PRÉPA DES INP

Couture de cycles pour la fabrication par dépôt de filaments fondus

Projet informatique S4

Equipe :
Hugo CAUZID
Teresa FERNANDEZ
Tristan LOURDE ROCHEBLAVE
Yan RABEFANIRAKA
Samuel SCHOIRFER

Rapport de :
Yan RABEFANIRAKA
Référent :
Xavier CHERMAIN

1er novembre 2024

Résumé

Le but de notre projet est, à partir d'un fichier *svg* contenant la coupe plane d'une impression 3D, de relier tous les différents cycles disjoints qui composent la forme. Cela peut être utile pour la réalisation d'impression 3D sous forme de tranches ou de "gravures". Mais, ce programme permet notamment d'éviter les dépôts superflux lors des va-et-vient de la buse de l'imprimante, résultant souvent en des imperfections voire des imprécisions.

Table des matières

Chapitre 1

Remerciements

Ce projet ayant été effectué dans le cadre de notre dernier semestre au sein de la Prépa des INP de Nancy, je tiens à remercier notre référent de projet et professeur de module informatique, M. Xavier Chermain, qui s'est rendu disponible pour nous encadrer et nous conseiller tout au long du projet, mais qui a aussi, de par l'enseignement de son module, beaucoup contribué à la finalisation de ce projet.
Je tiens également à remercier l'équipe MFX de l'Inria Nancy, dont est membre notre référent, pour leur travaux dont nous avons pu largement nous inspirer pour construire notre projet.
Aussi, je remercie notre camarade de promotion Tanguy Mathian, pour le précieux recul qu'il nous a offert concernant quelques aspects clés du programme, en échangeant avec nous régulièrement sur le projet, et avec qui nous avons pu comparer nos approches et résultats.
Enfin, je remercie en particulier mes camarades de groupe qui ont tous travaillé d'arrache-pied pour réaliser ce projet, mais qui ont aussi su détendre l'atmosphère quand il le fallait.

Chapitre 2

Introduction

Dans le cadre de notre 2ème année de scolarité au sein de la Prépa des INP de Nancy, un projet de groupe dans le domaine qui nous intéresse nous est imposé. Notre groupe, dont tous les membres ont suivi la Majeure Informatique proposée au 3ème semestre de la formation, a reçu pour consigne de trouver et appliquer en Python un algorithme inspiré par les travaux de l'équipe MFX à l'Inria sur l'impression 3D, dont est membre notre tuteur de projet M. Xavier Chermain.

Vous connaissez certainement le fameux "Problème du Dessin de l'Enveloppe", dans lequel au lieu de dessiner une enveloppe en plusieurs étapes, on s'arrange pour le faire en un seul trait, sans jamais lever le crayon de la feuille, ni passer plus d'une fois par le même point. Cela doit aussi vous évoquer, à plus grande échelle, le "Problème du Commis Voyageur" (*Traveler Salesman Problem* ou *TSP* en Anglais), où l'objectif de résolution est de trouver, à partir d'un ensemble de villes, le chemin le plus court pour qu'un voyageur de commerce passe par toutes ces villes, sans revenir plus d'une fois au même endroit. Maintenant, pour aller encore plus loin, imaginons un ensemble de points, groupés en cycles qui ne se croisent pas afin de former, en tout, un dessin. Si nous souhaitons imprimer en 3D ce dessin, l'imprimante aurait donc besoin de former tour à tour chacun de ces cycles, et donc de lever sa buse du lit d'impression plusieurs fois, et éventuellement de laisser des déchets qui pendent de la buse sur son chemin après chaque itération. Ainsi, en appliquant une résolution similaire au *TSP* à notre problème d'impression 3D mais en imposant en plus que les chemins ne s'intersectent jamais, on pourrait permettre à l'imprimante de ne dessiner qu'un seul cycle, qui serait alors l'assemblage de tous les cycles en entrée, qu'on aurait "cousus" ensemble un à un. Ce faisant, on réduit le risque de déchets et fioritures lors de l'impression. Cependant, l'équipe MFX va encore plus loin, puisqu'elle utilise cette technique de couture de cycles, pour réaliser des impressions anisotropes [1] ressemblant à des gravures, à l'aide de l'orientation de la buse de l'imprimante.

Notre objectif lors de ce projet n'est pas d'aller aussi loin, mais de s'inspirer de leurs travaux sur les impressions anisotropes afin de réaliser notre version, en Python puis en *NumPy*, de l'algorithme de couture de cycles. Ainsi, dans ce rapport, nous allons d'abord analyser theéoriquement le problème (au Chapitre 3). Puis, nous parlerons de sa mise en place, et notamment de la répartition des tâches (au Chapitre 4) ainsi que des difficultés rencontrées. Ensuite, nous reviendrons sur les

1. se dit d'un objet dont les propriétés dépendent d'une direction (ici, des propriétés visuelles dépendantes de l'orientation de la buse lors du dépôt de matières)

résultats obtenus et les observations qu'on peut en faire (au Chapitre 5). Enfin, nous concluerons en résumant les points qui peuvent être améliorés.

