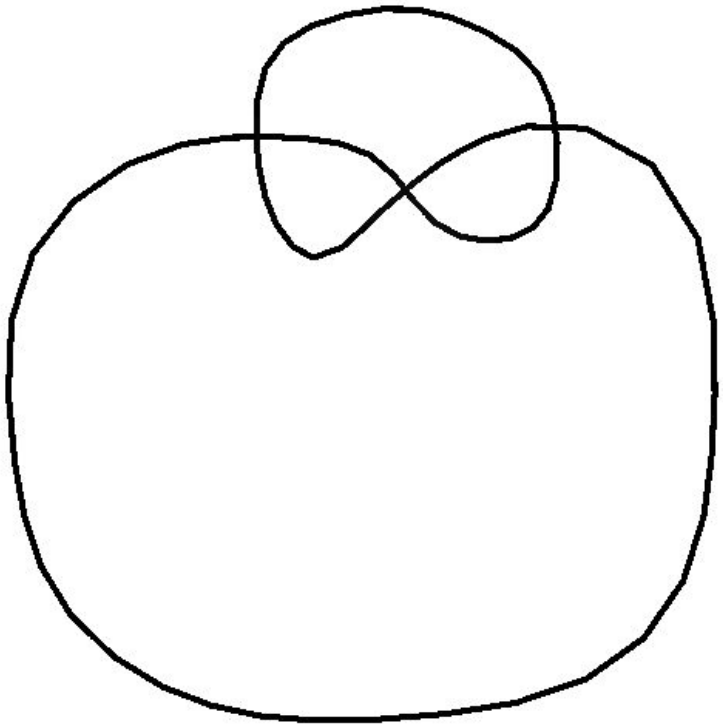
Exemple de nœud : nœud marin



Nœud marin dans sa configuration initiale.

Variante modélisée du même nœud marin.

Exemple de nœud : nœud simple



Vue initiale du nœud simple.

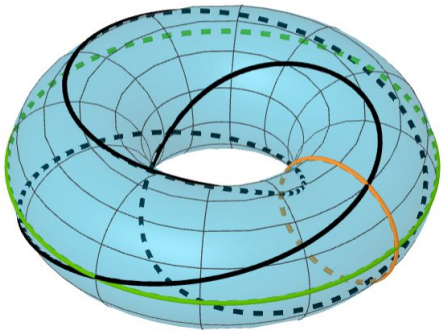
Le nœud simple en réalité.

Exemple de nœud : nœud torique

Nœud torique — équation paramétrique

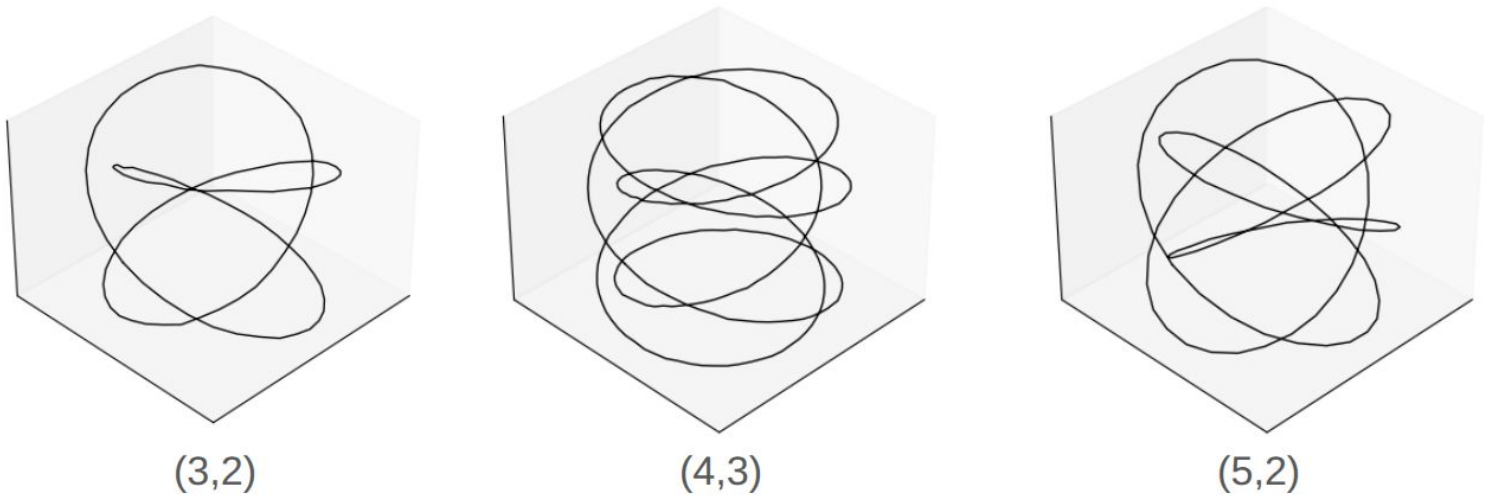
$$\begin{cases} x = [r_2 \cdot \cos(p \cdot \varphi) + r_1] \cdot \cos(p \cdot \varphi) \\ y = [r_2 \cdot \cos(p \cdot \varphi) + r_1] \cdot \sin(p \cdot \varphi) \\ z = -r_2 \cdot \sin(q \cdot \varphi) \end{cases}$$

- r_1 : rayon de révolution autour de l'origine,
- r_2 : rayon de révolution autour de l'axe du cercle,
- p : révolutions autour de l'origine,
- q : révolutions autour de l'axe du cercle.



Le point parcourt un cercle dont le centre lui-même décrit une révolution torique.

Nœud torique : exemple de rendu



Exemple visuel de nœuds toriques générés avec la paramétrisation précédente.

Algorithme de minimisation : stratégie et paramètres

Discrétisation

$$E(\gamma) = \sum_{u=1}^{N} \sum_{v=1}^{N} \left(\frac{1}{\|\gamma(u) - \gamma(v)\|^2} - \frac{1}{d(u,v)^2} \right)$$

- N : nombre de sommets du nœud (courbe discrétisée),
- $\gamma(u)$, $\gamma(v)$: positions des sommets u et v,
- $d(u,v)$: distance le long de la courbe entre les sommets.

Algorithme de minimisation : stratégie et paramètres

Descente de gradient

$$\gamma_t = -\nabla E(\gamma)$$

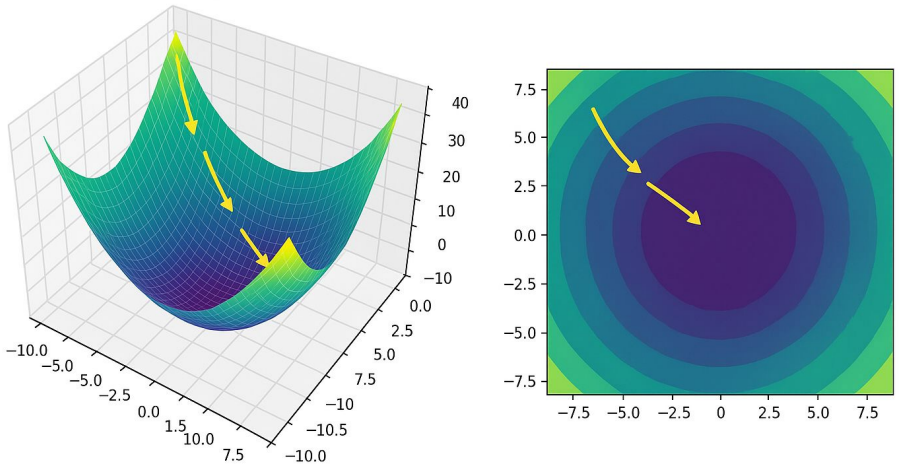


Schéma général d'une descente de gradient : la courbe suit la direction opposée au gradient de l'énergie.

Paramètre ε dans la descente de gradient

Pourquoi introduire un pas ε dans la mise à jour ?

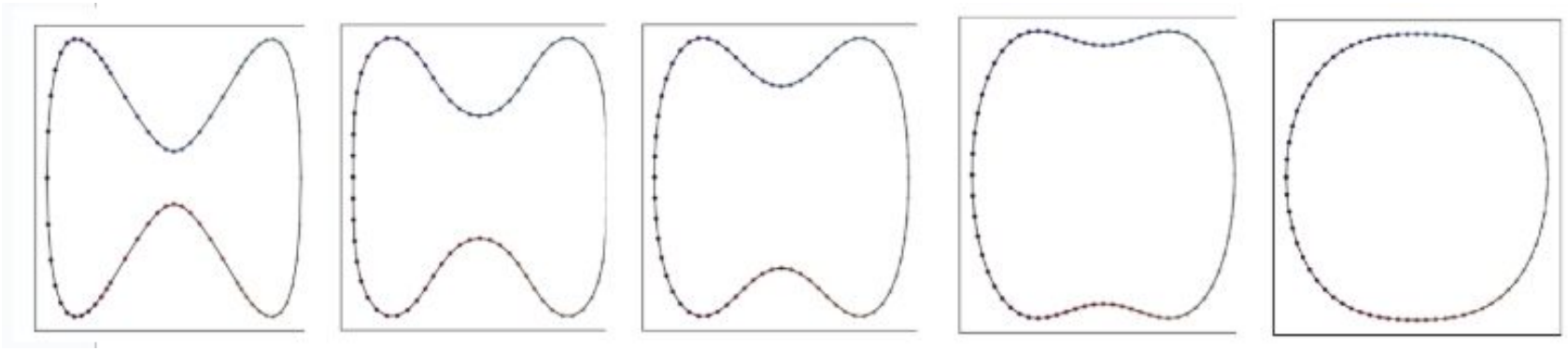
À chaque itération n, la courbe est mise à jour selon la règle :

$$\gamma_{n+1} = \gamma_n - \varepsilon \cdot \nabla E(\gamma_n)$$

- ε contrôle la taille du déplacement effectué par la descente :
  - Trop petit $\rightarrow$ stagnation de la courbe.
  - Trop grand $\rightarrow$ instabilité et distorsion du nœud.

