ECOLE NATIONALE POLYTECHNIQUE

COMPLÉMENT DE FORMATION MASTER

Algorithme génétique

APPLICATION D'UN ALGORITHME GÉNÉTIQUE POUR LA MINIMISATION D'UN CRITÈRE QUADRATIQUE ET LE CONTRÔLE D'UN DRONE

MEGHNOUDJ Houssem
houssem.meghnoudj@g.enp.edu.dz

NECHAT Ghiles
ghiles.nechat@g.enp.edu.dz

Département d'Automatique

5 février 2020

Table des matières

1 Introduction **4**

2 Définitions **4**
2.1 Gène 4
2.2 Individu 5
2.3 Population 5
2.4 Utilité (Fitness) 6

3 Principe **7**
3.1 Initialisation 7
3.2 Sélection 8
3.3 Pairing 9
3.4 Mating 10
3.5 Mutation 11
3.6 Critère d'arrêt 11

4 Exemple explicatif **12**

5 Convergence théorique **15**
5.1 La Théorie des schémas 15
5.2 Équations de convergence 16

6 D'autres types **17**
6.1 Sélection 17
6.2 Pairing 18
6.3 Croisement 18
6.4 Mutation 18
6.5 Critère d'arrêt 19

7 Exemples particuliers **20**
7.1 Trouver le message 20
7.1.1 La taille de population 20
7.1.2 Le pourcentage de mutation 20
7.2 Maximum d'une fonction 20
7.2.1 Codage des individus 20
7.3 Piles de cartes 21

8 Application **22**
8.1 Détails sur le code 22
8.2 Simulation 24

9 Référence **26**

10 Annexe **27**

Table des figures

1 Représentation d'un Gène. 4
2 Représentation d'un individu de longueur 8 5
3 Représentation d'une population de taille 8 5
4 Individu de taille 6 7
5 Population de 6 individus. 7
6 Calcul de la fitness de notre population. 8
7 Après classement de nos individus connaissant leur fitness. 8
8 Sélection de nos meilleur individus suivant leur fitness. 9
9 Les individus sélectionné après avoir été jumelés. 9
10 Illustration du point de croisement. 10
11 Création de nouveaux individus. 10
12 Illustration sur la mutation d'individu. 11
13 Déroulement de l'algorithme. 11
14 Population initiale. 12
15 Choix des parents 1 et 2. 12
16 Croisement des parents (à gauche) donne l'enfant (à droite). 13
17 Mutation de l'enfant (à gauche) donne l'enfant (à droite). 13
18 Population finale. 14
19 Effet du maximum fitness sur la solution. 19
20 Réponse en position z (haut) Thrust (bas). 24
21 Réponse en angle ϕ, θ, ψ (haut), signaux de commande des angles (bas). 25

Liste des Symboles

$\delta(H)$ Longueur fondamentale d'un schéma H.

A Chromosome ou individu.

a_i i^{me} gène du chromosome A.

d fonction de décodage passant du chromosome A à une valeur réel.

H Schéma.

l Taille du chromosome

$o(H)$ Ordre d'un schéma.

Pi Personne ou individu i de la population.

tol tolérance.

$U(A)$ Utilité du chromosome A.

1 Introduction

Les algorithmes évolutionnistes (evolutionary algorithms en anglais), sont la famille d'algorithmes qui s'inspirent de la théorie de l'évolution (méthodes de calcul bioinspirées) pour résoudre des problèmes divers. En utilisant itérativement des processus aléatoires pour faire évoluer un ensemble de solutions à un problème donné vers de meilleurs résultats.
La grande majorité de ces méthodes sont utilisées pour résoudre des problèmes d'optimisation, elles sont en cela des métaheuristiques. Ils ont la propriété de localiser plusieurs optima, mais peuvent s'avérer chères en calculs et leur mise en oeuvre pratique demande une bonne expérience.
On distingue quatre grandes familles dont celle des algorithmes génétiques (AGs).

Les algorithmes génétiques sont des algorithmes d'optimisation stochastique fondés sur les mécanismes de la sélection naturelle et de la génétique, apparus aux États-Unis dans les années 60 : les premiers travaux de John Holland sur les systèmes adaptatifs remontent à 1962, et c'est au livre de David Goldberg (1989) que l'on doit leur popularisation. Le fonctionnement est simple, en partant d'une population de solutions potentielles initiales arbitrairement choisies et qui joueront le rôle de chromosomes, on évalue leur performance relative (utilité), et sur la base de ces performance on crée une nouvelle population de solutions potentielles en utilisant des opérateurs évolutionnaires simples (sélection, croisement, mutation), qui sont censés apporter une amélioration au performances. On recommence ce cycle jusqu'à ce que l'on trouve une solution satisfaisante.

— Les algorithmes génétiques travaillent sur une population de points, au lieu d'un point unique.
— Les algorithmes génétiques n'utilisent que les valeurs de la fonction étudiée (fonction utilité), pas sa dérivée, ou une autre connaissance auxiliaire.

2 Définitions

Pour les algorithmes génétiques, un des facteurs les plus importants est le codage, la façon dont sont codées les solutions, c'est-à-dire les structures de données qui coderont les gènes. Admettons l'hypothèse du codage binaire :

2.1 Gène

Le Gène sera alors une valeur de l'ensemble $\{0, 1\}$, c'est le matériau de base qui permettera la construction de chaque individu, et qui, selon le problème traité, donne une certaine caractéristique à cette entité.

Figure 1 – Représentation d'un Gène.

2.2 Individu

L'Individu c'est une solution potentielle du problème, et sera une séquence de gènes, équivaut à une suite de bits en codage binaire.
Nous appelons une séquence (chromosome) A de longueur $l(A)$ une suite $A = \{a_1, a_2, ..., a_l\}$ avec $a_i \in V = \{0, 1\}$.
Dans le cas d'un codage non binaire, tel que le codage réel, la suite A ne contient qu'un point, nous avons $A = \{a\}$ avec $a \in \mathbb{R}$.

Figure 2 – Représentation d'un individu de longueur 8

2.3 Population

La Population est un ensemble d'individus de l'espace de recherche, sur lequels s'appliqueront les étapes de la boucle d'évolution, qui permettrons au générations nouvelles d'explorer l'espace tout en convergeant vers les séquences les plus performantes. En binaire, c'est un ensemble de séquences binaires de même longueur l, le cardinal de cet ensemble sera la taille de la population.

