Jeux évolutionnaires
- 3DNA -

- Compte rendu -
Enseignement d'intégration

CentraleSupélec - Université Paris-Saclay, Gif-Sur-Yvette
02 Février 2024

DAVID Erwan - FAYNOT Guillaume - SHINTANI Hiyu - ZAYANE Ali

Objectifs

Le projet réalisé durant cette semaine propose de modifier un modèle de conformation 3D connu, permettant de convertir une suite de nucléotides en une trajectoire tridimensionnelle, afin de rendre un plasmide circulaire. Le but est d'améliorer le modèle pour le rendre conforme à la réalité.
Deux algorithmes méta-heuristiques différents seront développés et comparés : un recuit simulé et un algorithme génétique. Ils seront développés en *Python* et implémentés sous forme de classes.

Table des matières

I. Introduction

Chaque cellule constitutive de la vie sur Terre contient une ou plusieurs molécules d'ADN, qui servent de support à l'information génétique. Ces molécules, de longueurs variables, se composent d'une séquence de nucléotides (ou bases : A, C, G et T) qui interagissent avec divers éléments cellulaires. Leur positionnement spatial joue un rôle crucial dans l'adaptation de la cellule à son environnement, que ce soit en réponse à la chaleur, à la famine, au stress, etc. Bien que les séquences d'ADN soient couramment étudiées à travers leur séquence textuelle, consistant en une succession de lettres A, C, G et T, il est instructif de les examiner également du point de vue de leur trajectoire tridimensionnelle. En 1993, des biophysiciens ont développé un modèle de conformation 3D permettant de convertir une suite de nucléotides, exprimée sous forme de lettres, en une trajectoire tridimensionnelle. Ainsi, il devient possible de représenter toute séquence textuelle d'ADN sous la forme d'une trajectoire 3D.

Étant conçu pour des séquences courtes d'ADN non lié, ce modèle ne tient pas compte de toutes les caractéristiques d'une longue chaîne à l'intérieur de la cellule, telles que les surenroulements, les nucléosomes et les interactions à longue distance. Par exemple, lorsqu'on examine un chromosome bactérien (une longue séquence d'ADN constituant une bactérie) ou un plasmide (une petite séquence présente chez les bactéries), on constate que ces chromosomes ou plasmides sont circulaires, c'est-à-dire que leurs deux extrémités ont été fusionnées. Le modèle mentionné précédemment ne tient pas compte de ce phénomène lorsqu'on représente la trajectoire 3D d'un chromosome bactérien ou d'un plasmide.

L'objectif est de décrire l'enroulement observé à grande échelle (c'est-à-dire à l'échelle d'un plasmide par exemple) à partir d'un modèle microscopique (l'agencement dans l'espace de deux nucléotides successifs). On dispose ainsi d'une table de rotation construite à partir d'observations, qui dicte la manière dont s'oriente un nucléotide donné par rapport à son prédécesseur. L'objectif est de trouver une une table de rotation avec des valeurs voisines de celle dont on dispose, et qui permette de modéliser un plasmide circulaire à grande échelle. Ce rapport se propose d'utiliser deux méthodes distinctes pour résoudre ce problème d'optimisation : un recuit simulé et un algorithme génétique. Ils seront développés en *Python* et implémentés sous forme de classes.

Quand ce n'est pas précisé dans ce rapport, les études portent sur le *Plasmid 8k*.

II. Contraintes

1. Contexte

Il est fourni avec les fichiers de tracé et de calcul de la trajectoire de séquence une table de rotation. Elle contient trois valeurs d'angles ($Twist$ (Ω), $Wedge$ (σ) et $Direction$ (δ)) , qui définissent une loi d'évolution de l'orientation de chaque dinucléotide, ainsi que leurs écarts types associés. Afin de trouver une table voisine qui répond au problème, ie obtenir une trajectoire de séquence ADN réaliste, les valeurs de la table de rotation initiale seront perturbées en respectant deux contraintes :

2. Contrainte d'écart-type

Les écarts-types peuvent permettre de définir un intervalle dans lesquels les valeurs de la table de rotation se trouvent avec certitude. Lorsque les coefficients sont perturbés, il est nécessaire qu'ils restent dans un intervalle de demi-largeur écart-type, centré en la valeur initiale. Ainsi la table de valeur obtenue in fine est cohérente avec les observations.
Remarque : les valeurs de $Direction$ ne sont pas modifiées étant donné que les écarts-types qui leur sont associés sont nuls respectivement.

3. Contrainte de symétrie

Dans une séquence d'ADN, l'information est codée en double, chaque nucléotide possédant un complémentaire (A et T sont complémentaires, C et G sont complémentaires). Toute chaîne de nucléotides (brin) possède donc un brin complémentaire qui lui est lié.

Figure 1: Brin principal (haut) et brin complémentaire(bas)

Ainsi les valeurs de $Twist$ et $Wedge$ d'un dinucléotide et de son complémentaire sont les mêmes.
On retrouve dans la table six cas de symétries dans lesquels les valeurs des rotations $Twist$ et $Wedge$ sont identiques:

AA	**TT**
AC	**GT**
AG	**CT**
CA	**TG**
CC	**GG**
GA	**TC**

Tableau 1: loi de symétrie des dinucléotides

Quatre paires de nucléotides sont leur propre symétrique : AT, TA, CG, GC.

Ces contraintes seront respectées lors de l'implémentation des algorithmes de perturbation.

III. Fonction objectif

Que ce soit à travers l'algorithme de recuit simulé ou l'algorithme génétique, une fonction "objectif" doit être définie. Cette fonction évalue la qualité de notre solution. Dans le recuit, elle sert à comparer la solution actuelle aux solutions explorées. Dans l'algorithme génétique, elle sert à évaluer les individus en prévision de la phase de sélection.

Le but est de minimiser la fonction objectif à l'image d'une fonction coût en Machine Learning. Un plasmide est effectivement circulaire s'il boucle sur lui-même et si cette boucle est physiquement réaliste, au sens des contraintes géométriques imposées par la table de rotation. Il faut donc calculer la distance ainsi que l'angle entre le premier et le dernier nucléotide.

La valeur retournée par la fonction objectif est la somme de deux termes différents :

- **Le terme de distance :** L'objectif de ce terme est de boucler la chaîne de plasmide. On définit alors la $Distance$ comme représentée dans le schéma ci-dessous:

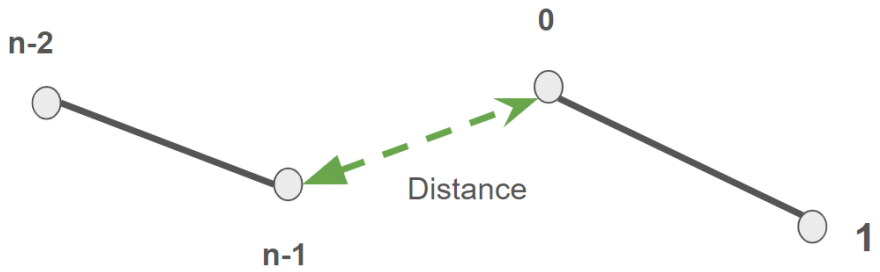


Figure 2 : Schéma présentant la loi de distance

De plus, on introduit une variable Python "relier" spécifiant le nombre de nucléotides virtuels ajoutés à la fin de la séquence qui doivent être superposés aux nucléotides du début. En pratique, relier un seul nucléotide virtuel est suffisant. La moyenne des distances entre ces nucléotides virtuels et ceux du début de la séquence est ensuite calculée. Cela permet de s'assurer que les contraintes de la table sont respectées au point de bouclage du plasmide. Voici le code Python correspondant à cette fonctionnalité :

```python
return (sommeDist / self.relier) + ...
```