Algorithme de minimisation : stratégie et paramètres

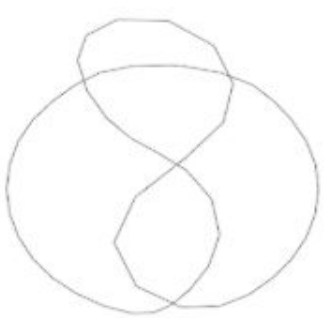
Premier exemple simple avec $\varepsilon = 0.1$



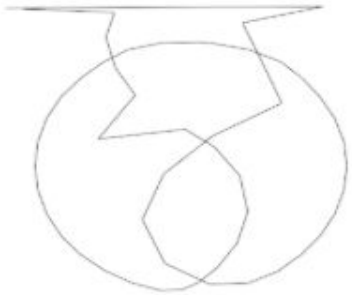
Itération 0 Itération 250 Itération 500 Itération 800 Itération 1000

Algorithme de minimisation : stratégie et paramètres

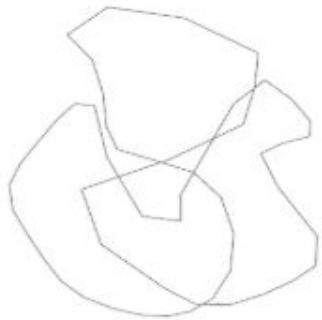
Effet de ε sur la stabilité de l'algorithme



$\varepsilon = 10^{-7}$



$\varepsilon = 10^{-6}$



$\varepsilon = 10^{-5}$

$\varepsilon = 10^{-4}$

Algorithme de minimisation : stratégie et paramètres

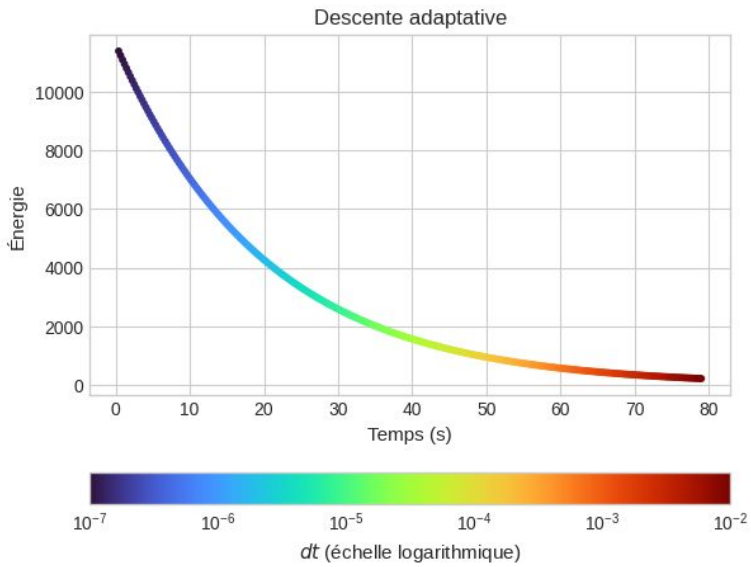
ε adaptatif

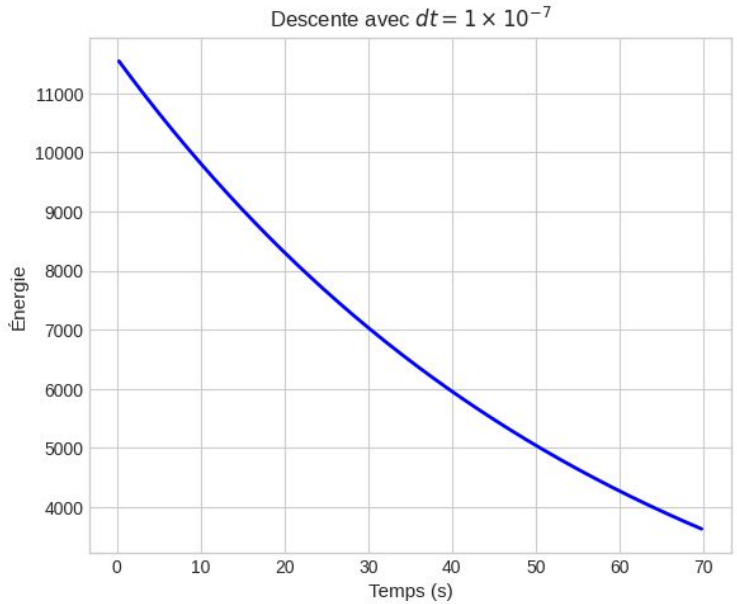
$$\varepsilon = \frac{d_{\min} \times \text{tolerance}}{\max(\nabla E)}$$

- $d_{\min}$: plus petite distance entre deux sommets,
- *tolerance* : paramètre qui guide la vitesse de dénouage.

Algorithme de minimisation : stratégie et paramètres

ε adaptatif : effet sur la convergence



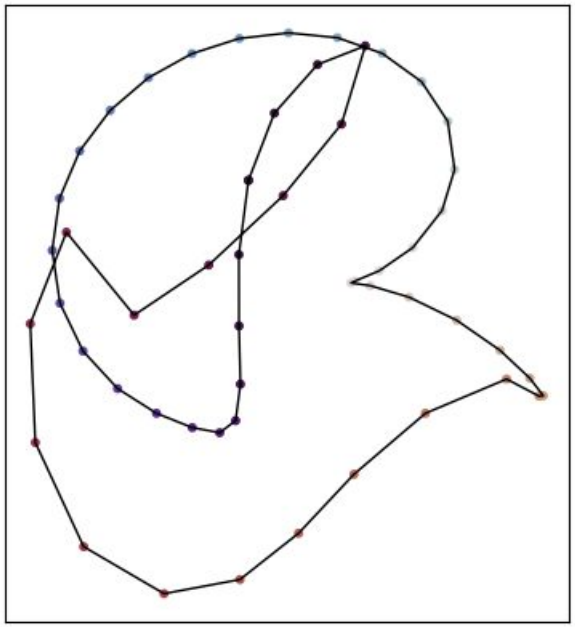


Descentre adaptative

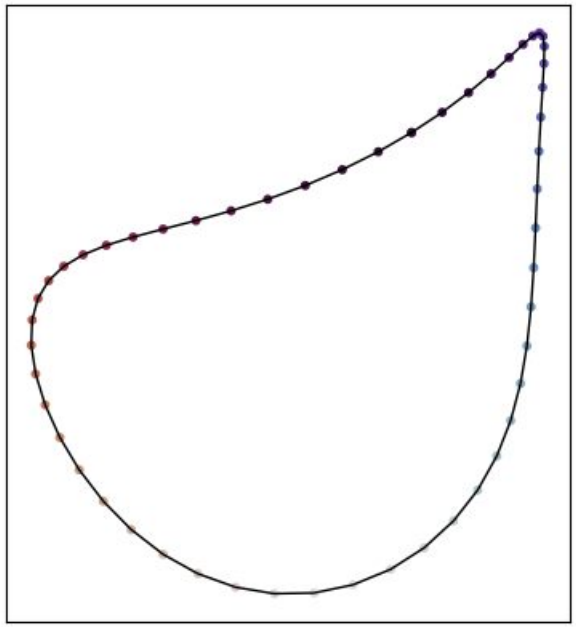
Descente avec $\varepsilon = 10^{-7}$

Algorithme de minimisation : stratégie et paramètres

Comparaison après 400 itérations



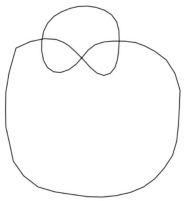
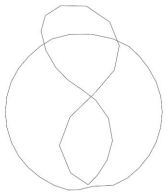
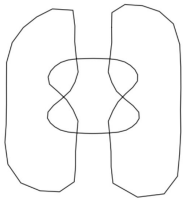
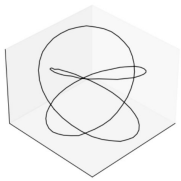
ε fixe



ε dynamique

Algorithme de minimisation : stratégie et paramètres

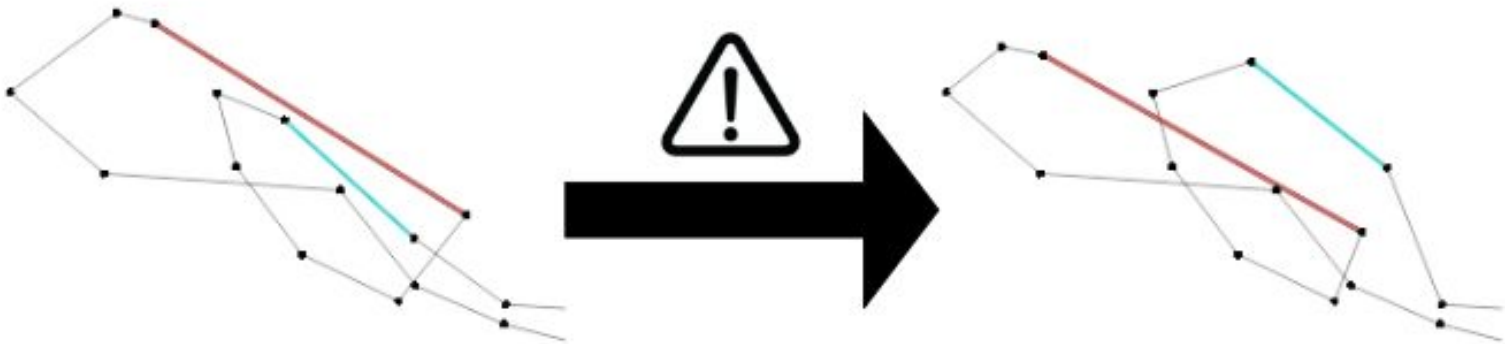
Résultats de l'algorithme

Nom	Illustration	Itérations	Temps
Nœud simple		1500	8min 10s
Nœud de huit		800	5min 24s
Nœud de marin		500	3min 05s
Nœud torique		600	3min 46s

Problématique de l'autointersection

Définition

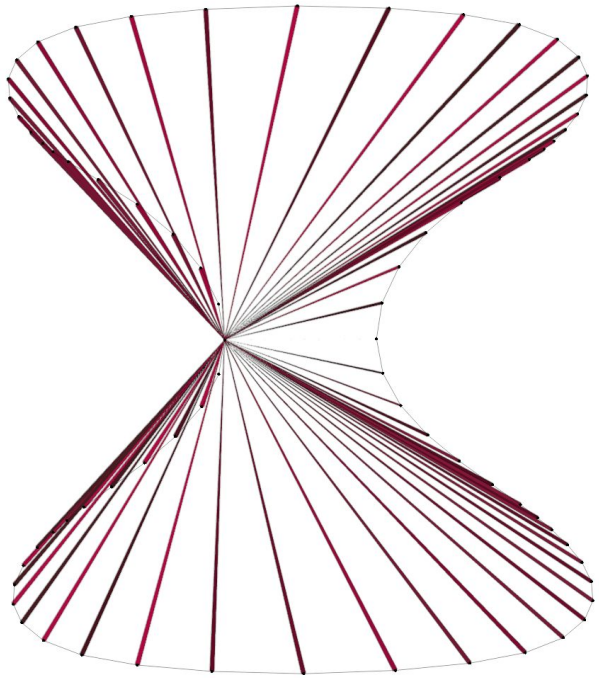
Une autointersection survient lorsqu'un nœud se croise lui-même, c'est-à-dire que deux parties de la courbe occupent la même position dans l'espace.