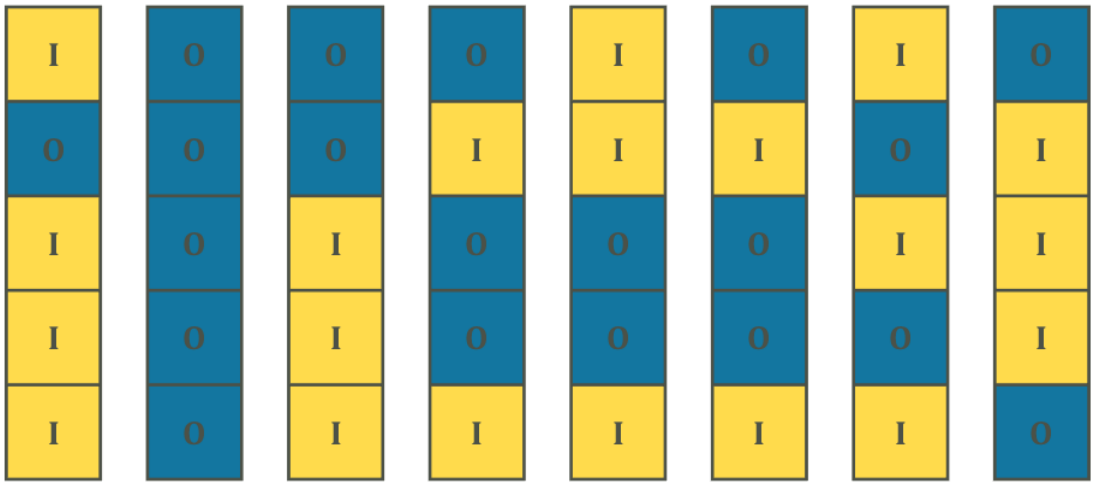

Figure 3 – Représentation d'une population de taille 8

2.4 Utilité (Fitness)

L'utilité (Fitness en anglais) $U(A)$, à valeurs positives, permet de calculer l'adaptation d'une séquence de bits. Idéalement, elle vaudra 1 si la séquence est parfaitement adaptée, et 0 si elle est inadaptée.
Cette utilité est donc donnée par une fonction à valeurs positives réelles. Dans le cas d'un codage binaire, nous utiliserons souvent une fonction de décodage qui permettra de passer d'une chaîne binaire à un chiffre à valeur réelle $d : \{0,1\}^l \rightarrow \mathbb{R}$ (où l est la longueur de la chaîne). La fonction de fitness est alors choisie telle qu'elle transforme cette valeur en valeur positive, soit $f : d(\{0,1\}^l) \rightarrow \mathbb{R}^+$. Le but d'un algorithme génétique est alors simplement de trouver la chaîne qui maximise cette fonction f . Bien évidemment, chaque problème particulier nécessitera ses propres fonctions d et f.

3 Principe

Pour illustrer le principe de l'algorithme nous avons pris un exemple simple, une population d'individu et ces derniers ont 6 gènes (6 paramètres) et la valeurs de ses paramètres est binaire $\{0, 1\}$. La fitness function est simplement la somme de chaque valeur du gène de l'individu (autrement dit combien de 1 il y a dans l'individu)

Figure 4 – Individu de taille 6

Le but de l'algorithme est de trouver l'individu qui maximise cette fintess function c'est à dire l'individu [111111] qui a une fitness de 6

3.1 Initialisation

C'est la partie où la création de notre population a lieu, on commence par créer nos individus de manière complètement aléatoire ou contrôlé tout dépends de ce qu'on cherche et de nos connaissance sur le problème, une fois qu'un individu est créer on le met avec l'ensemble de la population jusqu'à arriver au nombre d'individu dans notre population souhaité.

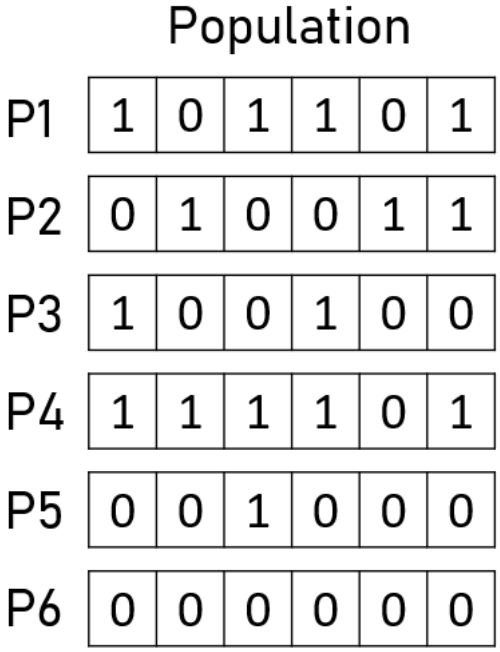

Figure 5 – Population de 6 individus.

R.Q : pour l'application au drone nous avons supposé que les angles son découplé donc chaque ligne de la matrice gain agit seulement sur l'angle et la vitesse d'un angles précis et non tous le vecteur d'état qui agit sur la commande d'un seul angle.
Cette connaissance apriori permet lors de la création des individus et de la population de garder des paramètres à zéro. Ayant un peu plus de connaissance ça permet même de connaitre l'intervalle de variation des paramètres (ça facilite la recherche).

3.2 Sélection

Une fois que nous avons nos individu (créer récemment ou bien c'est le résultats de plusieurs générations), On a besoin d'un moyen de noter les individu ou de les classer et pour cela on utilise la fitness function. c'est la fonction ayant l'individu elle évalue à quel point cet individu (solution) est bonne ou pas.