Si la variable "relier" vaut 0, seulement la distance entre le début et la fin de la séquence est calculée. Il n'y a pas de notions de contraintes angulaire de la table dans ce terme de distance. La valeur 3 correspond à la distance moyenne estimée entre deux dinucléotides dans le plasmide. Voici le code Python correspondant :

```python
return abs(dist - 3) + ...
```

- **Le terme d'angle :** L'objectif de ce terme est de tenir compte des contraintes angulaires géométriques entre le dernier dinucléotide et le premier dinucléotide, c'est-à-dire que le plasmide se referme en une boucle à courbure lisse, et non avec un coude aigü. On définit pour ce faire $v_{début}$, v_{milieu} et v_{fin}:

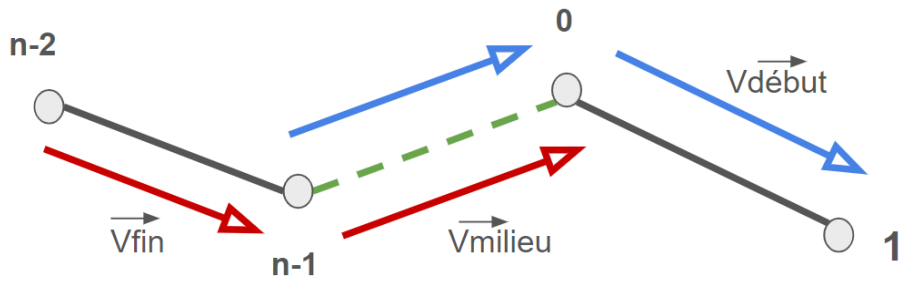


Figure 3 : Schéma présentant la loi d'angle

En notant, p_i la position du ième dinucléotide, et n le nombre de nucléotides dans la séquence, les vecteurs s'écrivent:

$$v_{fin} = p_{n-1} - p_{n-2}, \qquad v_{milieu} = p_0 - p_{n-1}, \qquad v_{début} = p_1 - p_0$$

Les similarités cosinus entre v_{milieu} et $v_{début}$ et v_{fin} et v_{milieu} sont respectivement notées $dot1$ et $dot2$. Ce sont les produits scalaires normalisés. Pour chaque produit scalaire, si les deux vecteurs sont orientés dans le même sens et la même direction, cette quantité tend vers 1.

La distance entre le début et la fin est notée $dist = ||p_{n-1} - p_0||_2$. Le terme d'angle est défini par :

$$6\exp(-\frac{dist}{50})(1 - \exp(-0.8((dot1 - 1)^2 + (dot2 - 1)^2))$$

- 6 est un facteur empirique d'amplitude.
- $\exp(-\frac{dist}{50})$ permet d'augmenter l'importance du terme d'angle lorsque la distance commence à être inférieure à ~ 100. Ainsi la fonction objectif tend d'abord à optimiser le terme de distance avant d'optimiser le terme d'angle. Cela permet une convergence plus rapide des solutions.
- Le dernier terme est représenté dans la figure 1. Il permet d'avoir un facteur qui est minimal lorsque $(dot1, dot2)$ est proche de $(1, 1)$, c'est à dire, lorsque les vecteurs v_{fin}, v_{milieu} et $v_{début}$ sont alignés et dans le même sens.

Voici le graphique de la fonction qui contrôle l'alignement des vecteurs :

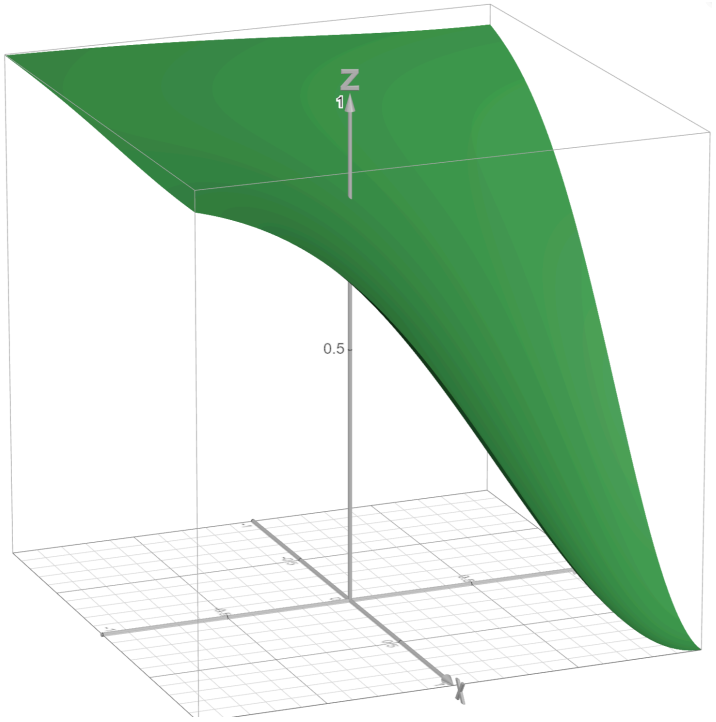


Figure 4 : Graphe de $z = 1 - \exp(-0.8((x-1)^2 + (y-1)^2))$

Ce graphique est un puit de potentiel qui capture les solutions qui alignent les vecteurs v_{fin}, v_{milieu} et $v_{début}$. Dans le cas d'un alignement, cette fonction est nulle, minimisant ainsi la fonction objectif.

Finalement, la fonction objectif est définie comme suit :

```python
return (sommeDist / self.relier) + 6. * exp(-dist / 50) * (1. - exp(-0.8*((dot1 - 1) ** 2 + (dot2 - 1) ** 2)))
```

Aussi, la distance entre le premier et le dernier nucléotide, ainsi que la contrainte angulaire doivent être optimisées par les deux algorithmes mis en place. La même fonction objectif sera donc utilisée dans les deux algorithmes.

IV. Recuit simulé

1. Présentation

Le recuit simulé est une technique d'optimisation inspirée du processus physique de recuit métallurgique. Le principe est de passer d'une exploration de l'environnement des solutions à une exploitation des solutions voisines à une certaine vitesse. Cette vitesse est représentée par un seuil de probabilité d'exploration. Il est donné par une exponentielle décroissante, analogue à la température dans un recuit métallurgique.

Voici son expression :

$$Seuil_{exploration} = exp(\frac{-\Delta Energie}{T°}),$$

- $\Delta Energie$: la différence de coût entre la solution voisine et la solution actuelle.
- $T°$: la température.

Ainsi, plus la température diminue, plus notre problème "refroidit", plus le seuil de probabilité d'exploration est faible. L'exploitation est alors privilégiée au début du processus.