Optimisations, contraintes et perspectives

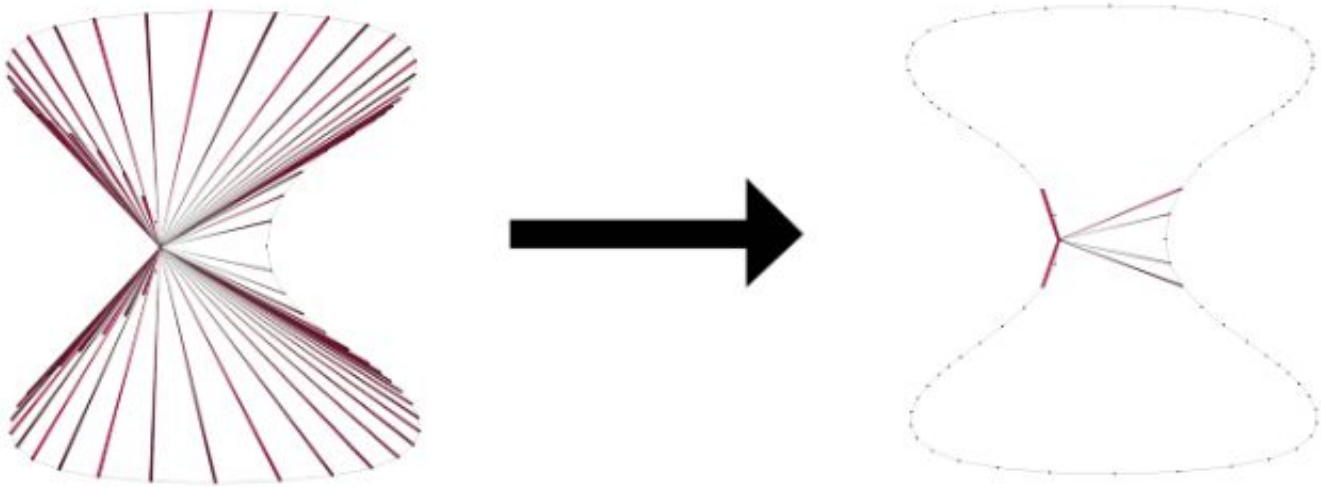
Algorithme actuel en $\mathcal{O}(N^2)$

Chaque sommet interagit avec tous les autres, ce qui entraîne une complexité quadratique.



Optimisations, contraintes et perspectives

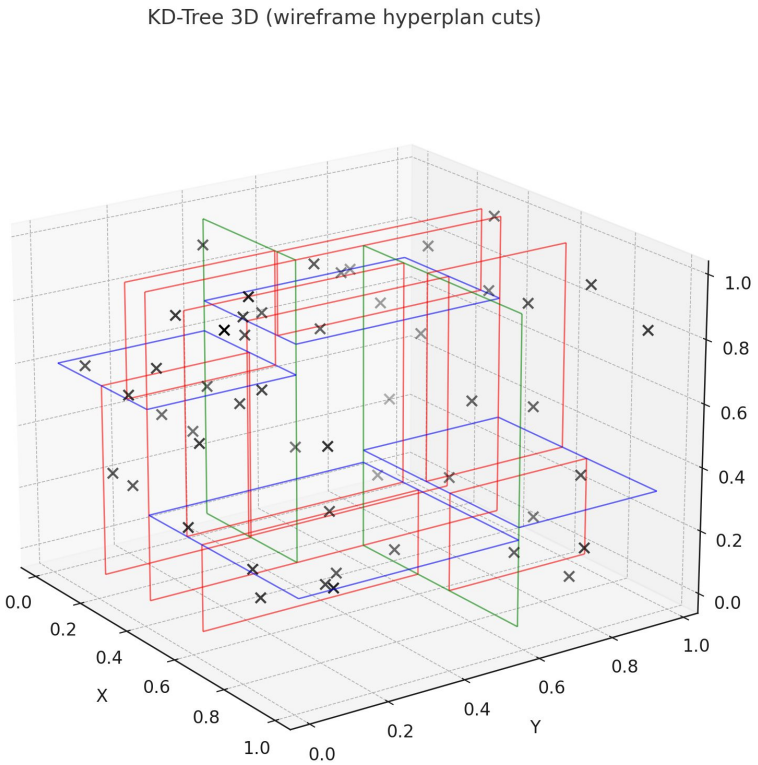
Élagage des paires



On traite uniquement les k plus proches sommets, ce qui réduit considérablement le nombre d'interactions à calculer.

Optimisations, contraintes et perspectives

Pour choisir les k plus proches voisins de chaque sommet :

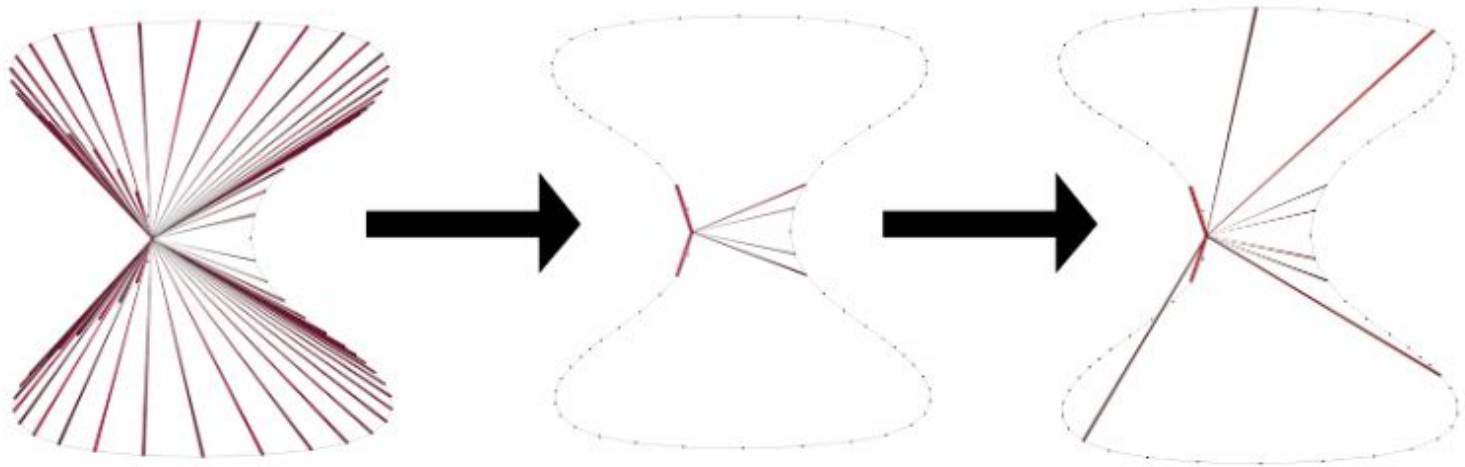


à chaque itération de l'algorithme :

1. Construction d'un k-d tree à partir des sommets — $\mathcal{O}(N \log N)$,
2. Pour chaque sommet, interrogation du k-d tree pour obtenir ses k plus proches voisins — $\mathcal{O}(\log N)$,
3. Restriction du calcul de l'énergie à ces voisins uniquement.

Optimisations, contraintes et perspectives

Ajout de points lointains par échantillonnage Monte Carlo



Ajout aléatoire de quelques sommets éloignés pour capturer les effets de longue portée.

Optimisations, contraintes et perspectives

Complexité finale

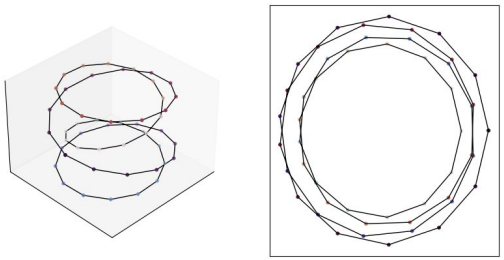
$$\mathcal{O}(N \log N + N(k+m))$$

Complexité estimée après optimisation (voisinage restreint + échantillonnage Monte Carlo).

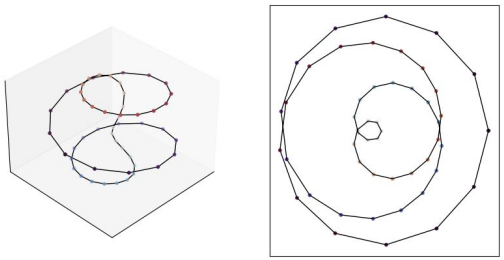
Avec k le nombre de voisins, et m le nombre de points échantillonnés.