Chapitre 3

Analyse théorique et mise en pratique

3.1 Diagramme du système : entrée et sortie du programme

Cette section est entièrement illustrée par le diagramme en Figure 6.1.

3.2 Traveler Salesman Problem ou *TSP*

Notre problème partageant des similarités au *TSP*, il est de mise de brièvement présenter ce dernier.
Comme dit précédemment, c'est un problème du plus court chemin, où l'objectif est d'aller le plus rapidement possible d'un point à un autre, en ne passant qu'une fois par endroit. *A priori*, sans représentation pertinente de son ensemble d'entrées, le résolveur ne peut pas avancer. Un graphe non orienté [1] pondéré [2] et complet (vu en Mineure Informatique) est un parfait modèle de la situation : on peut substituer les villes avec les nœuds, les temps de trajet reliant un endroit à un autre par les arêtes pondérées. De là, trouver le chemin le plus court entre deux points du graphe se fait habituellement avec l'algorithme de Dijkstra [3], ce qui nécessite donc de parcourir le graphe (en largeur) afin d'établir pour chaque nœud les routes de poids minimal.

3.2.1 Application aux cycles et gestion des entrées-sorties *svg*

Néanmoins, dans notre représentation qui dépend du format *svg*, le graphe doit être orienté pour rester fidèle à l'orientation des cycles et l'ordre des points. Pour ce faire, notre graphe est constitué de trois listes : une contenant la totalité des points complexes au format $|real + j * imag|$, une liste d'adjacence contenant une liste par point avec l'indice du point suivant et du point précédent sous la forme $[suivant, precedent]$ (justifiant alors l'orientation), et d'une dernière contenant une liste par cycle avec l'indice du point de départ et le nombre de points du cycle ($[departCycle, tailleCycle]$). En effet, là où notre représentation diffère, c'est dans l'entrée du problème ; notre graphe n'est pas connexe car séparé en différents cycles orientés. De même, notre sortie diffère en ce qu'il ne cherche pas le plus court chemin (puisqu'il n'en n'existe pas pour tous les couples de points du graphe),

1. graphe dont le sens de navigation entre deux noeuds consécutifs n'est pas précisé.
2. graphe dont les arêtes sont dotées de poids entre deux noeuds.
3. `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`

mais les meilleures liaisons entre deux arêtes de cycles différents. L'une des tâches à faire a donc été d'appliquer cette nouvelle représentation à nos données d'entrée, dans un premier programme (*svg handler*) où on récupère les cycles du *svg* pour les convertir en notre modèle de graphe, utilisable pour toute la suite de l'exécution.

3.3 Traitement des entrées : couture

Ensuite, l'assemblage de 2 cycles nécessite deux étapes : la recherche du cycle le plus proche et la fusion de ces 2 cycles dans la représentation qu'on utilise.
Pour déterminer le cycle le plus proche, nous avons utilisé la formule de l'énergie (c.f Formule 6.2) donnée dans les documents fournis [1]. Cette formule permet d'évaluer la pertinence d'une liaison entre deux segments, et ainsi de trouver le cycle le plus proche ainsi que les côtés des cycles à relier (c.f Figure 6.6). Plus tard, après avoir échangé avec nos camarades de l'autre groupe, nous avons pu utiliser cette formule afin d'obtenir directement le sens de couture des 2 cycles et économiser du temps de calcul. Ensuite nous devons changer la liste d'adjacence des points afin de relier les points des 2 cycles les plus proches pour n'en former plus qu'un seul. Enfin, nous enlevons un des cycles fusionné dans la liste de cycles pour n'en avoir plus qu'un seul à la fin de l'opération.
La généralisation de ce procédé à l'entièreté du graphe se fait simplement par une boucle *for*, qui va explorer un à un les cycles du graphe en commençant par celui de plus petite taille, pour ensuite envoyer en sortie un unique cycle cousu, qui pourra être traité par le gestionnaire d'entrée-sortie.
Au total, cette partie de l'algorithme est supposée atteindre une complexité théorique de $O(n^4)$, avec quatre boucles *for* imbriquées : deux pour les énergies, une pour la couture, et la principale pour le graphe complet. Avec le format *NumPy* de tableaux de dimension $(N, 2)$, on réduit d'un degré la complexité pour atteindre $O(n^3)$.