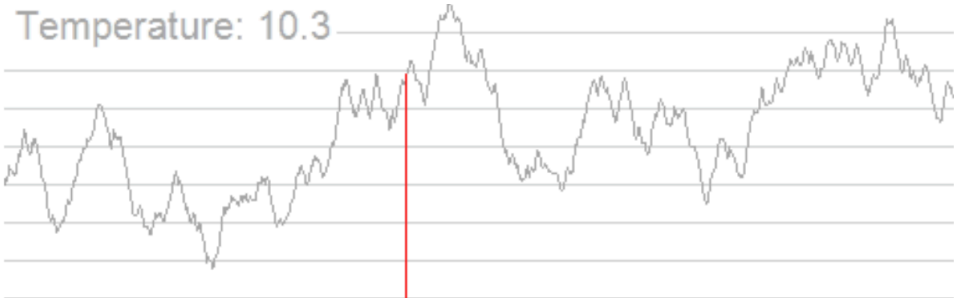


Figure 5 : Exemple de parcours des solutions à T = 10.3 - recuit simulé

2. Implémentation

La méthode d'optimisation par recuit simulé est décrite dans le fichier *recuit.py* qui introduit la classe *Recuit()*. Les critères d'arrêt sont choisis de sorte à limiter le nombre d'itération, ou arrêter l'exécution de l'algorithme lorsque que le niveau d'énergie atteint est jugée satisfaisant.

Dans un premier temps, on initialise les paramètres de la simulation. On définit :

- La séquence d'ADN à prendre en compte : seq. Cette séquence est contenue dans un fichier au format *.fasta*
- Le nombre d'itérations maximal de l'algorithme : k_{max}
- Le minimum d'énergie à atteindre : e_{min}
- Le minimum de distance à atteindre : $dist_{min}$
- La température initiale : $temp_{init}$
- Une vitesse de refroidissement : $refroidissement$
- Le nombre de nucléotides reliés entre la fin et le début de la séquence : $relier$

Le recuit simulé utilise un objet de la classe $RotTable$ en tant qu'état. Le voisin d'un état donné est généré en prenant en compte la symétrie présente au sein des paires de nucléotides. Pour chaque paire de nucléotide à modifier, les nouvelles valeurs de $Twist$ et $Wedge$ sont tirées suivant une distribution gaussienne qui a pour espérance la précédente valeur, et pour écart type $stdFact \times limit$.

- $limit$ est la valeur limite de la rotation définie dans l'objet $RotTable$
- $stdFact$ est un facteur qui décroît lorsque la température décroît, afin que le voisin reste proche de l'état précédent lorsque la température diminue. Il est définit par

$$stdFact = \frac{T}{3T_{init}}$$

où T est la température actuelle, et T_{init} la température initiale. Le facteur $\frac{1}{3}$ a été optimisé de manière empirique.

Toutes ces nouvelles valeurs de rotations sont ensuite bornées afin de respecter la limite définie par la contrainte d'écart-type.

Remarque : la valeur de $Direction$ n'est pas modifiée étant donné que l'écart-type qui lui est associé est nul.

3. Résultats

A. Temps de convergence

L'algorithme de recuit simulé est très rapide. Il tourne en un temps inférieur à quelques dizaines de seconde pour 1000 itérations en moyenne sur un ordinateur portable de puissance normale avec le plasmide 8k. En revanche, il est plus lent pour plasmid_180k.

Séqunce ADN	Temps de convergence observé
plasmid_8k	~10s
plasmid_180k	~4 min

Tableau 2 : temps de convergence en fonction de la séquence ADN pour le recuit simulé

Note : Par souci d'optimisation de temps de calcul dans le but de présenter des résultats significatifs, les résultats présentés par la suite sont pour plasmid_8k.

B. Convergence

Niveau convergence et vitesse, les résultats sont très acceptables. Le graphique ci-dessous nous montre que pour 5 seed différentes du module *random* la convergence est atteinte aux bout d'environ 500 générations :

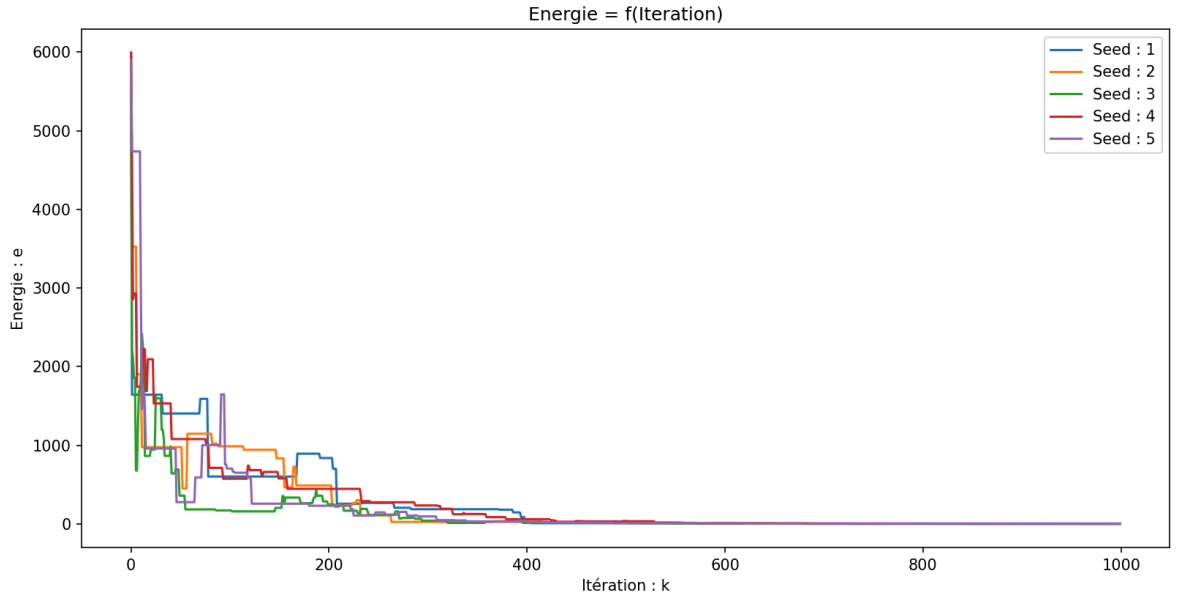


Figure 6 : Energie en fonction des itérations sur 5 seed différentes - RECUIT

C. Optimisation du paramètre T_init

Une étude d'évolution de l'énergie en fonction de la condition initiale à même d'être choisie dans le plus grand intervalle, la température initiale, a été réalisée dans ce cadre du recuit simulé. L'objectif est de trouver une valeur optimale pour T_{init}.

Les résultats sont les suivants:

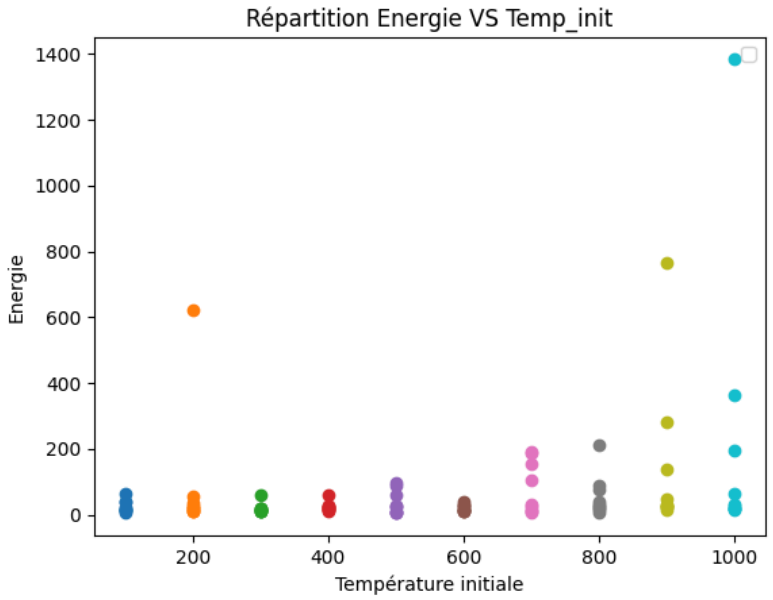


Figure 7 : Energie en fonction de la température initiale pour plasmid_8k

L'étude à été réalisée sur 10 simulations, une pour chaque valeur de T_{init} = 100, 200, 300 etc (chacune représentée par une couleur) avec 700 itérations maximum.