Optimisations, contraintes et perspectives

Résultats visuels



Itération : 0



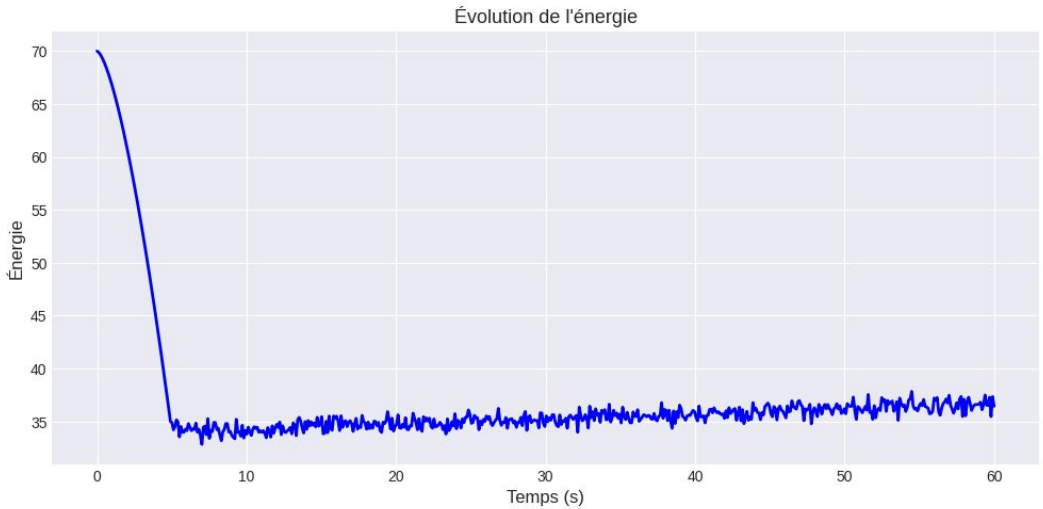
Itération : 800

Problème :

- k : trop petit, interactions importantes sont ignorées.
- k : trop grand, la compléxité redevient élevé.

Optimisations, contraintes et perspectives

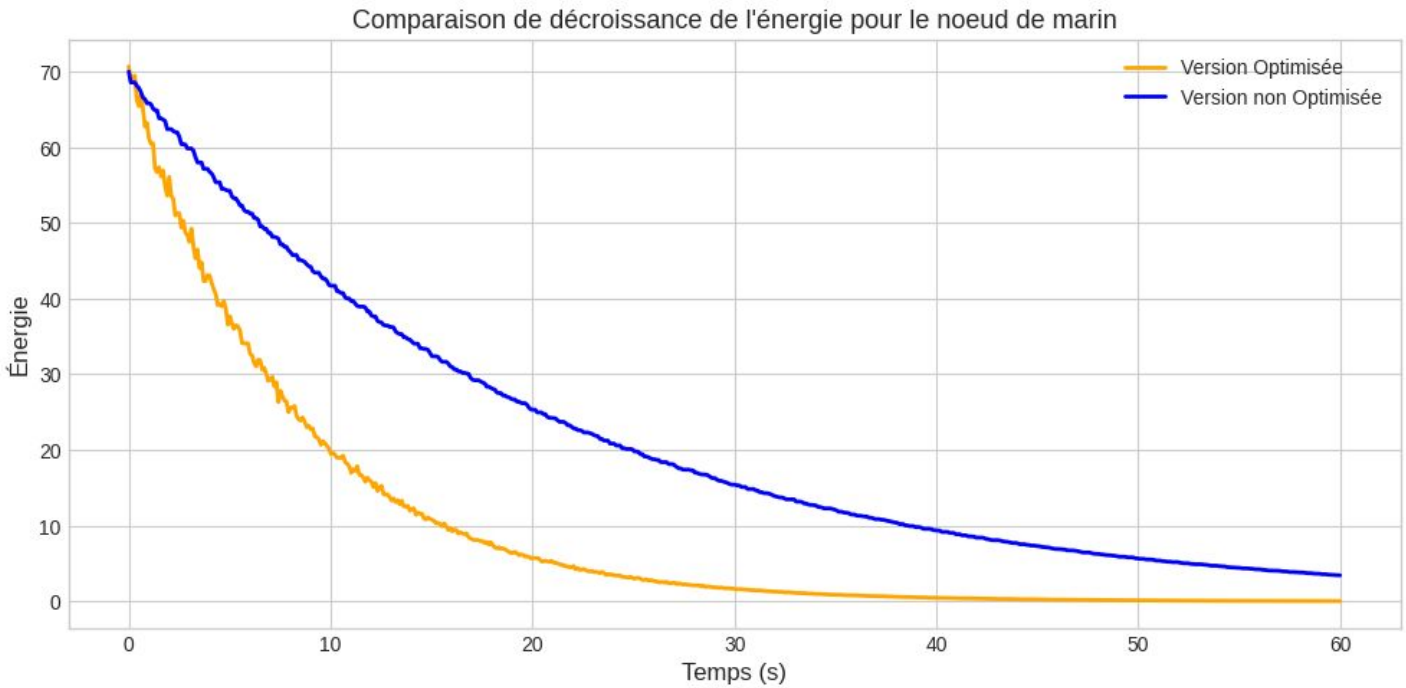
Élagage adaptatif



- Si l'énergie décroît de manière régulière, on conserve le paramétrage courant.
- Si l'énergie stagne, on augmente le nombre de voisins.

Optimisations, contraintes et perspectives

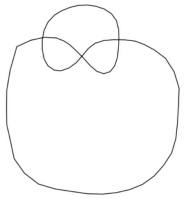
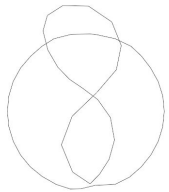
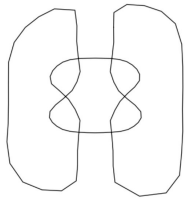
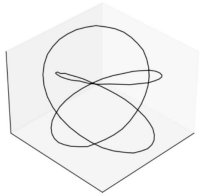
Élagage adaptatif — comparaison de performance



Comparaison de décroissance de l'énergie pour le nœud de marin : optimisation vs. version brute.

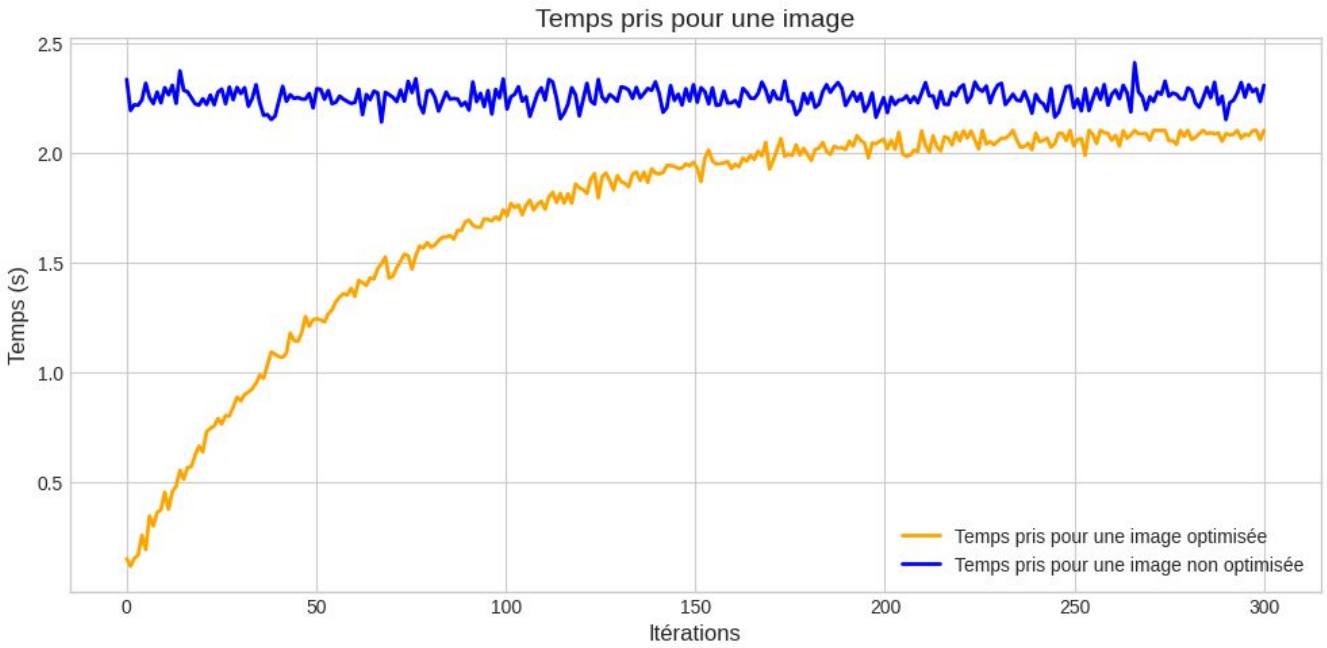
Optimisations, contraintes et perspectives

Élagage adaptatif — gain de performance

Nom	Illustration	Itérations	Non optimisé	Optimisé
Nœud simple		1500	8min 10s	3min 49s
Nœud de huit		800	5min 24s	2min 12s
Nœud de marin		500	3min 05s	1min 03s
Nœud torique		600	3min 46s	1min 18s

Optimisations, contraintes et perspectives

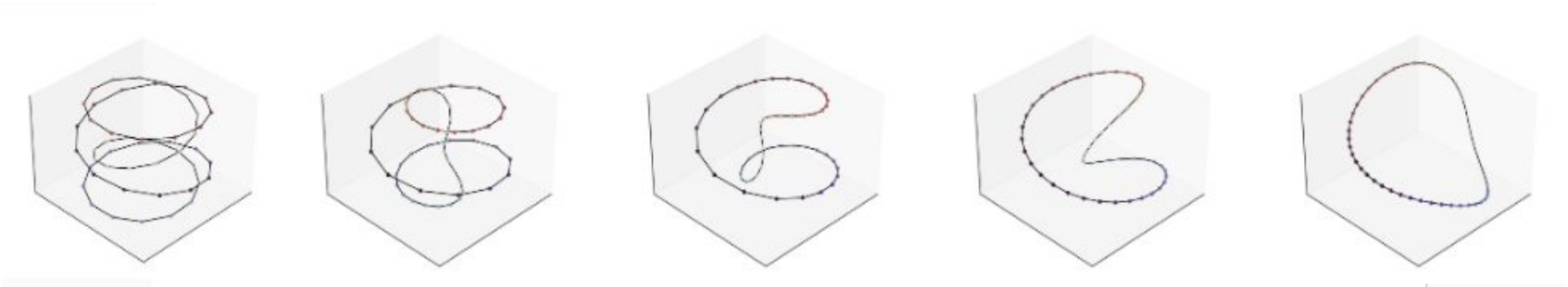
Élagage adaptatif — stabilité du temps de calcul



Version non optimisée et version optimisée.