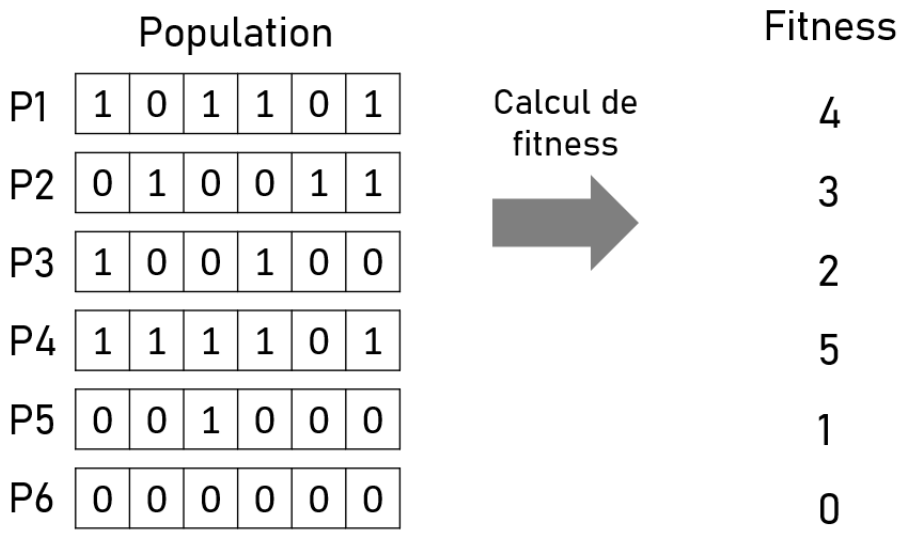

Figure 6 – Calcul de la fitness de notre population.

Une fois qu'on a évalué chaque individu et pris en compte leur fitness function on va les classer (croissant ou décroissant) suivant notre besoin (minimiser ou maximiser).

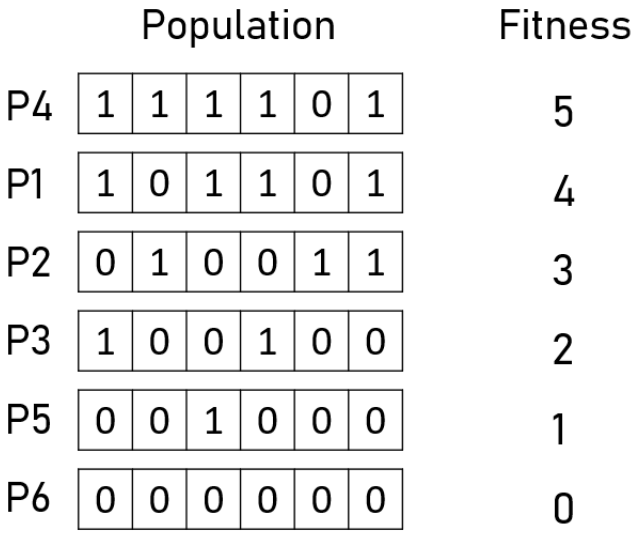

Figure 7 – Après classement de nos individus connaissant leur fitness.

Pour cette étape on va prendre une méthode simple de sélection pour comprendre l'idée, et dans la section *6.autres type* et *8.application* on verra d'autre méthodes meilleur et offrant plus de liberté et plus performantes.
La sélection consiste à prendre les meilleurs individu (ceux qui ont la plus grande fitness si on

cherche à maximiser la fonction objectif ou l'inverse dans le cas contraire) et comme pour notre exemple on cherche à maximiser la fonction objectifs on va prendre les 4 premiers.

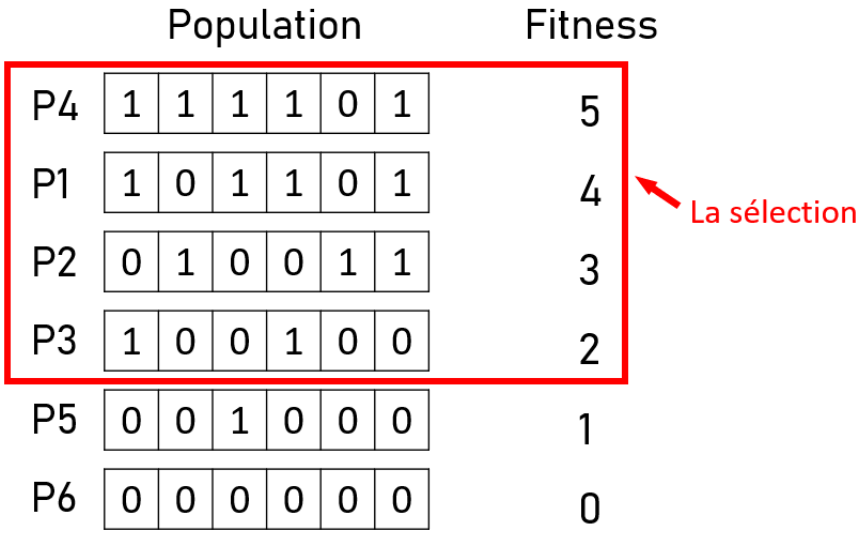

Figure 8 – Sélection de nos meilleur individus suivant leur fitness.

En ce qui concerne le nombre d'individu à prendre, c'est un hyper-paramètre et c'est à l'utilisateur de le déterminer (souvent par trial and error), plus de détails dans la section *7.Effet des paramètres*.

3.3 Pairing

Une fois la sélection faite, on a un certain nombre d'individu qui sont amené à se reproduire mais une question se pose : qui se reproduit avec qui ? C'est le rôle du pairing de déterminer cela. Le pairing c'est comment on choisi à partir des individu sélectionnés les couples d'individu ? Pour la simplicité de l'exemple on prendre les deux premiers (ayant la fitness la plus grande) ensuite les 2 autres qui sont moins bon et ainsi de suite, plus de détails dans la partie *6.autres type*.
Les individus qui vont se reproduire sont appelés parents.

Figure 9 – Les individus sélectionné après avoir été jumelés.

3.4 Mating

Une fois que nous avons notre ensemble de couples, une manière simple de créer des enfants c'est le (cross point). Un point est choisi aléatoirement et les parents sont obligé de s'échanger les gènes qui sont à droite de ce point pour créer de nouveau enfants.

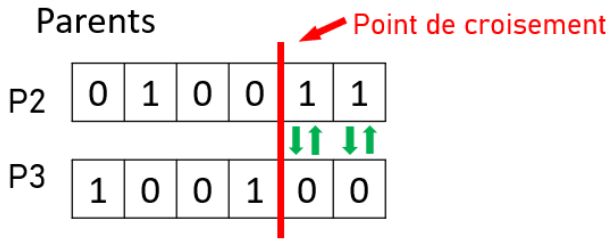

Figure 10 – Illustration du point de croisement.