En retirant les valeurs aberrantes dues à des non-convergences exceptionnelles du modèle, les moyennes et écarts-types sur les énergies sont les suivants :

Température	100	200	**300**	400	500	600	700	800	900	1000
Energie moyenne	24.55	23.46	**18.68**	24.58	35.44	20.98	74.86	53.60	67.83	83.87
Ecart type	17.67	13.92	**15.54**	13.22	33.75	10.18	76.75	60.87	87.78	119.44

Tableau 3 : moyennes et écarts-types des résultats

On remarque l'existence de paramètres initiaux plus favorables à une bonne convergence et avec une bonne reproductibilité : ici $T_{init} = 300$. On remarque une plus grande incertitude sur la convergence lorsque que T_{init} croît.

D. Etude de variabilité

Une étude de similarité est réalisée dans le but d'obtenir une moyenne des valeurs et un écart type sur plusieurs centaines d'essais par une méthode de Monte-Carlo. Un serveur AWS de 32 cœurs a été loué afin de réaliser ces centaines d'essais en parallèle.

Ainsi, en répétant l'algorithme un grand nombre de fois, la distribution de la distance entre le début et la fin et du produit scalaire :

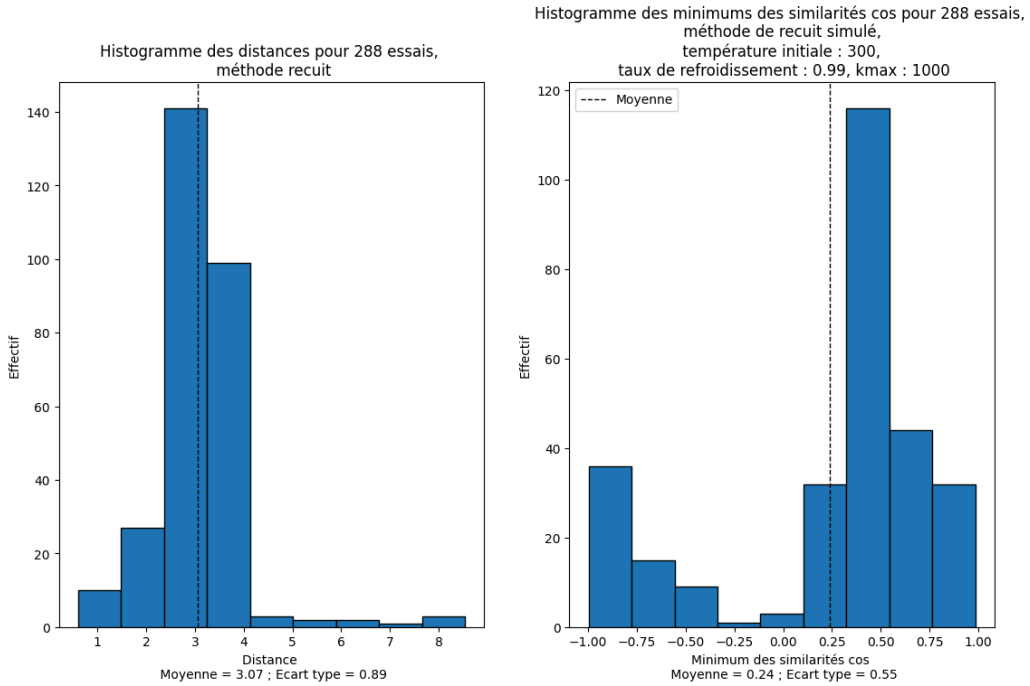


Figure 8 : Distribution de la distance entre le début et la fin de la séquence, et similarité cosinus pour 288 itérations du recuit simulé

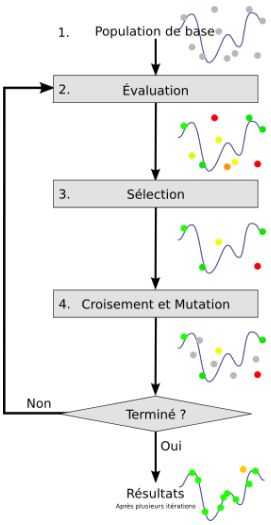
D'après ces résultats, la distance entre le début et la fin de la séquence tend à avoir une moyenne de 3.08 et un écart type de 0.89. Ce résultat est cohérent avec les deux translations de $\frac{3.38}{2}$ et la rotation effectuée entre les dinucléotides. Le minimum de la similarité cosinus, quant à lui, est dans la majorité des cas positif, ce qui se traduit par un alignement naturel, mais dans un quart des cas négatif : fermeture en coude. Ces cas de fermeture de coude sont peu fréquents, et les distributions sont très resserrées autour de bonnes valeurs, le modèle est globalement bon et stable.

V. Algorithme génétique

1. Présentation

Un algorithme génétique est une technique d'optimisation inspirée par le processus de sélection naturelle observé dans l'évolution biologique. Il s'agit d'une méthode de recherche heuristique qui est utilisée pour résoudre des problèmes d'optimisation et de recherche. Les algorithmes génétiques sont souvent employés pour trouver des solutions approximatives à des problèmes complexes, notamment dans les domaines de l'ingénierie, de l'optimisation, de l'apprentissage automatique, et d'autres domaines où une exploration approfondie de l'espace des solutions est nécessaire.

Le processus des algorithmes génétiques s'inspire des mécanismes biologiques. L'algorithme mis en place dans ce projet se déroule en 5 étapes :



A. **Genèse :** Création d'une population initiale de N individus.
B. **Evaluation des individus :** Quantification de la qualité de tous les individus avec la fonction objectif expliquée **partie III**.
C. **Sélection :** Sélection de $\frac{N}{2}$ individus à garder pour la génération $k + 1$. Plusieurs méthodes de sélection sont définies.
D. **Croisement :** Pour chaque couple, partage des gènes des 2 individus parents selon une méthode de croisement multi-points.
E. **Mutation :** Mutation aléatoire des individus selon une distribution Gaussienne centrée sur leur valeur actuelle.
F. **Terminer :** Arrêt de la sélection au bout de k_{max} itérations.

2. Implémentation

A. Présentation

La méthode d'optimisation par algorithme génétique est décrite dans le fichier *genetique.py* qui introduit la classe ***Genetique()***.

Les paramètres de la simulation sont initialisés comme suit :

- La séquence d'ADN à prendre en compte : seq
- Le nom de la stratégie de sélection : $methode\_utilisee$
- Un nombre de générations maximales décrivant l'évolution : $nbr\_generation\_max$
- Un nombre d'individus : N
- Une probabilité de mutation initiale : $probabilite\_mutation\_initiale$
- Une probabilité de mutation finale: $probabilite\_mutation\_finale$
- Le nombre de nucléotides reliés entre la fin et le début de la séquence : $relier$

Ci-dessous sont présentées les différentes méthodes et choix d'implémentation par étape clef de l'algorithme :

B. Genèse

La fonction *generation_population()* génère une population constituée d'individus. Un individu est une table de rotation générée aléatoirement en respectant le modèle donné par ***RotTable()*** ainsi que les symétries. Elle suit une loi gaussienne qui a pour espérance la valeur initiale donnée par ***RotTable()***, et pour écart-type le tiers de la limite de cette valeur. Ce facteur $\frac{1}{3}$ permet de garantir le fait que 98% des valeurs générées tombent dans l'intervalle limite, ceux en dehors de cet intervalle ont comme valeur la borne de cet intervalle.