Chapitre 4

Répartition des tâches et implication personnelle

Pour réaliser ce projet et faciliter la gestion des différents fichiers, nous avons décidé de créer un répertoire *Github*. Bien que difficile à prendre en main au début, cela nous a permis d'organiser et de garder à jour les différents programmes. Nous avons aussi choisi de créer un serveur *Discord* afin de communiquer nos résultats et d'organiser nos réunions de projet. Une autre chose que nous avons essayé de mettre en place est une nomenclature dans notre code à partir du *PEP8*[3], afin de faciliter la compréhension du code par les différents membres du projet (ce qui se révèlera insuffisant au long terme, quand certaines fonctions seront très denses). Concernant la répartition des tâches, nous avons initialement divisé le travail en 2 parties : la partie '*svg handler*' (c.f Algorithme 1) qui permet de traiter les données en entrée, et la partie '*stitch base*' (c.f Algorithme 2) qui permet de lier les différents cycles en un seul. Nous nous sommes donc divisés en 2 groupes basés sur nos différents modules, le groupe d'Hugo, Teresa et moi-même qui s'est occupé du *'stitch base'* et le groupe de Tristan et Samuel qui a fait le *'svg handler'*. Nous avons vite constaté que cette répartition initiale des tâches était inégale : une fois '*svg handler*' fini, le groupe de Tristan et Samuel s'est concentré sur l'implémentation de '*stitch base*' pour la version NumPy, aux côtés de Teresa.

4.1 Gestion de répertoire

Pour ma part, j'ai brièvement cherché, avant la première réunion, comment créer et gérer un répertoire *Github* à plusieurs utiliseurs, et ce directement à partir de l'environnement de travail *Visual Studio Code* [1]. Ayant trouvé une extension (c.f Figure 6.4) pour se connecter au répertoire, je me suis alors chargé de l'installation et de la mise en place de ce dernier sur la plupart des machines. Malheuresement, ne m'étant pas assez renseigné et ayant une vision encore trop superficielle de l'outil à cause d'usages trop ponctuels et pas assez réguliers, je n'ai pas non plus correctement renseigné le reste de l'équipe sur la manière de gérer le répertoire. Ce manque de renseignement nous a au départ causé un bon nombre de problèmes de gestion de version, entre les conflits de modifications et les problèmes de synchronisation. J'ai essayé de faire de mon mieux pour corriger les problèmes individuels quand les occasions se sont présentées, mais certaines de nos séances consistaient plus souvent en de la gestion de répertoires locaux que d'avancée dans le projet. Bref,

1. `https://en.wikipedia.org/wiki/Visual_Studio`

cela nous a apporté l'opposé de ce pourquoi git est intéressant, du moins au début, puisque nous avons peu à peu compris comment utiliser l'outil.

4.2 Gestion entrée-sortie

Samuel et Tristan s'étant chargé de l'entrée, je me suis proposé, pour alléger leur charge de travail, de gérer la sortie qui était elle beaucoup moins demandante, et ce grâce au module *svg-pathtools* [2]. J'ai donc créé la fonction qui récupère le chemin final offert par l'algorithme pour le convertir en svg. Au départ, en créant la fonction, je ne suis pas parti avec le même type d'entrée que notre programme final (c'est-à-dire un ensemble de points au lieu d'un chemin), ce qui fait que, même si à vue d'oeil le résultat semble bon, le *svg* en sortie n'était en fait qu'un amas indépendant de chemins, et non pas un seul cycle. C'est en inspectant les sorties que je m 'en suis rendu compte et que j'ai corrigé le problème. Enfin, ce n'est qu'un petit détail, mais je me suis amusé à prendre en compte dans cette fonction le nom du fichier d'entrée ainsi que le nombre de fichiers dans le dossier de sortie, et ce afin de nommer adéquatement le résultat (c.f Figure 6.5). Cela permet ainsi de faciliter la gestion et la vérification des résultats de test.

4.3 Intégration des formules mathématiques

Pour l'intégration du calcul de l'énergie (c.f Formule 6.2), notre première itération a été réalisée par Teresa et moi-même, mais nous avions seulement traduit en Python ce qui était donné dans [1]. Ensuite, je me suis essayé à intégrer la vérification de l'intersection entre deux segments à l'aide de la formule de Cramer([4] combiné aux équations trouvables ici [5]). Cependant, je n'ai pas pris en compte les différentes exceptions comme les équations de segment constantes, résultant très vite en des problèmes d'exécution. Hugo a ensuite réecrit la fonction afin de prendre en compte les cas limites et ainsi réaliser une fonction pertinente (c.f Formule 6.3). Notre version finale, où on utilise un résultat intermédiaire du calcul de l'énergie au lieu de la fonction de vérification d'intersection, a été principalement faite par Hugo avec moi qui lui sert de "canard" [3].