Optimisations, contraintes et perspectives

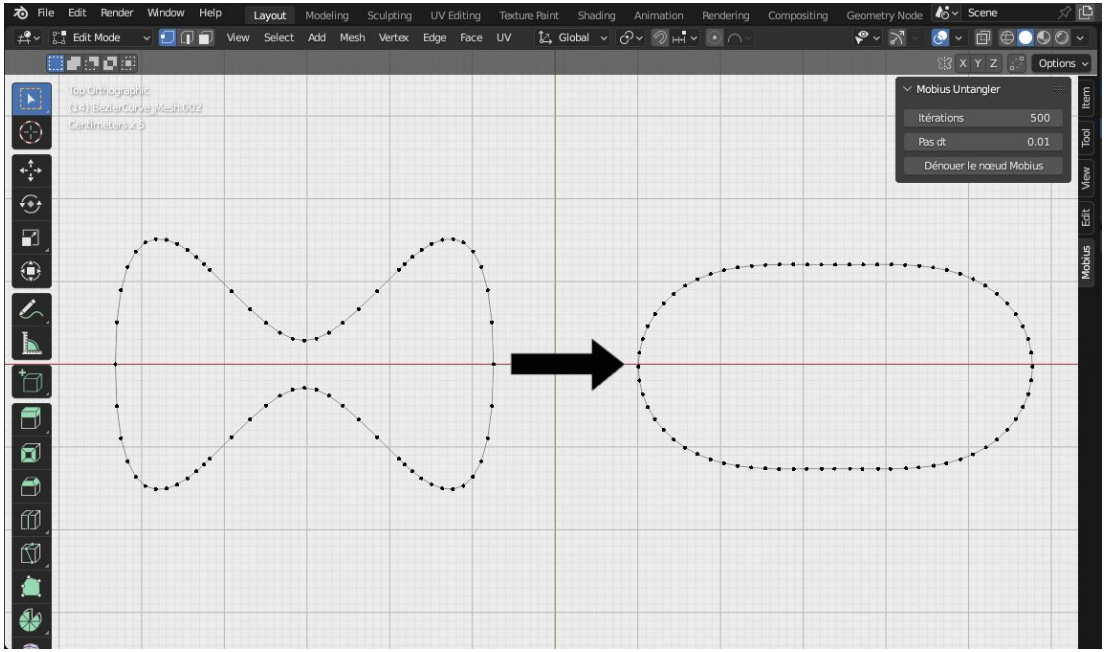
Élagage adaptatif — évolution visuelle du nœud

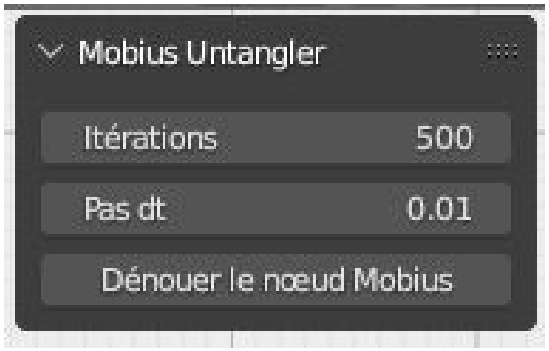


Visualisation de l'évolution d'un nœud torique.

Optimisations, contraintes et perspectives

Interface utilisateur du dénouage Möbius





Contrôle direct du processus : nombre d'itérations et pas ε.

Perspectives d'amélioration de l'algorithme

- Sensibilité aux paramètres manuels
- Non-unicité des minima : pièges locaux
- Lien entre convergence géométrique et dénouement topologique

Annexe

Détail du calcul de l'énergie discrétisée

On doit adapter la fonction de distance curviligne $d(u, v)$, où u, v sont des paramètres d'arc, à la courbe discrétisée γ.

La fonction $d(u, v)$ calcule la longueur du chemin le plus court P le long de la courbe entre les sommets $\gamma(u)$ et $\gamma(v)$, où u, v sont maintenant des indices :

$$d(u, v) = \sum_{ab \in P} \|\gamma(a) - \gamma(b)\|$$

où $\|\cdot\|$ désigne la norme euclidienne usuelle.

Annexe

Détail du calcul de l'énergie discrétisée

On cherche le gradient :

$$\nabla E = \begin{bmatrix} E_{\gamma(1)} \\ E_{\gamma(2)} \\ \vdots \\ E_{\gamma(N)} \end{bmatrix}$$

Chaque dérivée partielle $E_{\gamma(i)}$ s'obtient analytiquement :

$$E_{\gamma(i)} = \sum_u \sum_v \left[\frac{\partial}{\partial \gamma(i)} \left(\frac{1}{\|\gamma(u) - \gamma(v)\|^2} \right) - \frac{\partial}{\partial \gamma(i)} \left(\frac{1}{d(u,v)^2} \right) \right]$$

Le premier terme est appelé $A_i(u,v)$ et le second $B_i(u,v)$.

Annexe

Détail du calcul de l'énergie discrétisée

Soit $v \in \mathbb{R}^3$, on a :

$$\frac{d}{dv}\left(\frac{1}{\|v\|^2}\right) = -2\frac{v}{\|v\|^4}$$

On applique cette formule au cas $\gamma(u) - \gamma(v)$:

$$A_i(u,v) = \begin{cases} -2\frac{\gamma(i)-\gamma(v)}{\|\gamma(i)-\gamma(v)\|^4} & \text{si } i = u \\ +2\frac{\gamma(u)-\gamma(i)}{\|\gamma(u)-\gamma(i)\|^4} & \text{si } i = v \\ 0 & \text{sinon} \end{cases}$$

Annexe

Détail du calcul de l'énergie discrétisée

$$B_i(u,v) = -2\frac{1}{d(u,v)^3} \cdot \frac{\partial}{\partial\gamma(i)} d(u,v)$$

avec :

$$\frac{\partial}{\partial\gamma(i)} d(u,v) = \begin{cases} \frac{\gamma(i)-\gamma(i-1)}{\|\gamma(i)-\gamma(i-1)\|} - \frac{\gamma(i+1)-\gamma(i)}{\|\gamma(i+1)-\gamma(i)\|} & i \in P,\ i \neq u, v \\ \frac{\gamma(i)-\gamma(i-1)}{\|\gamma(i)-\gamma(i-1)\|} & i = u,\ i-1 \in P \\ -\frac{\gamma(i+1)-\gamma(i)}{\|\gamma(i+1)-\gamma(i)\|} & i = v,\ i+1 \in P \\ 0 & \text{sinon} \end{cases}$$

P est ici le chemin le plus court reliant les sommets u et v.

Annexe

Détail des variables globales

```
# Variables globales
gamma = edges = backward_edges = config = max_edge_length = fig = axes =
      components = components_inv = None

# Description des variables :
# gamma : la courbe elle-m me , une liste de N points dans       ,
#         chaque point est un tableau numpy 3D
#
# edges : dictionnaire repr sentant les ar tes de la courbe,
#         edges[u] = v repr sente   l  arte   uv
#
# backward_edges : dictionnaire repr sentant les ar tes invers es ,
#         backward_edges[u] = v repr sente   l  arte   vu
#
# config : dictionnaire de param tres de configuration     voir load_config()
#
# max_edge_length : longueur maximale  d une   ar te
#
# fig : figure matplotlib
# axes : axes matplotlib
#
# components : liste de composantes connexes disjointes
#
# components_inv : dictionnaire des composantes connect es
#         components_inv[i] = j si la composante i est reli e    j
```

Annexe

Paramétrage des courbes (1/2)

```
# Fichier de configuration par d faut pour le d nouage de n uds
# Options :
#   - geometry : 'file' ou 'torus'
#   - filename : chemin du fichier .obj
#   - frame folder : dossier de sauvegarde des images
#   - visuals : 'clean', 'arrows'
#   - figure setup : '2d 2x1', '2d 1x1', '3d'
#   - scattered : True ou False (affiche les points)
#   - flip : True ou False ( change  Y et Z : Blender utilise Y vers le haut)
#   - tolerance : float (tol rance sur  l  nergie  ou le d nouage)
#   - max iterations : int (nombre d'it rations)
#   - n : nombre de sommets si courbe g n r e
#   - p, q : param tres  d un  n ud torique
```

Annexe

Paramétrage des courbes (2/2)

```
# Suite des options :
#    - r1, r2 : rayons int rieur et ext rieur
#    - varying : True ou False (dt dynamique)
#    - clean plotting : True ou False
#    - angle : tuple (angle de vue pour affichage 3D)
#    - headless : True ou False (sans interface graphique)
#    - title : 'n', 'energy', 'dt', etc.
#    - name : nom de la configuration (affichage 3D)

default_config = {
    'geometry': 'file',
    'filename': './obj/reefknot2.obj',
    'frame folder': './frames/reefknot',
    'figure setup': '2d 2x1',
    'visuals': 'clean',
    'scattered': True,
    'flip': True,
    'tolerance': 0.1,
    'max iterations': 5_000,
}
```

Annexe

Fonction curve_distance (1/2)

```
# Trouve le plus court chemin entre u et v
def curve_distance(u ,v):
    global gamma, edges, backward_edges

    # Trouve le chemin entre u et v
    rpath = []
    lpath = []
    rlength = llength = 0
    rcur = lcur = u
    rnext = lnext = u
    # Parcours la courbe de la droite jusqu'   trouver v
    while rnext != v and lnext != v:
        # Avance    droite
        rnext = edges[rcur]
        rlength += np.linalg.norm(gamma[rnext] - gamma[rcur])
        rpath.append(rcur)
        rcur = rnext
```

Annexe

Fonction curve_distance (2/2)

```
        # Avance    gauche
        lnext = backward_edges[lcur]
        llength += np.linalg.norm(gamma[lnext] - gamma[lcur])
        lpath.append(lcur)
        lcur = lnext

    length = 0
    path = []

    if rnext == v:
        path = rpath
        length = rlength
    elif lnext == v:
        path = lpath
        length = llength

    path.append(v)

    return length, path
```