C. Evaluation des individus

L'algorithme génétique partage la même fonction objectif que la méthode de recuit simulé. Par conséquent, cette dernière est appelée énergie dans la suite du rapport et dans le code.

D. Sélection

Plusieurs méthodes de sélection traditionnellement mises en œuvre dans un algorithme génétique sont implémentées. Chaque méthode de sélection sélectionne $\frac{N}{2}$ individus :

Méthode de sélection	Critère de sélection	Fonction dans le code
Sélection par élitisme	La première moitié des individus, triés par par leur énergie, est sélectionnée	*selection_elitisme()*
Sélection par roulette	Les individus sont sélectionnés avec une pondération correspondant à leur énergie. Les énergies sont inversées et normalisées, de sorte que la probabilité d'être sélectionné est importante pour une énergie faible : $p_i = \frac{\frac{1}{e_i}}{\Sigma \frac{1}{e_j}}$	*selection_roulette()*
Sélection par rang	Les individus sont trié par énergie croissante et leur probabilité d'être sélectionné décroît lorsque que leur rang dans le tri croît	*selection_rang()*
Sélection par tournoi	A chaque tour, deux individus sont choisis aléatoirement et leurs énergies sont comparées. Le gagnant (celui qui a la plus grande énergie) est sélectionné avec une probabilité de 0.99. La moitié de la population initiale est sélectionnée.	*selection_tournoi()*

Tableau 4 : Définition des critères de sélection

E. Croisement

Les croisements sont réalisés en 1 ou N points entre deux individus pour produire deux enfants anti-symétriques. Ces opérations sont réalisées par les 2 fonctions *croisement()* et *croisement_N_points()*.

F. Mutation

La mutation est définie dans la fonction *mutation()*. La probabilité de muter une paire de nucléotide est définie par une interpolation linéaire entre $probabilite\_mutation\_initiale$ et $probabilite\_mutation\_finale$ en fonction de l'itération. Plus le nombre d'itérations augmente, plus la probabilité de mutation diminue. L'exploration, tout comme dans l'algorithme de recuit, est favorisée au début de "l'apprentissage".

La mutation modifie les valeurs de la table des rotations d'une paire de nucléotides, en respectant les contraintes définis **partie II**. La modification suit une distribution gaussienne qui a pour espérance la précédente valeur, et pour écart type $stdFact \times limit$.

Les paramètres sont :

- $limit$ est la valeur limite de la rotation définie dans l'objet $RotTable$
- $stdFact$ est un facteur qui décroît lorsque le nombre d'itération croît. L'état muté reste donc proche de l'état précédent lorsque le nombre d'itération augmente. Il est définit par :

$$stdFact = 1 - \frac{n}{3\ nbr\_generation\_max}$$

n est le nombre d'itération actuelle. Le facteur $\frac{1}{3}$ a été optimisé de manière empirique.

Toutes ces nouvelles valeurs de rotations sont ensuite bornées afin de respecter la limite définie.

L'algorithme présenté comporte quelques modifications le rendant plus performant:

- Le meilleur individu, d'énergie minimale, n'est pas muté afin de stabiliser le processus d'optimisation
- Les deux derniers individus, d'énergies maximales, sont plus fortement mutés afin d'explorer de nouveaux états. Le dernier est muté avec $stdFact = 0.9$ et une probabilité de 4% afin d'explorer un état éloigné. L'avant dernier est muté avec $stdFact = 0.06$ et une probabilité de 90%, il explore des états proches. Ce sont donc des individus "explorateur" qui ont pour rôle de continuer à explorer des possibilités même à la fin de l'"apprentissage". Dans le pire cas, aucun des 2 n'est sélectionné.

G. Terminer

Cette étape marque la fin de l'algorithme. C'est la seule condition d'arrêt.

3. Résultats

A. Temps de convergence

L'algorithme génétique est relativement lent. Son temps d'exécution pour 50 individus sur 150 générations est de l'ordre de plusieurs dizaines de secondes en moyenne sur un ordinateur portable de puissance normale avec la séquence plasmid_8k.

B. Convergence

Les résultats sont acceptables. Le graphique ci-dessous nous montre que pour 3 seed différentes du module *random* la convergence est atteinte aux bout d'environ 100 générations.

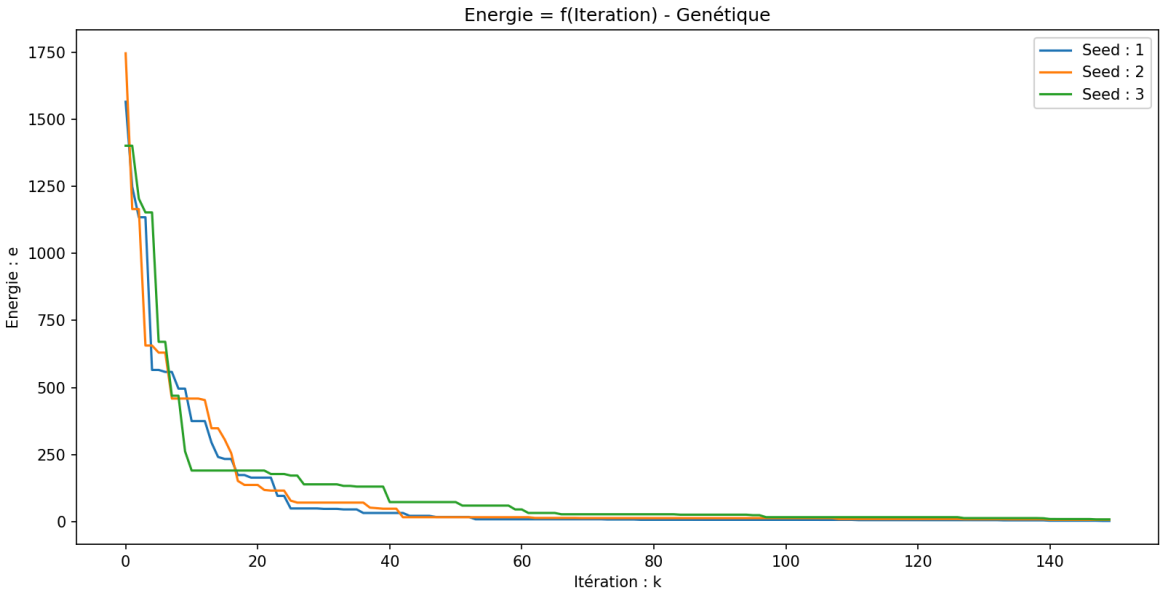


Figure 9 : Energie en fonction des itérations sur 3 seeds différentes - GÉNÉTIQUE

C. Optimisation du paramètre Nombre individus

Tout comme le recuit simulé, ci-dessous une analyse d'optimisation du paramètre Nombre d'individus.