Comme illustré dans l'exemple cela peut donner des enfants qui sont mieux ou pire que les parents (c'est l'évolution).
Mais comme nous à chaque fois on prends les meilleurs individu pour créer la nouvelle génération les mauvais individu se verront disparaitre avec les générations (c'est la sélection naturelle)

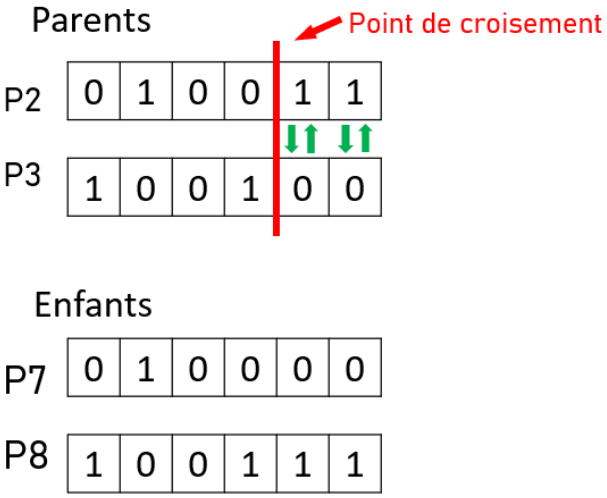

Figure 11 – Création de nouveaux individus.

Une fois que nous avons nos enfants, il suffit de les injecter dans la population en enlevant les mauvais individu (ayant la fitness la plus basse pour notre exemple)
R.Q : on injecte les enfants qu'ils soient bon ou mauvais (s'il sont bon ils vont perdurer, ou dans le cas contraire ils vont mourir), et avec cette sélection du meilleur avec le temps on commencera à se rapprocher de la solution optimale.

3.5 Mutation

La mutation est un phénomène naturel, qui a permis à l'homme d'avancer génétiquement et de devenir mieux (ou pire créer des maladies etc), c'est le même phénomène observé dans la nature qu'on retrouve dans les algorithmes génétique (vu que l'humain s'inspire de la nature). Chaque gène à une petite probabilité de muté (passer de 0 à 1 ou l'inverse)

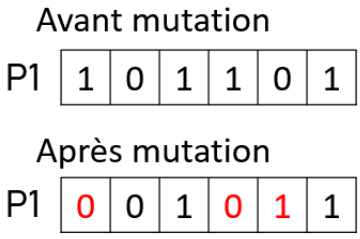

Figure 12 – Illustration sur la mutation d'individu.

Avec la mutation on peut voir l'apparition de nouveaux individus qui sont meilleur ou pire que les anciens (comme vous pouvez le remarquer sur l'exemple)

3.6 Critère d'arrêt

En ce qui concerne les critères d'arrêt on peut en avoir plusieurs, le plus basique c'est qu'après n génération on arrête le processus et c'est à l'utilisation de définir n.
Si le critère d'arrêt n'est pas satisfait, l'algorithme continue à boucler depuis la sélection, plus de détails dans l'illustration suivante.

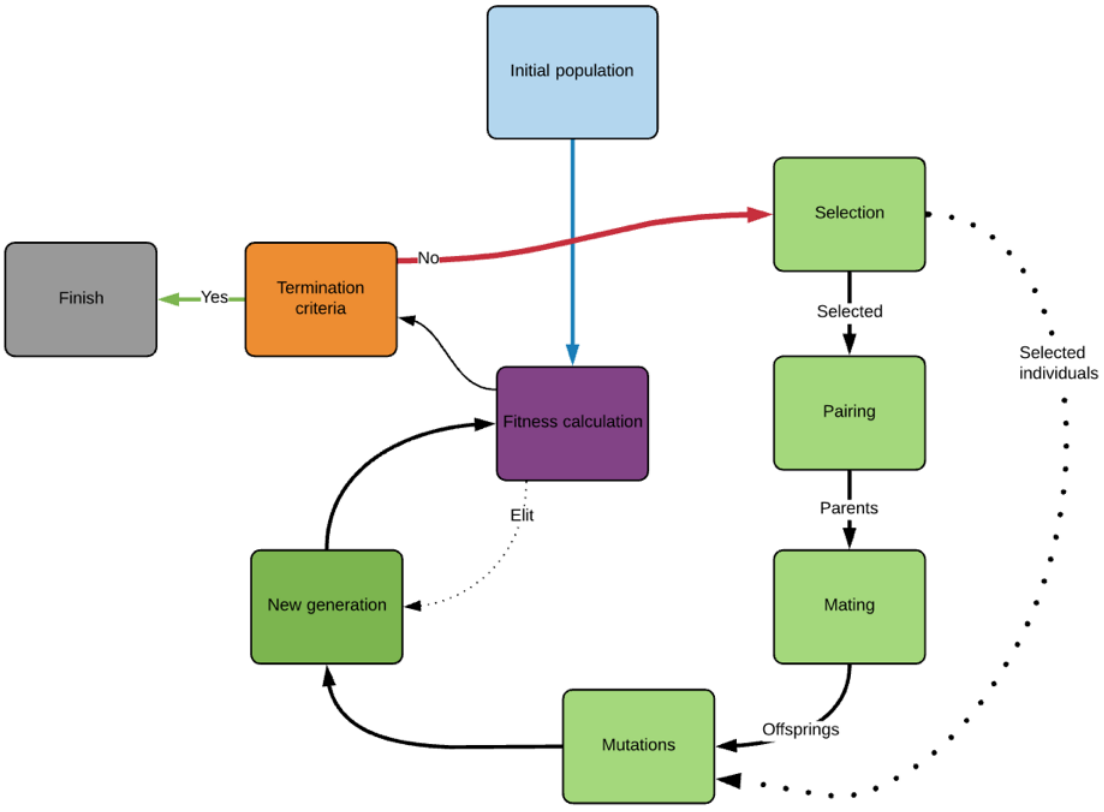

Figure 13 – Déroulement de l'algorithme.