4.4 En somme

Dans l'ensemble, Hugo a fait une part plutôt importante du travail dans le groupe Teresa-Yan-Hugo, en partie à raison de la répartition de départ qui a été trop inégale avec trop peu de codage à 4-6 mains pour notre trio, ce qui a alors fait que les différentes nomenclatures rendaient la compréhension du code pour tous difficile dans son entièreté.
Pour autant, nous avons essayé au maximum de l'assister avec Teresa lors des sessions de code à 4-6 mains, parfois seulement en tant que "canard", mais la plupart du temps en tant que correcteurs ou deuxième cerveau.
Une autre raison a aussi été le fait que j'ai été en charge de corriger tous les problèmes liés à *Github* et aux répertoires locaux ; ce problème nous a pris une quantité absurde de temps au début du projet,

2. `https://pypi.org/project/svgpathtools/`

3. La méthode du canard en plastique consiste à expliquer méticuleusement le code source que l'on a écrit à un collègue, à un simple passant, ou même à un objet inanimé comme un canard en plastique (*Wikipedia*).

et m'a à titre personnel souvent fait dévier de ma part initiale, en me faisant passer plus de temps sur les machines de mes camarades que la mienne. Pour autant, avec l'avancée du projet et le gain d'expérience de chacun, j'ai de moins en moins eu besoin de corriger les problèmes de répertoire et ai pu me concentrer sur le code à 4-6 mains avec Hugo et Teresa.

Chapitre 5

Résultats

Les fichiers en sortie du programme sont au format *svg* (*Scalable Vector Graphics*). Ces fichiers sont des regroupements de chemins (eux-mêmes des regroupement de lignes) en plusieurs "cycles" pour former un dessin(c.f Figure 6.7a). En particulier, l'algorithme transforme les multiples cycles du dessin en un seul, en les reliant à des endroits clés, i.e. là où les cycles voisins sont à distance minimale (c.f Figure 6.7b). Ainsi, évaluer la qualité du résultat est assez élémentaire sachant que ce dernier n'a qu'à vérifier les critères suivants :

— Ne contenir qu'un seul cycle.
— Passer par tous les points.
— Ne pas avoir de lignes qui s'intersectent ou se superposent.

Ce format de fichier étant ouvrable avec des logiciels supportant l'édition d'images vectorielles (par exemple, *Inkscape*), on pourra donc directement y vérifier l'intégrité du résultat.

Le processus de validation est alors assez simple : En ouvrant le résultat dans le logiciel *Inkscape*, on clique sur un endroit quelconque du dessin ; normalement, cela met en valeur le cycle complet auquel appartient ce point. Si tous les chemins sont mis en surbrillance dans le panneau de sélection des chemins, alors l'objet contient un unique cycle. Pour être sûr que le cycle reconstitué passe bien par tous les points, nous affichons la liste contenant tous les cycles au fur et à mesure de l'avancée du programme. De cette manière, nous pouvons voir si le nombre de points total reste le même après une couture de 2 cycles, nous assurant que le cycle final passe effectivement par tous les points. Enfin, la vérification des intersections se fait manuellement, en comparant le résultat avec le fichier en entrée. Cependant, les entrées et sorties attendues étant fournies, nous nous sommes aussi permis de comparer nos coutures avec ce qui nous a été envoyé. En ce qui concerne lesdites coutures, les différentes itérations du programme en Python natif semblent toutes faire un travail satisfaisant (c.f Figure 6.7). De même, la version NumPy offre de bonnes coutures qui valident les conditions de vérification susmentionnées (les résultats sont peu ou prou les mêmes).

Vis-à-vis du temps d'exécution, nous utilisons le module intégré *time*. Nous regroupons les différents temps d'exécution dans un fichier texte au format *'nom de fichier' : hh :mm :ss.mmm*, ce qui nous permet de comparer l'efficacité des diverses approches que nous avons implémentées (c.f Figure 6.8). Mais, pour avoir plus de visibilité lors de l'exécution du programme, nous avons implémenté le module *tqdm* [1] qui affiche une barre de progression estimée à partir du nombre d'ité-

1. `https://pypi.org/project/tqdm/`

rations à faire, tout en offrant une approximation du temps d'exécution restant. N'ayant que peu d'*overhead* (c'est-à-dire de coûts d'exécution supplémentaires) avec seulement 60 nanosecondes par boucle, cette barre de progression ne fausse pas le temps d'exécution final même pour un très grand nombre de points en entrée.

Ce que nous pouvons remarquer dans les temps d'exécution en Python natif, c'est que pour des fichiers allant jusqu'à 111ko (*'bunny small cycles'*), voire 408 ko (*'square cycles'*) , le programme se termine en un temps inférieur à une heure (peut-être supérieur dépendant de la version utilisée)(c.f Figure 6.8) . C'est à partir de *'dragon cycles'* avec ses 4567 ko (c'est-à-dire environ 11 fois plus grand que son prédécesseur), que le programme flanche : à ce jour, nous ne savons toujours pas si le programme a fini son exécution (qui aura duré entre 20h et 29h) puisque l'ordinateur utilisé pour les tests est tombé en panne lors (à cause ?) de ce test.