En retirant les valeurs aberrantes dues à des non-convergences exceptionnelles du modèle, on obtient les moyennes et écarts-types sur les énergies suivants:

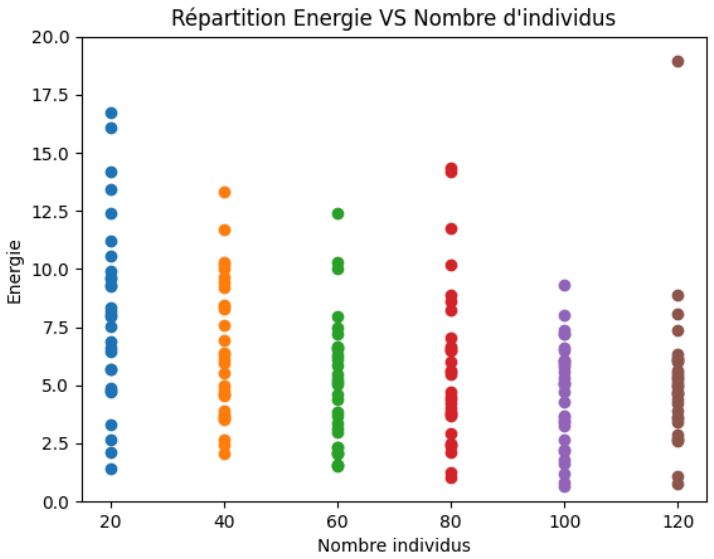


Figure 14 : Energie en fonction du nombre d'individu pour plasmid_8k

Les moyennes et écarts-types sur les énergies sont les suivants:

N	20	40	60	80	100	120
Energie moyenne	17,12	6,45	6,16	5,74	4,63	5,18
Ecart type	35,14	2,89	6,81	3,31	2,19	3,02

Tableau 6 : Moyennes et écarts types des énergies minimales pour plusieurs nombres d'individus

Mis de côté les valeurs de $N < 40$, le tableau ci-dessus présente des résultats très similaires tant au niveau des moyennes que des écarts-types. L'utilisation d'un $N = 50$, récurrent dans ce rapport, est un choix qui permet des simulations rapides et performantes.

D. Etude de variabilité

De même que pour le recuit simulé, une étude de similarité est réalisée dans le but d'obtenir une moyenne des valeurs et un écart type sur plusieurs centaines d'essais. Un serveur AWS de 32 cœurs a été loué afin de réaliser ces centaines d'essais en parallèle.

Ainsi, un grand nombre d'itérations de l'algorithme génétique a été effectué avec les différentes méthodes de sélection. Voici les distributions obtenues :

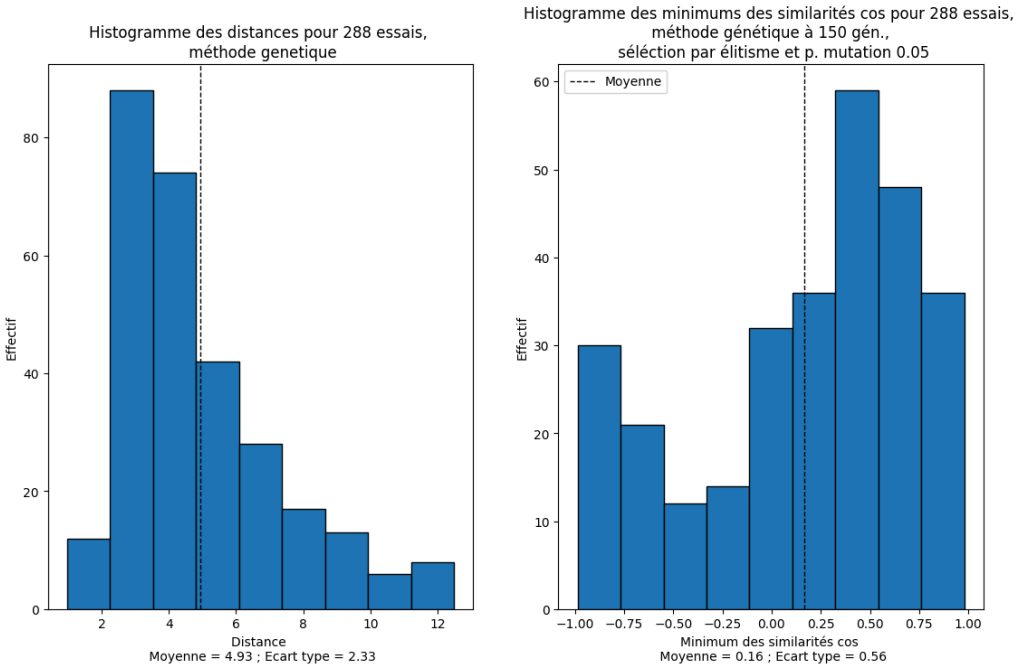


*Figure 10 : Distribution de la distance entre le début et la fin de la séquence, et similarité cosinus pour 288 itérations de la méthode génétique avec une sélection par **élitisme***

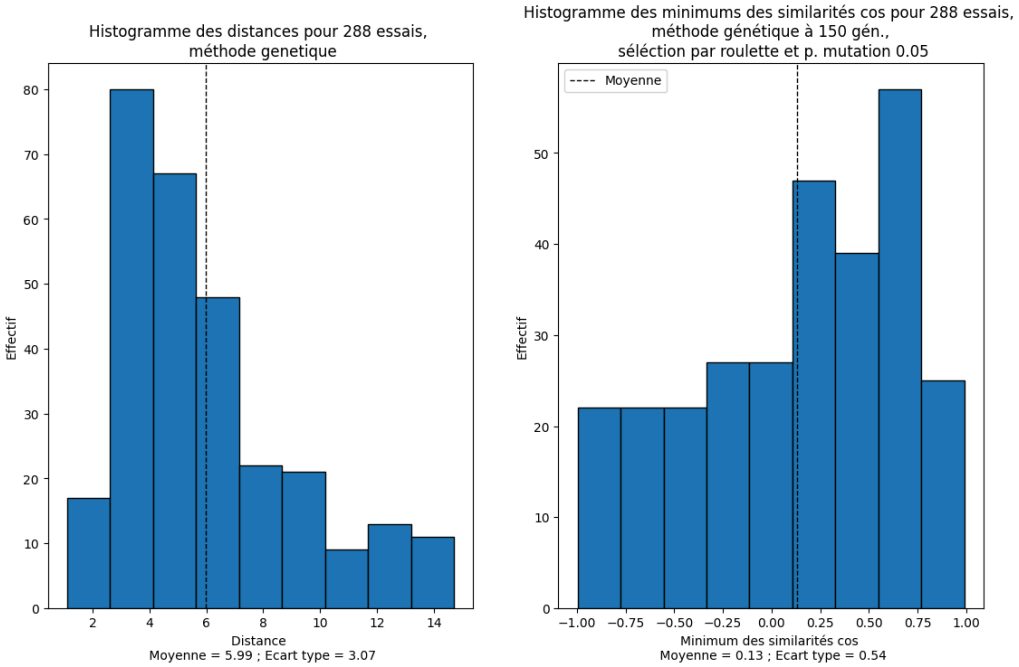


*Figure 11 : Distribution de la distance entre le début et la fin de la séquence, et similarité cosinus pour 288 itérations de la méthode génétique avec une sélection par **roulette***