Annexe

Fonction `Mobius_gradient` (1/2)

```
# Calcule l' nergie  de M bius et le gradient pour une paire (u,v)
def Mobius_gradient(pair):
    global gamma, edges, backward_edges, config

    u, v = map(int, pair)

    energy = 0
    gradient = np.zeros((len(gamma), 3))

    # Terme de distance euclidienne A
    A = 1 / np.linalg.norm(gamma[u] - gamma[v]) ** 2
    distance = np.linalg.norm(gamma[u] - gamma[v])
    gradient[v] +=  2 * (gamma[u] - gamma[v]) / distance ** 4
    gradient[u] += -2 * (gamma[u] - gamma[v]) / distance ** 4

    if components_inv[u] != components_inv[v]:
        energy += A
        return energy, gradient, distance
```

Annexe

Fonction Mobius_gradient (2/2)

```
    # Terme de distance le long de la courbe B
    length, path = curve_distance(u, v)
    B = 1 / length ** 2
    energy += A - B

    for i in path[1:-1]:
        dist_I = np.linalg.norm(gamma[i] - gamma[backward_edges[i]])
        dist_I1 = np.linalg.norm(gamma[edges[i]] - gamma[i])
        der = (2 * (gamma[i] - gamma[backward_edges[i]]) / (length ** 3 * dist_I
            )
            - 2 * (gamma[edges[i]] - gamma[i]) / (length ** 3 * dist_I1))
        gradient[i] += der

    # Cas particuliers : extr mit s du chemin
    if path[0] == u:
        dist = np.linalg.norm(gamma[edges[u]] - gamma[u])
        gradient[u] += -2 * (gamma[edges[u]] - gamma[u]) / (length ** 3 * dist)
        dist = np.linalg.norm(gamma[v] - gamma[backward_edges[v]])
        gradient[v] += 2 * (gamma[v] - gamma[backward_edges[v]]) / (length ** 3
            * dist)
    else:
        dist = np.linalg.norm(gamma[edges[v]] - gamma[v])
        gradient[v] += -2 * (gamma[edges[v]] - gamma[v]) / (length ** 3 * dist)
        dist = np.linalg.norm(gamma[u] - gamma[backward_edges[u]])
        gradient[u] += 2 * (gamma[u] - gamma[backward_edges[u]]) / (length ** 3
            * dist)

    return energy, gradient, distance
```

Annexe

Fonction Mobius_gradient_descent_opti_fixe (1/3)

```
# Descente de gradient (version optimis e avec k-NN fixe + Monte Carlo)
def Mobius_gradient_descent_opti_fixe():
    global gamma, config, max_edge_length, edges, backward_edges, fig

    # 0) Initialisation
    clear_frames()
    compute_max_edge_length()

    # 1) Param tres
    max_iters   = config['max iterations']
    dt          = config['dt']
    tol         = config['tolerance']
    k_neighbors = config.get('k_neighbors', 2)
    m_samples   = config.get('monte_carlo_samples', 5)
    varying     = config.get('varying', False)
    visuals     = config.get('visuals', 'clean')
    frame_dir   = config['frame folder']

    N = len(gamma)
    txt = fig.texts[0]
```

Annexe

Fonction `Mobius_gradient_descent_opti_fixe` (2/3)

```
    # Pr -calcul de la longueur cumul e
    edge_vecs    = np.diff(np.vstack([gamma, gamma[0]]), axis=0)
    edge_len     = np.linalg.norm(edge_vecs, axis=1)
    cumlen       = np.concatenate([[0], np.cumsum(edge_len)])
    total_length = cumlen[-1]
    cumlen[-1]   = total_length

    t_acc = 0.0
    times, energies, dts = [], [], []

    for it in range(max_iters):
        # Reconstruction du KD-tree
        tree = cKDTree(gamma)

        # Initialisation
        gradient = np.zeros((N, 3))
        energy   = 0.0
        min_dist = np.inf
```

Annexe

Fonction Mobius_gradient_descent_opti_fixe (3/3)

```
        for u in range(N):
            # --- 1) k plus proches voisins locaux ---
            kq = min(k_neighbors + 1, N)
            _, idxs = tree.query(gamma[u], k=kq)
            locals_ = idxs[1:]  # on enl ve u lui-m me

            # --- 2) voisins lointains al atoires ---
            far_candidates = list(set(range(N)) - set(locals_) - {u})
            far = random.sample(far_candidates, min(m_samples, len(
                far_candidates)))

            # --- 3) calcul de l' nergie /gradient ---
            for v in np.concatenate([locals_, far]):
                e, g, d = Mobius_gradient((u, v))
                energy   += e
                gradient += g
                min_dist  = min(min_dist, d)

        # --- 4) Adaptation de dt si activ e ---
        if varying:
            gmax = np.max(np.abs(gradient))
            dt   = min_dist * tol / max(gmax, 1e-8)
        t_acc += dt

        # --- 5) Mise    jour de gamma ---
        gamma -= dt * gradient
```

Annexe

Fonction `Mobius_gradient_descent_opti_fixe` (fin)

```
        # --- 6) Visualisation ---
        if visuals == 'clean':
            clean_plot(it)
        elif visuals == 'arrows':
            plot_arrows(gradient, energy, it, min_dist)
        else:
            norms = np.linalg.norm(gradient, axis=1)
            txt.set_text(f"t={t_acc:.4f} dt={dt:.2e} E={energy:.5f} gmax={norms.
                max():.2f}")

        # --- 7) Sauvegarde ---
        if not config.get('headless', False) and config.get('visuals') != 'none'
            :
            plt.savefig(f"{frame_dir}/frame{it:05d}.png")

        # Archivage des mesures
        times.append(t_acc)
        energies.append(energy)
        dts.append(dt)
        print(f"Iter {it+1}/{max_iters}     E={energy:.5f}", end="\r")

    # Fin
    if not config.get("headless", False):
        plt.show()
    plot_energy(times, energies, dts)
```

Annexe

Fonction Mobius_gradient_descent_non_opti (1/3)

```
# Descente de gradient non optimis e (complexit  quadratique)
def Mobius_gradient_descent_non_opti():
    global gamma, config, max_edge_length, edges

    clear_frames()
    compute_max_edge_length()

    tolerance = config['tolerance']
    max_iters = config['max iterations']

    count = 0
    txt = fig.texts[0]

    uv_pairs = [(u, v) for u in range(len(gamma)) for v in range(len(gamma)) if
        u != v]

    t = 0
    true_time = []
    dts = []
    energies = []
    start = time.time()
    dt = config['dt']
```

Annexe

Fonction Mobius_gradient_descent_non_opti (2/3)

```
    while count < max_iters:
        # Calcul parall le des gradients/ nergies
        pool = mp.Pool(processes=mp.cpu_count())
        results = pool.map(Mobius_gradient, uv_pairs)

        # Ajout des termes d' nergie  et de gradient
        energy = 0
        gradient = np.zeros((len(gamma), 3))
        min_dist = np.inf
        for e, g, dist in results:
            energy += e
            gradient += g
            min_dist = min(min_dist, dist)

        # Calcul adaptatif de dt si activ
        if config['varying']:
            m = gradient.max()
            dt = min_dist * tolerance / m
        t += dt
```

Annexe

Fonction Mobius_gradient_descent_non_opti (3/3)

```
        # Visualisation selon l'option
        if config['visuals'] == 'clean':
            clean_plot(count)
        elif config['visuals'] == 'arrows':
            plot_arrows(gradient, energy, count, min_dist)
        else:
            fill_var = 30
            norms = np.linalg.norm(gradient, axis=1)
            txt.set_text(f'{"time: "}{t:.8f}'.ljust(fill_var)+
                         f'{"dt: "}{dt:.2e}'.ljust(fill_var)+
                         f'{"energy: "}{energy:.5f}'.ljust(fill_var)+
                         f'{"curve length: "}{curve_length():.3f}'.ljust(
                              fill_var)+
                         f'{"max grad: "}{norms.max():.2f}'.ljust(fill_var))
            draw_angles(norms, gradient)

        plt.savefig(f'{config["frame folder"]}/frame' + str(count).zfill(5) + '.
             png')
        count += 1

        # Mise    jour de la courbe
        gamma -= dt * gradient
        print(f'energy: {energy}', end='\r')
        true_time.append(time.time() - start)
        dts.append(dt)
        energies.append(energy)

    plt.show()
    plot_energy(true_time, energies, dts)
```

Annexe

Mobius_gradient_descent_opti_dynamique (1/5)

```
# Descente optimis e avec k adaptatif et Monte-Carlo
def Mobius_gradient_descent_opti_dynamique():
    global gamma, config, max_edge_length, edges, backward_edges, fig

    # 0) Initialisation
    clear_frames()
    compute_max_edge_length()

    # 1) Param tres
    max_iters     = config['max iterations']
    dt            = config['dt']
    tol           = config['tolerance']
    visuals       = config.get('visuals', 'clean')
    frame_dir     = config['frame folder']
    varying       = config.get('varying', False)
```

Annexe

Mobius_gradient_descent_opti_dynamique (2/5)

```
    # Param tres adaptatifs pour k
    k_min          = config.get('k_neighbors_start', 4)
    k_max          = config.get('k_neighbors_max', len(gamma)-1)
    k_step         = config.get('k_neighbors_step', 2)
    k_neighbors    = k_min
    e_thresh       = config.get('k_neighbors_energy_thresh', 1e-3)

    # Monte-Carlo
    m_samples      = config.get('monte_carlo_samples', 20)

    # Pr -calcul de la longueur cumul e
    N              = len(gamma)
    edge_vecs      = np.diff(np.vstack([gamma, gamma[0]]), axis=0)
    edge_len       = np.linalg.norm(edge_vecs, axis=1)
    cumlen         = np.concatenate([[0], np.cumsum(edge_len)])
    total_length   = cumlen[-1]
    cumlen[-1]     = total_length
```