Cette progression drastique et disproportionnée du temps d'exécution peut s'expliquer par la complexité de notre algorithme Python pur en O(n^4) (c.f Chapitre 3). En effet, supposons que notre entrée A contient n points, et B en contient 10n. Alors, là où l'entrée A nécessite n^4 opérations, l'entrée B a besoin de $(10n)^4 = 10^4 n^4 = 10000n^4$ opérations élémentaires, i.e 10000 fois plus que pour l'entrée A, pour une taille seulement 10 fois plus grande.

En partant de cette logique, le programme en *NumPy*, qui est en théorie de complexité O(n^3) (c.f Chapitre 3), devrait être, pour le même exemple, respectivement n et $10n$ fois plus rapide. Pourtant, en regardant les relevés de temps, on remarque que les temps d'exécutions *NumPy* sont excessivement plus grands que ceux en Python pur. Nous ne sommes pas sûrs d'avoir localisé la source exacte de ce problème, mais notre hypothèse est que là où le programme en Python ne compare les distances des points appartenant au cycle courant qu'avec ceux qui n'y sont pas, la version *NumPy* compare les distances avec tous les points du graphe, dont ceux du même cycle. Sur les derniers cycles les plus massifs, cela pourrait grandement affecter l'efficacité temporelle du programme.

Chapitre 6

Conclusion

Le but de notre projet était d'implémenter en Python et en *Numpy* un algorithme permettant de transformer un fichier *svg* contenant plusieurs cycles en un autre fichier *svg* ne contenant qu'un unique cycle, afin de faciliter l'impression 3D. Les résultats n'ont été qu'en partie concluants puisque le programme renvoie bien le résultat attendu, mais avec le temps d'exécution de la version *NumPy* plus long que celui en python pur alors que l'inverse devrait être observé.

Nous avons rencontré différents problèmes lors de la réalisation de ce projet, dont : une difficulté pour trouver des créneaux où tout le monde est disponible et non pas fatigués des cours antérieurs ; ce dernier point a pu causer des sessions de travail parfois très peu productives. Une répartition initiale inégale des tâches a aussi causé des complications. Nous avons cependant pu rebondir sur ce genre problèmes, par exemple en redistribuant les tâches ou en s'aidant ponctuellement. Le code a 4 mains s'est aussi révélé très efficace : deux cerveaux valent mieux qu'un. Le frein majeur a été la grande difficulté d'utilisation du logiciel *Github* au début du projet. Ce problème, comme tant d'autres, s'est réglé tout seul avec la pratique ainsi que l'expérimentation. D'autres petits contretemps ont occasionnellement changé notre organisation, comme la panne de la machine du groupe utilisée pour *benchmark* le programme. Celui-ci nous a paradoxalement permis de nous reconcentrer sur d'autres tâches importantes, comme l'implémentation en *NumPy* ou encore le début du rapport. Nous avons aussi eu des soucis de convention et de compréhension d'écriture de code menant à des difficultés pour aider un autre membre du groupe à écrire une partie qu'il/elle a déjà commencée. Ce point nécessite encore des améliorations ; une nomenclature plus stricte pour rendre le code plus lisible et accessible aux autres membres du groupe serait une bonne idée. De même, nous aurions aussi pu commencer l'implémentation *Numpy* du code en même temps que celle en python natif, ce qui aurait pu simplifier la répartition des tâches mais aussi permettre de plus se concentrer sur la version *Numpy*, qui est actuellement moins efficace que la version python native mais qui a pourtant plus de potentiel.
Pour aller plus loin dans le projet nous pourrions donc finir l'implémentation *Numpy* afin de la rendre plus rapide que la version native python.