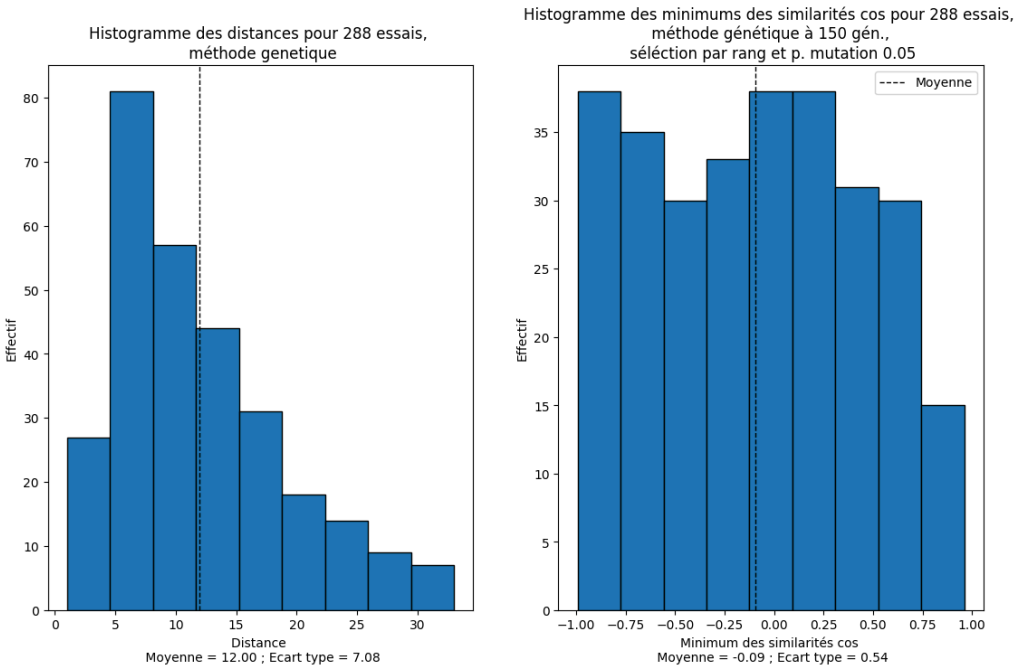


*Figure 12 : Distribution de la distance entre le début et la fin de la séquence, et similarité cosinus pour 288 itérations de la méthode génétique avec une sélection par **rang***

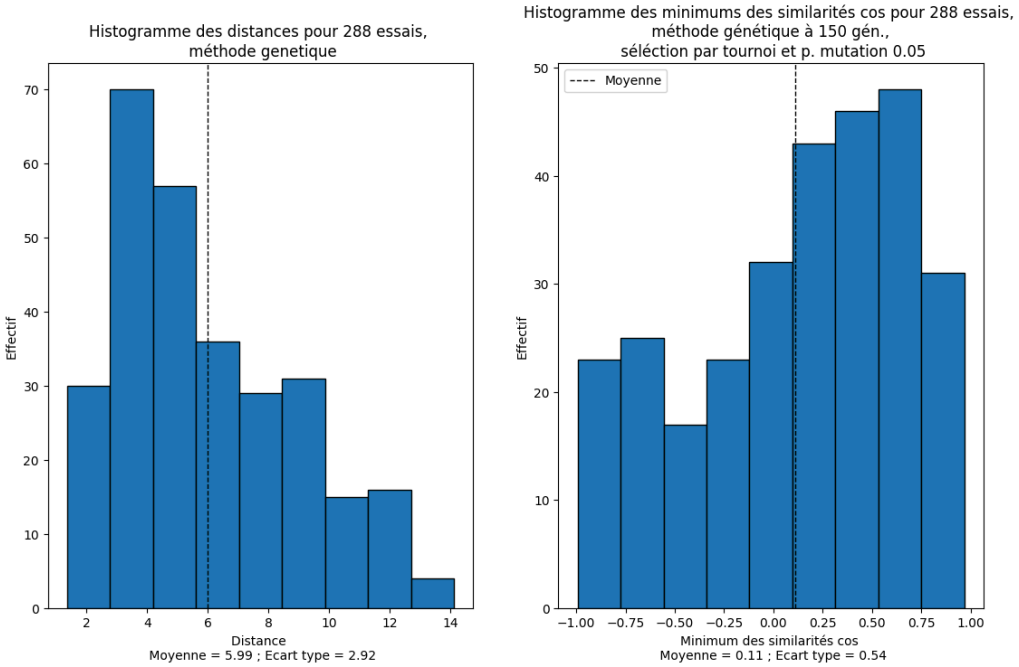


*Figure 13 : Distribution de la distance entre le début et la fin de la séquence, et similarité cosinus pour 288 itérations de la méthode génétique avec une sélection par **tournoi***

Ci-dessous une comparaison qualitative des résultats obtenus pour des paramètres initiaux similaires: Séquence plasmid_8k, N = 50 individus, 150 générations, p. mutation = 0,05.
La machine utilisée est un ordinateur portable de puissance normale.

Méthode	Temps de convergence observé	Distance	Produit scalaire
Élitisme	98s	++	+
Roulette	96s	+	_ _
Rang	97s	_ _	_
Tournoi	92s	+	+

Tableau 5 : Comparaison des méthodes de sélection

On remarque que la sélection par élitisme est celle qui présente les meilleurs résultats, avec une distribution des distances concentrée autour de 3 et de similarité cosinus concentrée autour de 0.5. La sélection par rang donne les moins bons résultats, notamment en termes de similarité cosinus, pour laquelle la distribution semble être uniforme entre -1 et 1. Cela est cohérent avec le fait que ce mode de sélection est le “moins violent” dans la mesure où la valeur de l’énergie n’intervient pas, seul le classement importe, ce qui laisse plus de possibilité aux individus moins bons de se reproduire.

Concernant l'évolution du minimum d'énergie en fonction du nombre d'individus, des algorithmes génétiques avec sélection par élitisme sont lancés et les résultats peuvent être visualisés ci-dessous pour 32 essais par nombre d'individus parmi N=20, 40, 60, 80. Comme attendu, les résultats sont de meilleure qualité lorsque le nombre d'individu est grand, mais le temps de calcul est doublé lorsque N est doublé.

VI. Visualisation des résultats

Ci-dessous, la visualisation 3D d'une solution pour le *plasmid 8k* :

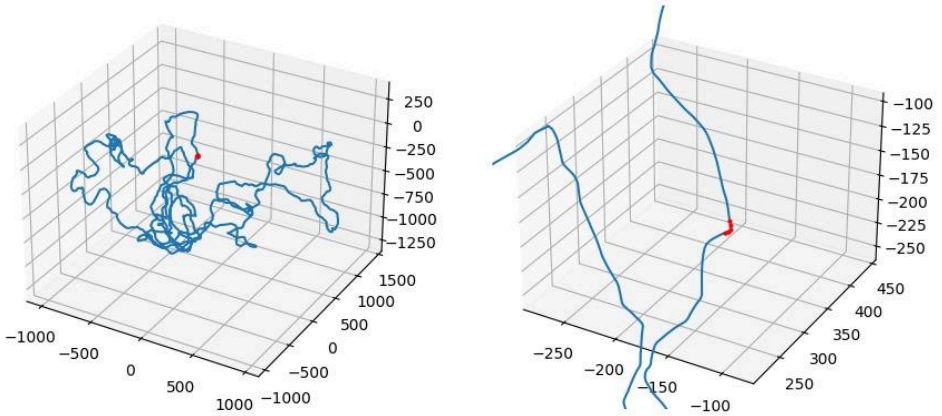


Figure 15 : Trajectoire 3D d'une solution : le bouclage est mis en évidence en rouge

Visuellement, les "bonnes" solutions fournies par les deux algorithmes sont similaires. On obtient un plasmide qui boucle : en effet le début et la fin de la chaîne sont confondus. On voit également que l'angle entre le début et la fin de la chaîne n'est pas trop aigü. On peut dire que ces méthodes permettent donc d'aboutir, au bout de quelques exécutions, à des solutions satisfaisantes.