Annexe

Mobius_gradient_descent_opti_dynamique (3/5)

```
    pool = mp.Pool(mp.cpu_count())

    t_acc      = 0.0
    txt        = fig.texts[0]
    prev_energy = None

    for it in range(max_iters):
        # Reconstruction du KD-tree
        tree = cKDTree(gamma)

        # Requ te k-NN
        kq, k_neighbors = min(k_neighbors+1, N), k_neighbors
        _, idx_knn = tree.query(gamma, k=k_neighbors+1)

        energy   = 0.0
        gradient = np.zeros((N,3))
        min_dist = np.inf
```

Annexe

Mobius_gradient_descent_opti_dynamique (4/5)

```
        for u in range(N):
            locals_ = idx_knn[u][1:]
            far_pool = list(set(range(N)) - set(locals_) - {u})
            far = random.sample(far_pool, min(m_samples, len(far_pool)))

            for v in np.concatenate([locals_, far]):
                e, g, d = Mobius_gradient((u,v))
                energy   += e
                gradient += g
                min_dist  = min(min_dist, d)

        if varying:
            gmax = np.max(np.abs(gradient))
            dt   = min_dist * tol / max(gmax,1e-8)
        t_acc += dt

        gamma -= dt * gradient

        if prev_energy is not None:
            if abs(prev_energy - energy) < e_thresh:
                k_neighbors = min(k_neighbors + k_step, k_max)
        prev_energy = energy
```

Annexe

Mobius_gradient_descent_opti_dynamique (5/5)

```
        if visuals=='clean':
            clean_plot(it)
        elif visuals=='arrows':
            plot_arrows(gradient, energy, it, min_dist)
        else:
            norms = np.linalg.norm(gradient,axis=1)
            txt.set_text(f"t={t_acc:.4f} dt={dt:.2e} E={energy:.5f} k={
                k_neighbors}")

        plt.savefig(f"{frame_dir}/frame{it:05d}.png")
        print(f"Iter {it+1}/{max_iters}     E={energy:.5f}  k={k_neighbors}",
              end="\r")

    pool.close()
    pool.join()
    if not config.get("headless", False):
        plt.show()
```

Annexe

Fonctions curve_length et compute_max_edge_length

```
# Calcule la longueur totale de la courbe
def curve_length():
    length = 0
    cur = 0
    next = -1
    while next != 0:
        next = edges[cur]
        length += np.linalg.norm(gamma[cur] - gamma[next])
        cur = next
    return length

# Calcule la plus grande longueur d'ar te de la courbe
def compute_max_edge_length():
    global max_edge_length, gamma, edges, backward_edges
    max_edge_length = 0
    for u, v in edges.items():
        max_edge_length = max(max_edge_length, np.linalg.norm(gamma[u] - gamma[v
            ]))
    ratio = config.get('max_edge_length_ratio', 1.0)
    max_edge_length = max_edge_length * ratio
    print("max length: ", max_edge_length, end='\r')
```

Annexe

Fonction `clear_frames`

```
# Supprime tous les fichiers .png et .pdf dans le dossier des frames
def clear_frames():
    folder = config['frame folder']
    import os
    for filename in os.listdir(folder):
        if filename.endswith(".pdf") or filename.endswith(".png"):
            os.remove(os.path.join(folder, filename))
```

Annexe

Fonction `clean_plot` (1/2)

```
# Affiche la courbe proprement dans les subplots configur s
def clean_plot(iters):
    global fig, axes, gamma, config, components
    X, Y, Z = gamma[:,0], gamma[:,1], gamma[:,2]
    X, Y, Z = np.array([*X,X[0]]), np.array([*Y,Y[0]]), np.array([*Z,Z[0]])

    for ax in axes.flatten():
        ax.cla()
        if config['figure setup'] == '3d 1x1':
            ax.axis('off')
```

Annexe

Fonction `clean_plot` (2/2)

```
        colors = ['black', 'green', 'blue', 'red', 'purple', 'yellow', 'pink', '
             brown', 'gray', 'orange']
        if ax.name == '3d':
            ax.set_xticks([])
            ax.set_yticks([])
            ax.set_zticks([])

            if len(components) > 1:
                for component in components:
                    color = colors.pop(0)
                    ax.plot([*X[component], X[component][0]], [*Y[component], Y[
                         component][0]], [*Z[component], Z[component][0]], color
                         ='black', linewidth=1)
                    if 'scattered' in config and config['scattered']:
                        ax.scatter(X[component], Y[component], Z[component],
                             color=color, s=20, alpha=1.0)
            else:
                ax.plot(X, Y, Z, color='black', linewidth=1)
                if 'scattered' in config and config['scattered']:
                    ax.scatter(X, Y, Z, cmap='twilight_shifted', c=range(len(X))
                         , s=10)
            if 'zlims' in config:
                ax.set_zlim(config['zlims'])
```

Annexe

Fonction `draw_angles` (1/2)

```
# Affiche les vecteurs gradients dans diff rentes vues (3D et 2D)
def draw_angles(norms=None, gradients=np.zeros(0)):
    global gamma, max_edge_length

    X, Y, Z = gamma[:,0], gamma[:,1], gamma[:,2]
    X, Y, Z = np.array([*X, X[0]]), np.array([*Y, Y[0]]), np.array([*Z, Z[0]])
    gx, gy, gz = gradients[:,0], gradients[:,1], gradients[:,2]
    gx, gy, gz = np.array([*gx, gx[0]]), np.array([*gy, gy[0]]), np.array([*gz,
        gz[0]])
    g = [(gx, gy, gz)] * 4
    data = [(X,Y,Z)] * 4
    titles = ['Normal view', 'Top view', 'Side view', 'Front view']
```

Annexe

Fonction `draw_angles` (2/2)

```
    for ax in axes.flatten():
        ax.cla()
        dat = data.pop(0)
        grad = g.pop(0)
        if len(dat) == 3:
            if 'zlims' in config:
                ax.set_zlim(config['zlims'])
            ax.scatter3D(*dat, c = range(len(X)))
            ax.quiver(*dat, *grad, length=max_edge_length/2, color='black',
                  arrow_length_ratio=0.1, normalize=True)
            ax.plot(*dat, c = 'black')
        elif len(dat) == 2:
            X, Y = dat
            grad = np.array(grad).T
            norms = np.linalg.norm(grad, axis=1).reshape(-1)
            ax.scatter(*dat, c = norms)
            grad = grad / np.stack([norms,norms],axis=1)
            grad *= max_edge_length/2
            grad = list(grad.T)
            ax.quiver(*dat, *grad, color='black', width=0.005, headwidth=2,
                  headlength=2, headaxislength=1.5)
            cmap = cm.get_cmap('rainbow')
            c = cmap(norms)
            ax.plot(*dat, c = 'black')

        ax.set_title(titles.pop(0))

    if not config['headless']:
        plt.draw()
```

Annexe

Fonction `plot_arrows` (1/2)

```
# Affiche les fl ches de gradient sur la courbe
def plot_arrows(gradients, energy, count, min_dist):
    global gamma, edges, max_edge_length, fig, axes

    X, Y, Z = gamma[:,0], gamma[:,1], gamma[:,2]
    gx, gy, gz = gradients[:,0], gradients[:,1], gradients[:,2]

    plt.rcParams["axes.titley"] = 0.95

    for ax in axes.flatten():
        ax.clear()
        ax.set_title(r'\(\mathcal{{E}}={energy:.2f}\)'.format(energy=energy))
        ax.grid(False)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_zticks([])
        ax.axis('off')
```

Annexe

Fonction `plot_arrows` (2/2)

```
        if len(components) > 1:
            for component in components:
                ax.plot([*X[component], X[component][0]],
                        [*Y[component], Y[component][0]],
                        [*Z[component], Z[component][0]],
                        color='black')
        else:
            ax.plot([*X, X[0]], [*Y, Y[0]], [*Z, Z[0]], color='black')

        ax.scatter(X, Y, Z, color='black', s=8)
        ax.quiver(X, Y, Z, gx, gy, gz, length=min_dist,
                  color='blue', normalize=True)

    if not config['headless']:
        plt.draw()
        plt.pause(0.005)

    folder = config['frame folder']
    plt.savefig(f'{folder}/frame{count:05d}.pdf')
    plt.savefig(f'{folder}/frame{count:05d}.png')
```

Annexe

load_config (1/7)

```
# Charge la configuration et initialise tous les param tres
def load_config(config_):
    global config, gamma, edges, backward_edges, fig, axes, components,
        components_inv
    config = config_

    plt.rcParams.update({
        "text.usetex": False,
        "font.family": "",
        "font.size": 20
    })
```

Annexe

load_config (2/7)

```
    # Valeurs par d faut
    if 'headless' not in config:
        config['headless'] = False
    if 'dt' not in config:
        config['dt'] = 0.01
    if 'frame folder' not in config:
        config['frame folder'] = './frames'
    if 'tolerance' not in config:
        config['tolerance'] = 1e-2
    if 'varying' not in config:
        config['varying'] = True
    if 'figure setup' not in config:
        config['figure setup'] = '3d'
    if 'clean plotting' not in config:
        config['clean plotting'] = False
        config['ignore edges'] = False
    if 'max iterations' not in config:
        config['max iterations'] = 99_999
    if 'title' not in config:
        config['title'] = ''
```

Annexe

load_config (3/7)

```
    # Chargement de la g om trie
    geom = ''
    if 'geometry' not in config:
        geom = 'circle'
        gamma, edges, backward_edges = torus_knot(1, 0, config['n'], 3, 0)
    elif config['geometry'] == 'torus':
        p = config['p']
        q = config['q']
        n = config['n']
        r1 = config['r1']
        r2 = config['r2']
        geom = 'torus knot' + f' p={p}, q={q}, n={n}, r1={r1}, r2={r2}'
        gamma, edges, backward_edges = torus_knot(p, q, n, r1, r2)
    elif config['geometry'] == 'file':
        filename = config['filename']
        gamma, edges, backward_edges, components = read_obj(filename)
        geom = filename.split('/')[-1].split('.')[0]
```