Annexes

Annexe 1

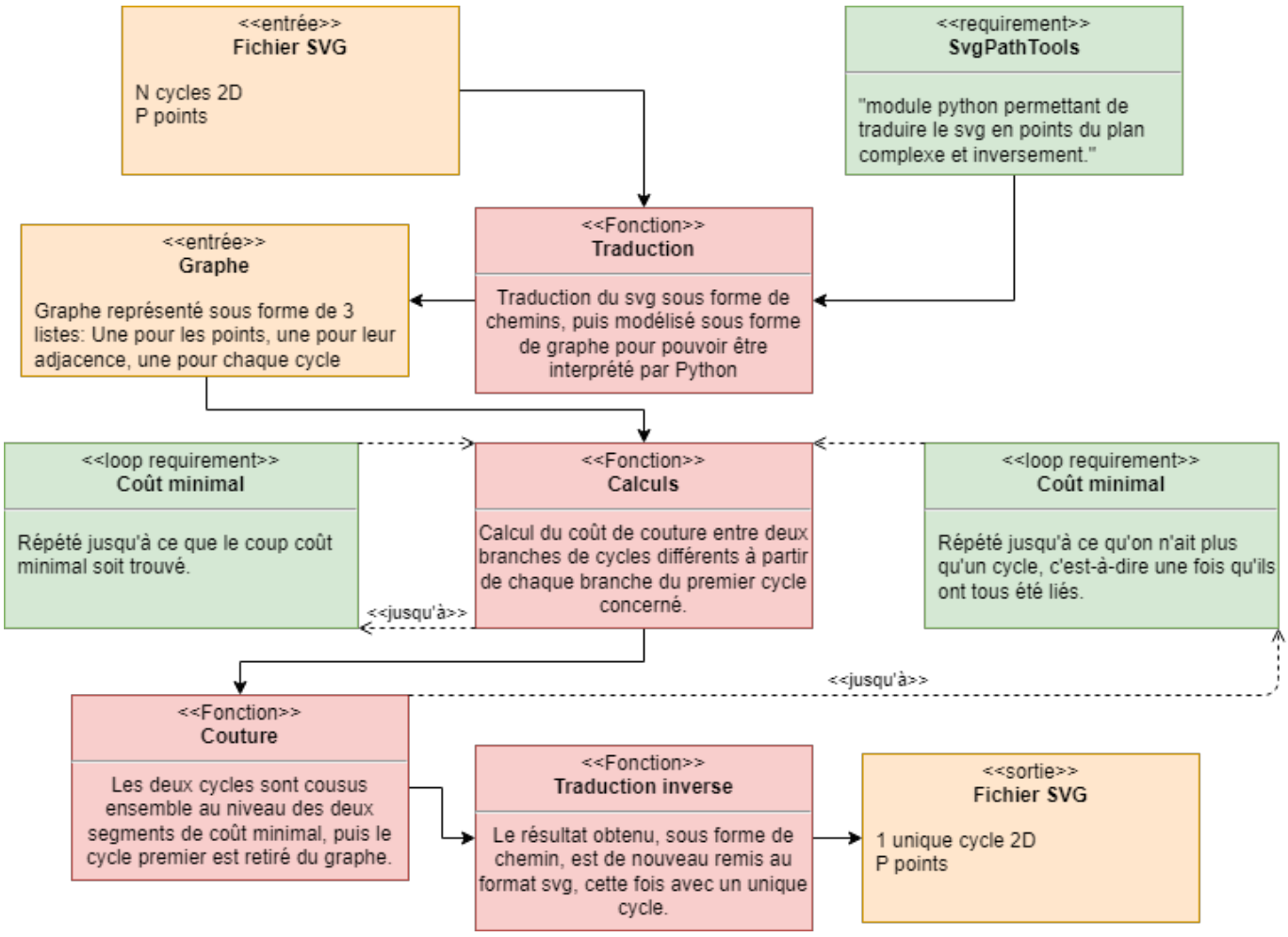


FIGURE 6.1 – Diagramme des entrées et sorties

avec $e = (u, v)$ un segment de cycle, l'énergie de couture entre e_1 et e_2 est :

$$s(e_1; e_2) = min(\, d(u_1; v_2) + d(v_1; u_2);\; d(u_1; u_2) + d(v_1; v_2)\,) - d(u_1; v_1) - d(u_2; v_2)$$

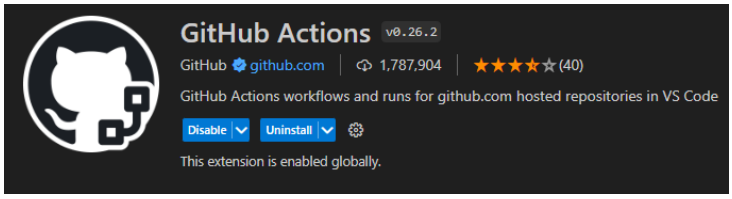
FIGURE 6.2 – Formule de l'énergie

$$\exists (x, y) \in \mathbb{R}^2,\ \begin{cases} y = a_1 x + b_1 & (1) \\ y = a_2 x + b_2 & (2) \end{cases}$$
$$d'où\ en\ faisant\ (2) - (1) :$$
$$\begin{cases} x = (b_2 - b_1)/(a_1 - a_2) \\ y = a_1 x + b_1 \end{cases}$$

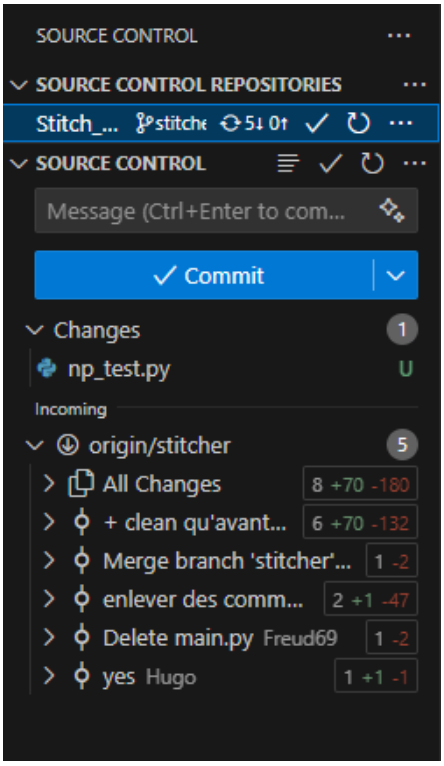
$Si\ a_1 = a_2$, alors les droites sont parallèles et il n'y a pas d'intersection

Si les deux équations de droites sont: $\begin{cases} x = c \\ y = ax + b \end{cases}$,
$alors\ (c, ac + b)$ est le point d'intersection des droites.