4 Exemple explicatif

Prenons le cas d'une maximisation uni-dimensionnelle où l'on cherche l'entrée optimale de la fonction $f(x) = 4x(1-x)$ dans un intervalle qui se limite à $[0, 1[$. Supposons un codage binaire en 8-bit, et une fonction fitness égale à $f(x)$.
La population initiale est choisie aléatoirement et avec une taille de 5 individus, on est prêts à exécuter la boucle d'évolution.

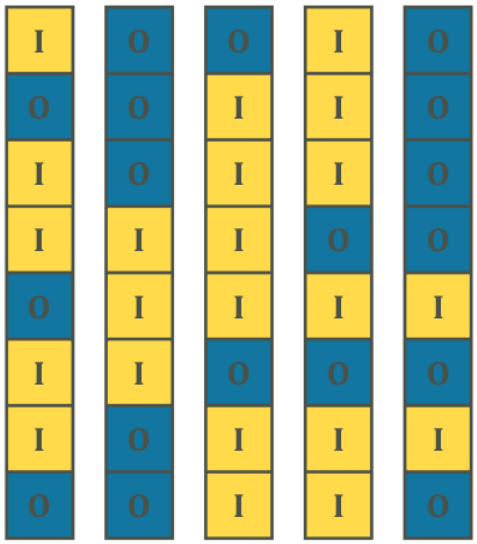

Figure 14 – Population initiale.

L'étape de reproduction consiste à choisir les parents de la nouvelle génération ce qui se fera de façon aléatoire pondérée, c'est-à-dire avec les individus les plus utiles ayant plus de chance d'être choisis, l'algorithme permettant la réalisation de cette étape consiste à calculer les utilités de chaque individu puis de faire une somme cumulative suivie d'une mise à l'échelle entre zéro et un, ceci à pour effet de diviser l'intervale en n parties proportionnelles à leur fitness respéctive. Cette méthode permet de créer deux ensembles de parents, que l'on peut appairer directement des deux premiers au deux derniers.

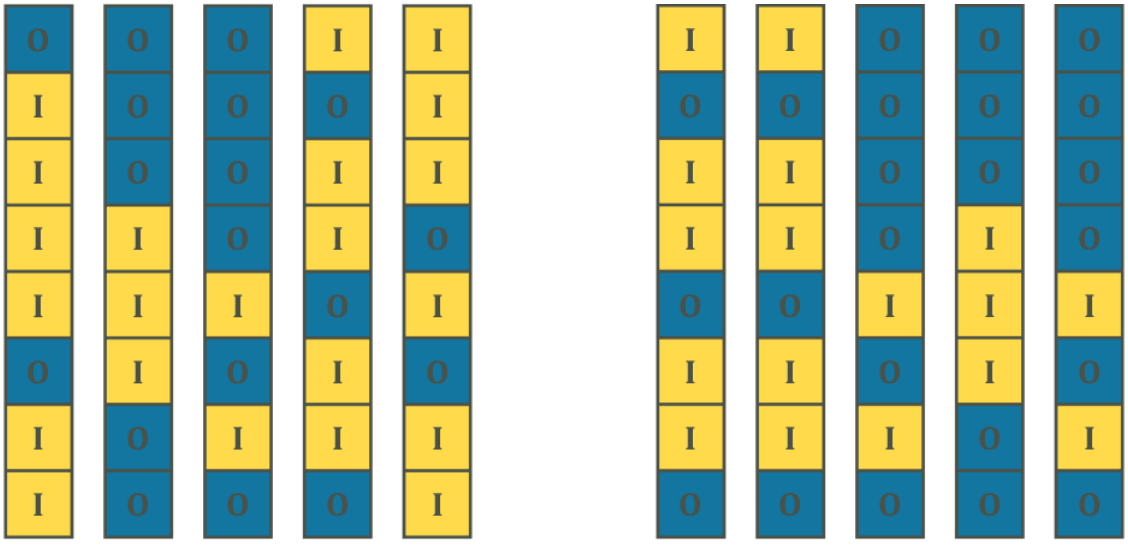

Figure 15 – Choix des parents 1 et 2.

L'étape de croisement consiste en la création de deux individus enfants qui se partagerons de façon aléatoire la gènes de leurs deux parents, l'algorithme qui réalise cette étape crée une séquence binaire avec pour chaque bit une probabilité 50%/50% d'être vrai ou faux, cette séquance servira à choisir les gènes que l'enfant hérite du premier parent et le reste du second parent.

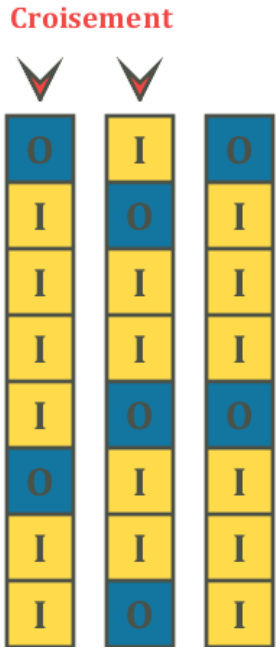

Figure 16 – Croisement des parents (à gauche) donne l'enfant (à droite).

L'étape de mutation consiste en le basculement de certain gènes vers d'autres valeurs, évènement qui s'avère se produire que rarement. l'algorithme qui réalise cette étape génère une séquence binaire pour chaque individu avec une probabilité de 5%/95% d'être vrai ou faux, cette séquence désignera les gènes qui seront mutés, équivaut à une inversion dans le cas binaire.

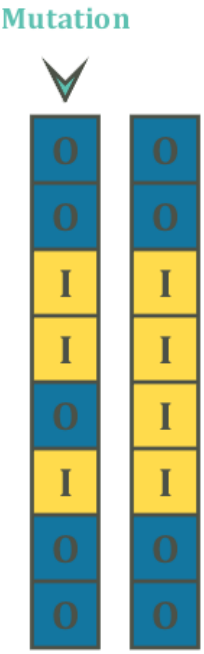

Figure 17 – Mutation de l'enfant (à gauche) donne l'enfant (à droite).