Annexe

load_config (4/7)

```
    elif config['geometry'] == 'equilateral triangle':
        geom = 'equilateral triangle'
        gamma, edges, backward_edges = simple_equilateral_triangle()
    elif config['geometry'] == 'circle':
        geom = 'circle'
        gamma, edges, backward_edges = torus_knot(1, 0, config['n'], 3, 0)

    if components is None:
        components = [np.arange(len(gamma))]

    components_inv = {v:k for k, component in enumerate(components) for v in
         component}

    if 'name' not in config:
        config['name'] = geom

    if 'ignore edges' in config and config['ignore edges']:
        gamma, edges, backward_edges = create_edges(gamma)
```

Annexe

load_config (5/7)

```
    # V rifie si la boucle est ferm e
    edges = check_loop(edges)

    # Transformations
    if 'flip' in config and config['flip']:
        gamma = invert_YZ(gamma)  # remet  l axe  Z en haut
    if 'flatten' in config and config['flatten']:
        gamma = set_Z(gamma, 0)   # projette la courbe sur Z=0
    if 'decimate' in config and config['decimate'] < 1:
        gamma, edges = decimate(gamma, edges, config['decimate'])
        backward_edges = {v:k for k,v in edges.items()}
```

Annexe

load_config (6/7)

```
    # Configuration des figures
    if config['figure setup'] == '3d':
        angle = (30,30)
        fig = plt.figure(figsize=(10, 10))
        ax1 = fig.add_subplot(221, projection='3d', xlabel='x', ylabel='y',
              zlabel='z')
        if 'angle' in config:
            angle = config['angle']
        ax1.view_init(*angle)
        ax2 = fig.add_subplot(222, projection='3d', xlabel='x', ylabel='y')
        ax2.view_init(90, 0)
        ax3 = fig.add_subplot(223, projection='3d', xlabel='x', ylabel='z')
        ax3.view_init(0, 90)
        ax4 = fig.add_subplot(224, projection='3d', xlabel='x', ylabel='y',
              zlabel='z')
        ax4.view_init(0, 0)
        axes = np.array([ax1, ax2, ax3, ax4])
        txt = fig.text(0.07, 0.97, '', fontsize=10, color='black')
```

Annexe

load_config (7/7)

```
    # Autres cas possibles : 2d 2x1, 2d 1x1, 3d 1x1
    elif config['figure setup'] == '2d 2x1':
        angle = (35, 45, 0)
        if 'angle' in config:
            angle = config['angle']
        fig = plt.figure(figsize=(10, 5))
        ax1 = fig.add_subplot(121, projection='3d', xlabel='x', ylabel='y',
              zlabel='z')
        ax1.view_init(*angle)
        ax2 = fig.add_subplot(122, xlabel='x', ylabel='y')
        axes = np.array([ax1, ax2])
        txt = fig.text(0.07, 0.97, '', fontsize=10, color='black')

    # (idem pour 2d 1x1 et 3d 1x1, omis pour concision)

    if 'max_edge_length_ratio' not in config:
        config['max_edge_length_ratio'] = 1

    print(f'Loaded {geom}     with {len(gamma)} vertices')

    if 'visuals' not in config:
        config['visuals'] = 'clean'

    if config['visuals'] == 'gradient computation':
        plot_gradient_computation()
        exit(0)
```

Annexe

Fonction `read_obj` (1/3)

```
# Lit un fichier .obj et retourne :
# - tableau des sommets (x, y, z)
# - dictionnaire des ar tes {u:v}
# - dictionnaire des ar tes invers es {v:u}
# Remarque : les indices commencent    0 et non 1 comme dans le fichier .obj

def read_obj(filename):
    with open(filename) as f:
        lines = f.readlines()
        vertices = [line.split()[1:] for line in lines if line.startswith('v ')]
        vertices = [[float(x) for x in vertex] for vertex in vertices]
        vertices = np.array(vertices)
        edges = [line.split()[1:] for line in lines if line.startswith('l ')]
        new_edges = {int(edge[0])-1:int(edge[1])-1 for edge in edges}
        backward_edges = {v:k for k,v in new_edges.items()}
```

Annexe

Fonction `read_obj` (2/3)

```
        # V rifie les doublons dans les ar tes
        if len(new_edges.keys()) < len(edges) or len(backward_edges.keys()) <
             len(edges):
            new_edges = fix_obj(edges)

    # Trie les sommets selon les ar tes
    components = []
    component = []
    new_vertices = np.zeros((len(vertices), 3))
    cur = list(new_edges.keys())[0]
    visited = set()
    for i in range(len(vertices)):
        new_vertices[i] = vertices[cur]
        component.append(i)
        visited.add(cur)
        if cur in new_edges:
            next = new_edges[cur]
```

Annexe

Fonction `read_obj` (3/3)

```
        # Si next a d j     t   visit , nouvelle composante
        if next in visited:
            components.append(component.copy())
            component = []
            new_edges = {k:v for k,v in new_edges.items() if k not in visited}
            if len(new_edges) > 0:
                cur = list(new_edges.keys())[0]
        else:
            cur = next

    shifted_components = [component[1:] + component[:1] for component in
         components]
    new_edges = {}
    for component, shifted_component in zip(components, shifted_components):
        for u,v in zip(component, shifted_component):
            new_edges[u] = v

    backward_edges = {v:k for k,v in new_edges.items()}

    return new_vertices, new_edges, backward_edges, components
```

Annexe

Fonction `create_edges` (1/2)

```
# Cree les aretes a partir de gamma (liste de sommets)
# Retourne : dictionnaire {u:v} des aretes ou u est connecte a v

def create_edges(gamma):
    edges = {}

    # Choisit le sommet le plus isole comme point de depart
    start = -1
    max_sum_dist = 0
    for u in range(len(gamma)):
        sum_dist = 0
        vertex = gamma[u]
        for v in gamma:
            sum_dist += np.linalg.norm(vertex - v)
        if sum_dist > max_sum_dist:
            max_sum_dist = sum_dist
            start = u
```

Annexe

Fonction create_edges (2/2)

```
    # Creation des aretes
    u = start
    neighbor = None
    while neighbor != start:
        min_dist = np.inf
        neighbor = None
        for v in range(len(gamma)):
            dist = np.linalg.norm(gamma[u] - gamma[v])
            if u != v and dist < min_dist and v not in edges.values():
                min_dist = dist
                neighbor = v
        edges[u] = neighbor
        u = edges[u]

    new_vertices = np.zeros((len(vertices), 3))
    cur = 0
    next = edges[cur]
    for i in range(len(vertices)):
        new_vertices[i] = vertices[cur]
        cur = next
        next = edges[cur]

    edges = {i:(i+1)%len(vertices) for i in range(len(vertices))}

    return vertices, edges, {v:k for k,v in edges.items()}
```

Annexe

Fonction `fix_obj`

```
# Tente de corriger les doublons d'aretes dans un fichier .obj
# En inversant certaines aretes si besoin

def fix_obj(edges):
    new_edges = {}
    dup = []
    for u,v in edges:
        u = int(u) - 1
        v = int(v) - 1
        if u not in new_edges:
            new_edges[u] = v
        elif v not in new_edges:
            new_edges[v] = u
            dup.append(u+1)
        else :
            dup.append(u+1)
            dup.append(v+1)
            raise Exception(f'Found duplicate vertices: {set(dup)} ...')
```

Annexe

Fonction `torus_knot`

```
# Cree un noeud torique
# - r1 : rayon du grand cercle (centre du tore)
# - r2 : rayon du petit cercle
# - p : nombre de tours autour de r1
# - q : nombre de tours autour de r2

def torus_knot(p=2, q=3, n=200, r1 = 4, r2 = 2):
    phi = np.linspace(0, 2 * np.pi, n+1)
    phi = phi[:-1]
    r = r2 * np.cos(q*phi) + r1
    X = r * np.cos(p*phi)
    Y = r * np.sin(p*phi)
    Z = -r2* np.sin(q*phi)

    edges = {i:(i+1)%n for i in range(n)}

    return np.array([X, Y, Z]).T, edges, {v:k for k,v in edges.items()}
```

Annexe

Fonction `check_loop`

```
# Ferme une boucle en connectant les points de discontinuite
# - avance jusqu'a une discontinuite
# - recule jusqu'a une discontinuite
# - connecte les deux

def check_loop(edges):
    backward_edges = {v:k for k,v in edges.items()}

    first = list(edges.keys())[0]
    v = edges[first]
    while v in edges.keys() and v != first:
        v = edges[v]
    if v != first:
        w = first
        while w in backward_edges.keys():
            w = backward_edges[w]
        edges[v] = w
        print('Closed loop')
    else:
        print('Already closed loop')
    return edges
```

Annexe

Fonction `write_obj`

```
# Ecrit les sommets dans un fichier .obj au format standard
def write_obj(filename, vertices):
    with open(filename, 'w') as f:
        X, Y, Z = vertices
        for x,y,z in zip(X, Y, Z):
            f.write('v {} {} {}\n'.format(x,y,z))
        for i in range(1, len(X)):
            f.write('l {} {}\n'.format(i, (i+1)))
        f.write('l {} {}\n'.format(len(X), 1))
```

Annexe

Fonctions `invert_YZ` et `set_Z`

```
# Inverse les coordonnees Y et Z
def invert_YZ(vertices):
    return np.array([[x, z, y] for x,y,z in vertices])

# Remplace toutes les coordonnees Z par une constante
def set_Z(vertices:np.array, Z:float):
    return np.array([[x, y, Z] for x,y,_ in vertices])
```

Annexe

Fonction decimate

```
# Reduit le nombre de sommets par echantillonnage aleatoire
# - percentage = 0.5 garde un sommet sur deux
# - cree les nouvelles aretes entre les sommets conserves

def decimate(vertices:np.array, edges:dict, percentage:float=0.5):
    new_size = int(len(vertices)*percentage)
    old_indices = np.random.choice(len(vertices), size=new_size, replace=False)
    old_indices.sort()
    new_indices = {old_index:new_index for new_index, old_index in enumerate(
        old_indices)}
    new_vertices = vertices[old_indices]

    new_edges = {}
    for new_index in range(new_size):
        old_index = old_indices[new_index]
        old_neighbor = edges[old_index]
        new_neighbor = old_neighbor
        while new_neighbor not in old_indices:
            new_neighbor = edges[new_neighbor]
        new_neighbor = new_indices[new_neighbor]
        new_edges[new_index] = new_neighbor

    return new_vertices, new_edges
```