FIGURE 6.3 – Intersections de droite avec x et y donnés par la formule de Cramer

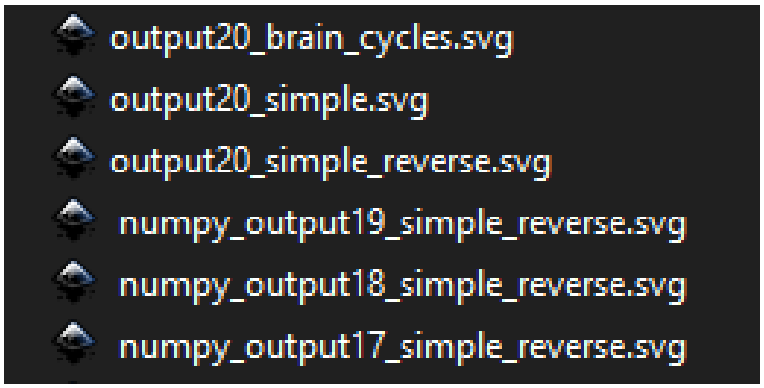


(a) Extension Github Actions permettant de gérer son répertoire depuis Github



(b) Contrôle de source permettant d'envoyer et recevoir les mises à jour du code

FIGURE 6.4 – Extension de l'environnement de travail Visual Studio Code utilisée.



(a) Dossier des résultats

```
paths2svg.wsvg(paths, filename=f'outputs\output{number_outputs + 1}_{filename}.svg')
```

(b) Nomination automatique du résultat à partir du nombre de fichiers dans le dossier

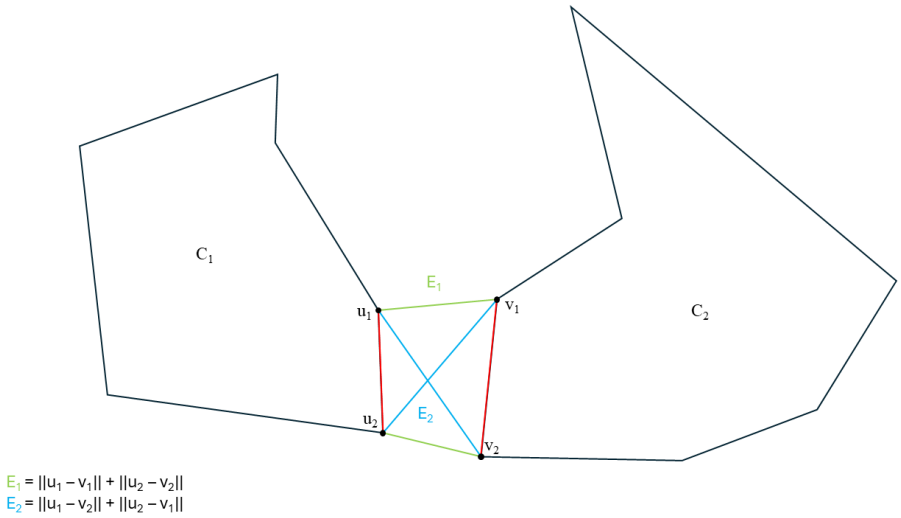
FIGURE 6.5 – Gestion de la sortie du programme à l'aide de *svgpathtools*, avec nomination automtique pour une meilleur visibilité des résultats

Algorithm 1 CyclesToGraph

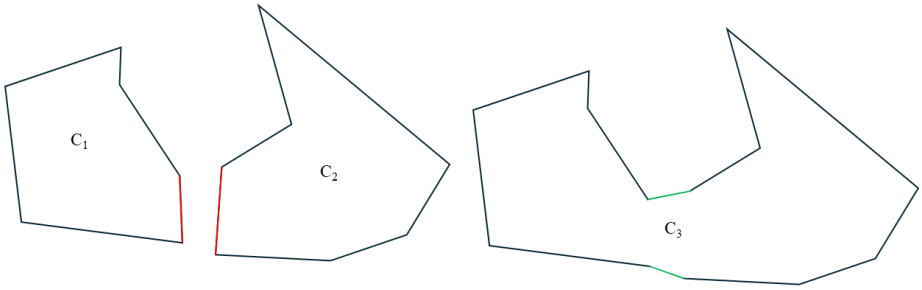
Require: svg
Ensure: Graph
ListePointsInitiale ← coordonnées complexes des points du fichier
$idCycle_A \leftarrow 0$
for *point dans ListePointInitiale* **do**
    $ListeCycles \leftarrow ajouter$[longueur ListePoints , longueur ListePointsInitiale]
    $ListeAdjacence \leftarrow ajouter$[IndicePointSuivant, IndicePointPrécédent]
    fin For ;
Fin .