A ce stade les enfants auront assez muri pour êtres considéré comme une nouvelle population, la nouvelle génération existe, et avec elle de nouvelles valeurs pour la fonction d'utilité et un nouvel individu dominant qui donne les meilleurs performances possibles à ce stade. L'algorithme génère cette population et calcule ses caractéristiques.
Dans cet exemple, il est possible d'atteindre le maximum de fitness ($f(x) = 1$), et donc, un critère d'arrêt adéquat serait de se fixer une tolérance assez faible (pourquoi pas nulle), et d'attendre que l'on atteigne $f(x) = 1 - tol$;

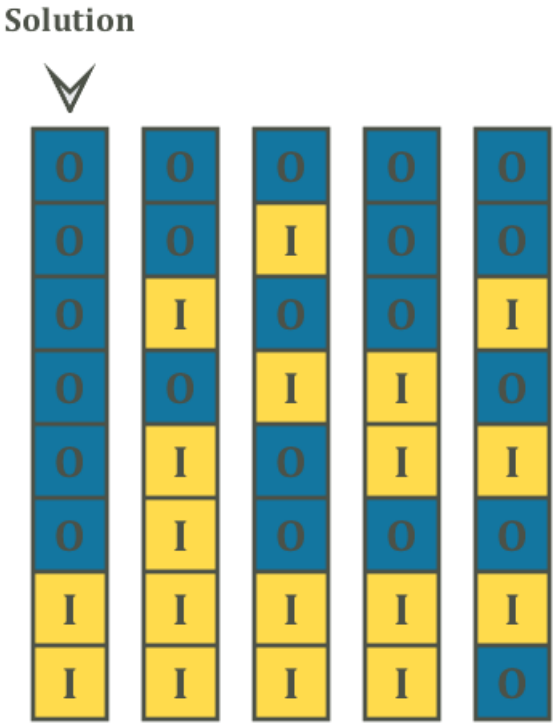

Figure 18 – Population finale.

5 Convergence théorique

Plusieurs approches ont été envisagées. Nous aborderons le cas le plus simple et qui s'applique à l'hypothèse formulée plus haut où le codage se fait sur une séquence binaire. On sera alors capables de donner dans un cadre simplifié une idée du mécanisme de convergence des AGs.

5.1 La Théorie des schémas

Séquence

On appel séquence A de longueur l une suite $A = a_1a_2...a_l$ tel que $\forall i \in \{1, ..., l\}, a_i \in \{0, 1\}$ Ceci correspond à la notion d'individu que nous avons utilisé jusqu'à présent.

Schéma

On appel schéma H de longueur l une suite $H = h_1h_2...h_l$ tel que $\forall i \in \{1, ..., l\}, h_i \in \{0, 1, *\}$ Comme nous allons le voir par la suite, une $*$ en position i veut dire que h_i peut être indifféremment zéro ou un.

Instance

On dit qu'une séquence $A = a_1a_2...a_l$ est une instance d'un schéma $H = h_1h_2...h_l$ si $\forall i \in \{1, ..., l\}, h_i \neq * \rightarrow h_i = a_i$ Ainsi, le schéma $H = 010 * 0$ admet pour instance 01000 et 01010 (les deux seules instances possibles).
La position i de H pour laquelle $h_i = *$ est dite position libre, sinon elle sera position fixe ($h_i = 0$ ou $h_i = 1$).

On définit l'ordre d'un schéma $o(H)$ comme étant le nombre de positions fixes de H, on en déduit que le nombres d'instances différentes admises par un schéma H est de $2^{l(H)-o(H)}$.
On définit aussi la longueur fondamentale d'un schéma $\delta(H)$ comme étant la distance séparant la première et la dernière position fixe de H.
Tel que : $H = 1 * *10 * 10$ admet $o(H) = 1 + 2 + 2 = 5$ et $\delta(H) = 7 - 1 = 6$

Adaptation d'un schéma

L'adaptation est ce qu'on a appelé jusqu'à maintenant la fonction utilité, dans le cas d'un schéma et non d'une séquence cette valeur est obtenue à partir des utilités de ses instances (La moyenne des adaptations de ses instances) $U(H) = \frac{\sum_{i=1}^{2^{l(H)-o(H)}} U(A_i)}{2^{l(H)-o(H)}}$ avec A_i étant l'ensemble de toutes les instances de H.

Soit un ensemble $S = A_1, ..., A_i, ..., A_n$ de séquences de bits tirées aléatoirement, cet ensemble jouera le rôle d'une population de taille n en début de boucle.
Posons la moyenne d'adaptation des séquences : $\bar{f} = \frac{\sum_{i=1}^{n} f(Ai)}{n}$

5.2 Équations de convergence

Effet Reproduction

Prenons le cas où A_i se reproduit avec une probabilité :

$$p_i = \frac{U(A_i)}{\sum_{i=1}^{n} U(A_i)}$$

Supposons qu'à un instant t nous ayons $m_H(t)$ d'instances représentant le schéma H dans la population, à l'instant $t+1$ nous avons (statistiquement) avoir un nombre :
$m_H(t+1) = \frac{m_H(t) f(H)}{\bar{f}_t}$, posons : $a_t(H) = \frac{f(H)}{\bar{f}_t} \Rightarrow m_H(t+1) = a_t(H) m_H(t)$
Un schéma dont l'adaptation est au-dessus de la moyenne voit son nombre de représentants augmenter, suivant une croissance de type géométrique, si nous faisons l'approximation que $a_t(H)$ est constante nous obtenons bien : $m_H(t+1) = a^t(H) m_H(0)$
Cette première équation montre que si la reproduction seule était en jeu, les schémas forts élimineraient rapidement les schémas faibles.

Effet Croisement

Il y a ce qu'on appel, une probabilité de survie $p_s(H)$ d'un schéma H, qui apparait lors d'une opération de croisement, dans ce cas on choisit un site de croisement et on prend la première partie du premier parent et la deuxième partie du second parent. Prenons par exemple $H = **01*11*$, on peut être sûr que si le site de croisement est inférieur à 3 (avant la première position fixe) ou supérieur à 6 (après la dernière valeur fixe), les séquences croisées ne détruiront aucune instance de H. On peut alors définir une borne inférieure de la survie qui dépendra de la longueur fondamentale $\delta(H)$.