VII. Comparaison des deux algorithmes

Le tableau ci-dessous compare qualitativement les 2 algorithmes pour le *plasmide 8k*: recuit simulé et algorithme génétique suivant les résultats obtenus :

Critères	**Recuit simulé**	**Algorithme génétique**
Temps de convergence	++	-
Nombre d'itérations à convergence	-	++
Stabilité	++	-
Nombre de valeurs optimal	++	-

Tableau 7 : Comparaison entre les algorithmes recuit simulé et génétique

Les résultats des nombreuses itérations des deux algorithmes permettent d'affirmer que l'algorithme de recuit simulé obtient globalement des résultats plus cohérents par rapport aux contraintes naturelles.

Il est à noter que le temps d'exécution de l'algorithme de recuit simulé est environ 4 fois plus rapide que celui de l'algorithme génétique avec des configurations optimales. Une mémoïsation a été implémentée dans la fonction énergie de l'algorithme génétique afin d'accélérer le calcul du résultat de cette fonction pour un même état. Cependant, l'algorithme génétique nécessite moins d'itérations pour arriver à convergence. La **partie IV** nous montre que la distribution des distances est plus "resserrée" pour le recuit simulé, il est donc plus stable. Le nombre de valeurs optimales, proches d'une distance de 3 ou d'un produit scalaire élevé, est plus grand pour le recuit simulé, là ou les distributions pour la méthode génétique avec sélection par élitisme (qui donne les meilleurs résultats) sont bien plus étalées et autour de valeur moins bonnes.

Finalement, l'algorithme de recuit simulé semble être meilleur que l'algorithme génétique pour nos simulations sur le *plasmid_8k*. Peut être que l'algorithme génétique serait plus performant sur un problème plus complexe tel que le *plasmid_180k*.

VIII. Tests de couverture

Les tests unitaires et la couverture de code sont des éléments importants, permettant de vérifier la qualité et la fiabilité du code produit. Ils garantissent le bon déroulé des fonctions du code, et facilitent la correction des problèmes. En parallèle, la couverture de code offre une visibilité sur l'étendue des tests réalisés, identifiant les zones du code qui ne subissent pas de test. De manière générale, ces pratiques renforcent la robustesse du code et facilitent la maintenance continue du projet.

Les deux scripts implémentés sont *recuit.py* et *genetique.py*. Il s'agit donc d'écrire deux scripts *test_recuit.py* et test*_genetique.py* qui consistent en une suite de fonctions qui appellent toutes les fonctions du script à tester, en vérifiant par une assertion que le résultat obtenu est bien celui attendu. Par ailleurs, il s'agit de vérifier la couverture du code, c'est-à-dire si la totalité du code implémenté est bien exécutée lors des tests unitaires.

Pour garantir une bonne reproductibilité des tests, ils ont été réalisés avec une seed : random.seed(91).

Un exemple de fonction réalisant un test unitaire :

Python

```python
from RotTable import RotTable
from Traj3D import Traj3D
from recuit import Recuit
from random import seed

# Seed pour la reproductibilité des résultats de test
seed(91)

def test_energie():
    # Initialisation des paramètres
    seq=''.join([line.rstrip('\n') for line in
open(r'..\data\plasmid_8k.fasta')][1:])
    k_max=1
    e_min=1
    temp_init=1
    refroidissement=1
    dist_min=1

    # Test n°1
    test_recuit=Recuit(seq, k_max, e_min, temp_init,
refroidissement,dist_min)
    assert test_recuit.energie(RotTable())==(5987.480078776554,
5990.480078776554)
```

Les tests de couverture peuvent être réalisés et visualisés dans un rapport nommé *index.html* dans le dossier **htmlcov** avec la commande suivante à écrire dans le terminal:

```
Unset
python -m pytest --cov=. --cov-report html
```

En particulier, le test de couverture pour les scripts *recuit.py* et *genetique.py* donne les résultats suivants:

Module	statements	missing	excluded	coverage ↑
genetique.py	176	0	0	100%
test_genetique.py	48	0	0	100%
recuit.py	79	0	0	100%
test_recuit.py	22	0	0	100%

Figure 16 : Couverture des tests

Outre le fait de passer les tests, une couverture de 100% est obtenue. Toutes les lignes de codes sont donc soumises aux tests.

La couverture du code par fonction test est donc très bonne : c'est essentiel pour avoir un code solide. Si le code est repris plus tard, cette couverture permettra de vérifier que les fonctions de bases ne sont pas altérées.

IX. Gestion du travail

1. Découpage en sous-tâches

Nous nous sommes répartis les rôles au mieux pendant cette semaine qui est passée très vite.

Phase 1	**Phase 2**	**Phase 3**
Hiyu & Guillaume : Algo de recuit **Ali & Erwan :** Algo génétique	**Hiyu & Erwan :** Optimisation des algos **Ali & Guillaume :** Test unitaires et algos de statistiques	**Tout le monde :** Soutenance et rapport

Bien sûr, cette répartition n'est que théorique. En pratique nous échangions régulièrement afin de partager des idées ou apporter des modifications.

2. Carnet de bord

Voici le carnet de bord de notre avancement sur ce projet :

Jour 1 - Lundi	Cours théoriques - Création Git - Séparation des rôles - Code du recuit
Jour 2 - Mardi	Finalisation du recuit - Mise en place de l'algorithme génétique
Jour 3 - Mercredi	Finalisation de l'algorithme génétique - Redéfinition de la fonction objectif
Jour 4 - Jeudi	Finalisation du code - Rédaction du rapport
Jour 5 - Vendredi	Relecture du rapport - Répétition pour l'oral

Tableau 8 : Carnet de bord

X. Conclusion

Ce projet a permis d'optimiser un modèle de conformation 3D connu, permettant de convertir une suite de nucléotides en une trajectoire tridimensionnelle, afin de rendre un plasmide circulaire.
Il est à noter que les résultats de ce rapport ont été obtenus à partir du *Plasmid 8k*. Le *Plasmid 180k* n'a pas été traité dans les études statistiques car la taille de la séquence était trop importante pour nos machines.

Les résultats ainsi obtenus permettent de coller au mieux à la réalité : le plasmide est circulaire et la jonction est géométriquement correcte.
Il faut garder en tête que les 2 algorithmes utilisés restent des heuristiques. Ils ne donnent donc pas la solution optimale mais une solution approchée.

Ces 2 algorithmes ont des temps de convergence différents, et il semblerait que le recuit simulé soit suffisamment performant vis à vis de notre problème qui reste "relativement" simple. L'avantage de l'algorithme génétique est l'exploration de l'univers des solutions : elle est bien plus grande qu'avec le recuit de par le grand nombre d'individus considéré par génération.

Ce projet nous a aussi permis de prendre en main l'algorithme de recuit simulé et le concept des algorithmes génétiques, méthodes métaheuristiques fréquemment utilisées pour trouver une solution à un problème complexe.

XI. Ressources externes et annexes

Ce projet a nécessité l'utilisation de la documentation en ligne de :

- *Random* : https://docs.python.org/3/library/random.html
- *Argparse* : https://docs.python.org/3/library/argparse.html
- Chat GPT, uniquement pour des documentations sur certaines fonctions et bibliothèques Python : https://chat.openai.com/
- Documentation externes diverse sur le fonctionnement des algorithmes de recuit simulé et algorithmes génétiques