Algorithm 2 StitchEdges

Require: svg file
Ensure: new path
ListePoints, ListeAdjacence, ListeCycles ← CyclesToGraph(svg file)
$idCycle_A \leftarrow$ *argmin(ListeCycles)*
for *i allant de 0 à longueur (ListeCycles - 1)* **do**
    isInverse, edge1Id, edge2Id, liaison ← NearestCycle($idCycle_A$)
    $idCycle_B \leftarrow$ SelectIdCycle($edge2_id$)
    $idFirstPoint \leftarrow edge2Id$
    if isInverse **then**
        Reverse(edge2Id)
        ChangeAdjacence(edge1Id, edge2Id, liaison)
        $idCycle_A \leftarrow$ MergeCycles($idcycle_B, idcycle_A$)
    fin For ;
new path ← $TraductionPath(ListeCycle)$
return new path
Fin .



(a) Ëvaluation des deux coûts/énergies de couture possibles



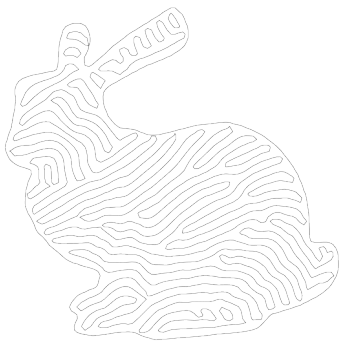
(b) Résultat de la couture : Entrée puis sortie

FIGURE 6.6 – Calcul de l'énergie pour savoir laquelle des deux coutures possible sera faite (diagonale ou non)

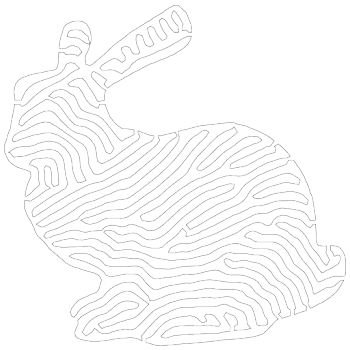
Annexe 2

(a) brain cycles - entrée

(b) brain cycles - sortie

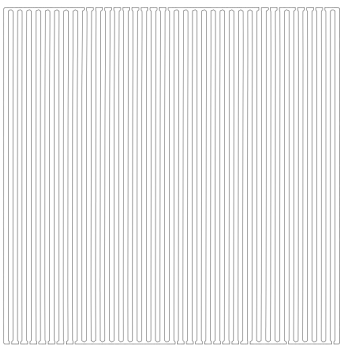


(c) small bunny cycles - entrée



(d) small bunny cycles - sortie

(e) square cycles - entrée



(f) square cycles - sortie

FIGURE 6.7 – Exemples de différents résultats

Nom du fichier	*Temps d'exécution (HH:MM:ss.mmm) algo 1*	*Temps d'exécution (HH:MM:ss.mmm) algo 2*	*Temps d'exécution (hh:mm:ss.mmm) algo 3 (NumPy)*	*Nombre de points*	*Taille (ko)*
simple	00:00:00.007	00:00:00.008	00:00:00.049	16	2
simple_reverse	00:00:00.007	00:00:00.008	00:00:00.027	16	2
brain_cycles	00:01:15.935	00:01:09.066	00:26:05:316	2122	81
bunny_small_cycles	NA	00:03:18.904	00:59:32:249	2974	111
square_cycles	NA	01:20:19.952	NA	11301	408
dragon_cycles	NA	NA	NA	124763	4567
bunny_big_cycles	NA	NA	NA	188544	6879

FIGURE 6.8 – Temps d'exécution pour chaque résultat. L'algo 1 est celui qui utilise la formule de Cramer pour vérifier les intersections. L'algo 2 utilise uniquement la formule de l'énergie.

Bibliographie

[1] Xavier Chermain, Cédric Zanni, Jonàs Martínez, Pierre-Alexandre Hugron, and Sylvain Lefebvre. "orientable dense cyclic infill for anisotropic appearance fabrication". *ACM Trans. Graph. (Proc. SIGGRAPH)*, 42(4), 2023.

[2] Andrew B Kahng and Sherief Reda. "match twice and stitch : a new tsp tour construction heuristic". *Operations Research Letters*, 32(6) :499–509, 2004.

[3] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, 2001.

[4] Wikipedia contributors. Cramer's rule — Wikipedia, the free encyclopedia. https ://en.wikipedia.org/w/index.php ?title=Cramer [Online ; accessed 5-April-2024].

[5] Wikipédia. Intersection (géométrie) — wikipédia, l'encyclopédie libre, 2023. [En ligne ; Page disponible le 1-juin-2023].