On voit donc qu'une borne inférieure de la probabilité de détruire un schéma H est $\frac{\delta(H)}{l-1}$. La probabilité de survie dans un croisement est alors $1 - \frac{\delta(H)}{l-1}$. Si d'autre part on ne croise qu'une fraction p_c de séquences dans une population donnée, la probabilité de survie est donnée par : $p_s \geq 1 - p_c \frac{\delta(H)}{l-1}$

Effet Mutation

Supposons que la probabilité de mutation d'une position dans une séquence soit p_m. Dans un schéma H, seules les positions libres peuvent être détruites. Comme la probabilité de survie d'un bit est $1 - p_m$, la probabilité de survie d'un schéma H contenant $o(H)$ positions fixes est $(1-p_m)^{o(H)}$. La probabilité de mutation étant toujours petite devant 1, un développement limité au premier ordre donne une probabilité de survie de $1 - o(H)p_m$.

En introduisant ces deux derniers termes, nous avons donc l'équation finale :

$$m_H(t+1) \geq m_H(t) a_t(H) \left[1 - p_c \frac{\delta(H)}{l-1} - o(H) p_m\right]$$

Remarques

- les schémas qui ont une longueur fondamentale petite sont plus favorisés que les autres, lors de la génération d'une nouvelle population.
- les schémas qui ont un ordre petit sont plus favorisés que les autres, lors de la génération d'une nouvelle population.
- Ceci enseigne une chose fondamentale pour le codage de données : les schémas qui codent les données « intéressantes » pour le problème doivent avoir un ordre et une longueur fondamentale faibles, alors que les données « sans intérêt » doivent être codées par des schémas qui ont un ordre et une longueur fondamentale élevés.
- Les résultats présentés ci-dessus ne sont qu'une introduction à la théorie des schémas, il existe de nombreux approfondissements. On donne à titre de référence le livre de Goldberg 1989b.

6 D'autres types

Dans cette partie on va détailler d'autres méthodes (non exhaustifs) des méthodes qu'on a vu dans la partie *3.Principe*.

6.1 Sélection

Roulette wheel selection

Dans cette méthode de sélection chaque individu a une chance d'être sélectionné. La chance qu'un individu sois sélectionné dépends de sa fitness. Les individus ayant une grande fitness (ou petit tout dépends de notre problème) ont plus de chance d'être sélectionné.
La méthode de roulette wheel selection prends la somme cumulée pour chaque individu. Après avoir normalisé (si nécessaire) la somme cumulée. Un chiffre aléatoire "*chance*" (entre 0 et 1) est sélectionné. Tous les individus qui ont la somme cumulée supérieur à la valeur tiré au hasard sont sélectionnés.
R.Q : La manière dont se fait sélection du chiffre aléatoire est déterminante pour l'algorithme, on peut par exemple choisir $chance \sim \mathcal{U}(0,1)$ à la fin il se pourrai qu'on sélectionne tous le monde, ou bien seulement l'élite, ou bien la moitié.
Pour l'application au drone on a pris $chance \sim \mathcal{N}(\mu = 0.6,\ \sigma^2 = 0.11)$, ceci permet de prendre environs 40% des individus mais on garde toujours la possibilité que tous le monde peut être sélectionné.

Sélection complètement aléatoire

Dans cette méthode les individus sont sélectionné aléatoirement, avec cette méthode permet de casser l'élitisme qu'on les autres méthodes ce qui ralenti la convergence et diminue la stabilité de l'algorithme au profit de la diversité et l'algorithme n'a plus la tendance à converger vers un optimum local.

Demi sélection

C'est la méthode déjà énoncé dans le principe de base, on prends les 50% de la meilleur population classé par leur fitness.

6.2 Pairing

Aléatoire

Dans cette méthode de pairing une fois que nous avons notre ensemble d'individu sélectionné pour créer une nouvelle progéniture les couples de parents sont créer aléatoirement.
Cette méthode augmente la diversité (donc réduit la convergence vers des optimum locaux) mais l'algorithme prends plus de temps à trouver l'optimum global.
Cette méthode ne touche pas à la stabilité car on a déjà sélectionné les personnes.

Le plus apte

Dans cette méthode les individus sont jumelées deux à deux, en commençant par les plus apte (fittest), c'est la méthode présenté dans les principes de bases. En faisant cela les individus les plus aptes sont jumelées ensemble et les individus les moins aptes sont jumelées ensemble aussi, donc on remarque que la variance de la fitness des enfants augmente.

Aléatoire et pondéré

Dans cette méthode les individus sont jumelée deux par deux, les individus les plus aptes (fittest) ont une plus grande chance d'être mis ensemble.

6.3 Croisement

Un seul point de croisement

C'est la méthode vu dans la partie explication du principe.

Deux points de croisements

La méthode consiste à générer deux nombre aléatoire distincts, ces deux nombres représente l'emplacement des points de croisement ensuite pour créer la nouvelle génération, les parents s'interchangent les gênes qui sont entre les deux points de croisements.
Cette méthode permet d'accélérer la recherche de nouveau individus mais un peu plus gourmande en performance.

6.4 Mutation

Cas binaire

Pour le cas binaire lorsqu'une mutation apparait elle modifie la valeur du gêne le faisant passer de 0 vers 1 ou bien l'inverse.

Cas continu, Gauss

Lorsqu'on utilise des valeurs continus et réel $a \in \mathcal{R}$, l'une des méthodes trouvée pour la mutation c'est qu'on change un peu la valeur de ce gêne suivant une loi normale ayant comme moyenne l'ancienne valeur de a ainsi qu'une variance donnée. $a \sim \mathcal{N}(\mu = a, \sigma^2 = 0.8)$.
C'est la méthode utilisé dans l'application du drone car les valeurs dans la matrice de gain sont réels.

Cas continu, Rénitialisation

Dans cette méthode la mutation d'un gène est faite en tirant un nombre aléatoire dans l'intervalle permis, c'est comme une réinitialisation du gêne (nouvelle création).

6.5 Critère d'arrêt

Maximum fitness

Cette méthode vérifie si l'individu le plus apte (fittest) parmi notre génération satisfait notre critère (au dessus ou en dessous de notre fitness value), cette méthode est très sujet au optimum locaux comme vous pouvez le voir dans la figure suivante.

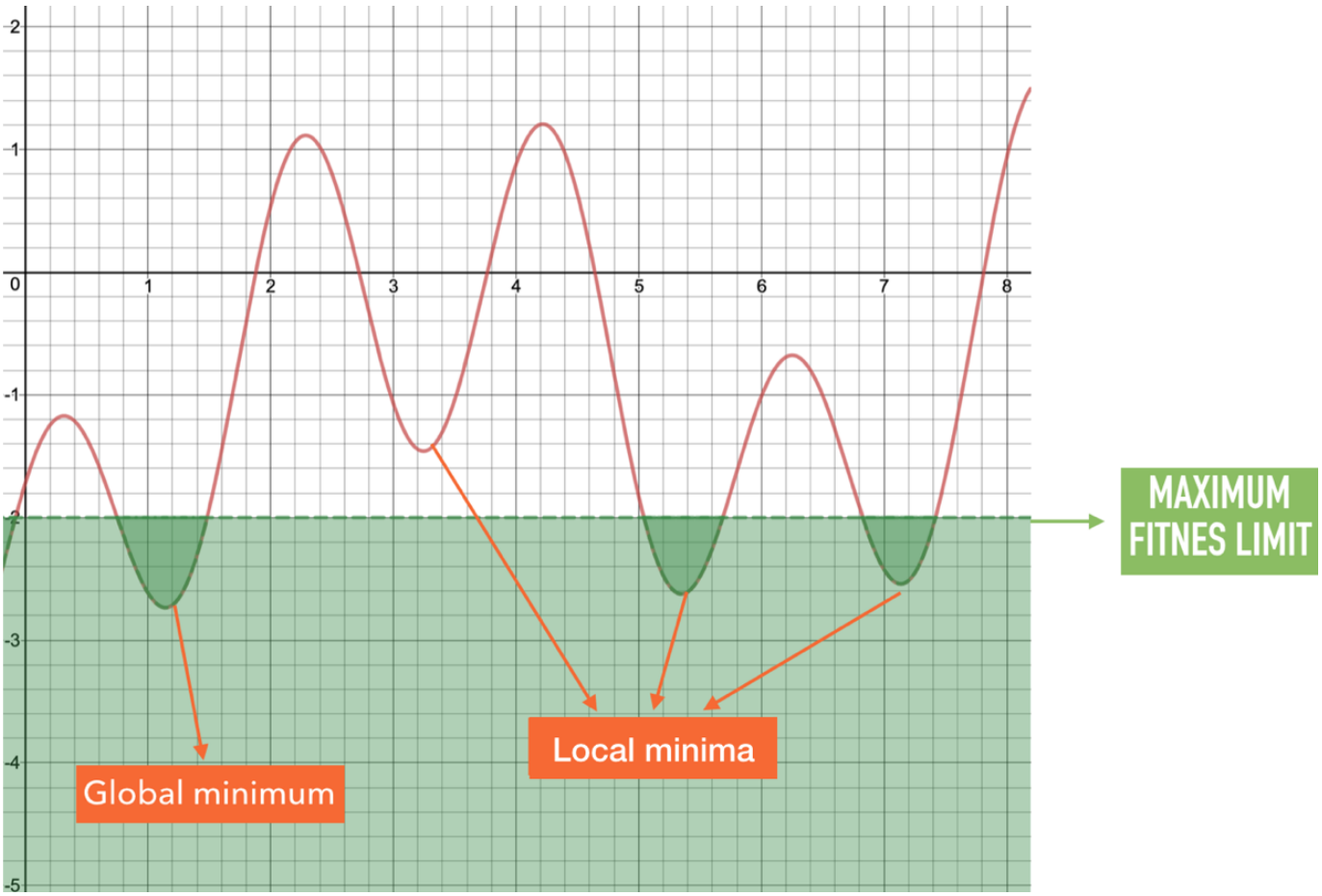

Figure 19 – Effet du maximum fitness sur la solution.

Nombre d'étape

Une fois que nous avons n génération on arrête l'algorithme et ca reste à l'utilisateur de définir n suivant la vitesse de convergence de l'algorithme avec une petite marge de sécurité (pour être sur qu'il converge).

Moyenne max de la fitness

Si notre génération (en totalité) à une fitness au dessus d'un seuil qu'on fixe l'algorithme s'arrête.

7 Exemples particuliers

7.1 Trouver le message

Dans cet exemple le but est de retrouver un message cible qui sera (pour faire simple) "Message Cible.", ce message est codé sur 14 caractères, donc une suite de 14 codes ASCII à retrouver en partant de zéro. Pour compliquer les recherches, on restreint la fonction utilité à la somme des caractères justes qui sera mise à l'échelle sur l'intervalle $[0, 1]$, donc aucun moyen de savoir quel caractère est juste et lequel est faux.
L'exécution du programme avec une population de 400 individus et un pourcentage de mutation égal à 1 atteint la solution en moyenne en 22.24 générations (écart-type de 17.26).
Ce qui est important à observer c'est que, même pour un exemple relativement simple, le pourcentage de mutation et la taille de la population jouent déjà un premier rôle dans le temps d'exécution du programme ou similairement en quantité de calculs, c'est ce à quoi nous nous intéresserons pour la suite.

7.1.1 La taille de population

Après avoir présenté le cas à 400 individus, faisons les mêmes calculs pour une population plus ou moins grande, nous avons donc les statistiques pour 200 individus qui admettent comme moyenne 79.19 générations (écart-type de 96.44), aussi les statistiques pour 800 individus qui admettent comme moyenne 16.62 générations (écart-type de 2.11). On peut se faire une idée sur le poids des calculs en comparant les individus créés lors de l'évolution.
Nous remarquons qu'une population de taille 400 à besoin de créer 8904 individus pour atteindre la perfection. La population de taille 200 à besoin de créer 15838 individus pour atteindre la perfection, plus la population est petite, moins il y a de diversité et plus d'individus sont requis. La population de taille 800 à besoin de créer 13296 individus pour atteindre la perfection, plus la population est grande, moins il y a de dominance et plus d'individus sont requis.

7.1.2 Le pourcentage de mutation

Pour une taille de population faible, la mutation joue à son tour un rôle important dans l'évolution. Un bon choix est très souvent inférieur à 10%, dans ce cas et avec 1% des gènes mutés on aura besoin de 1500 générations, un pourcentage inférieur augmente les étapes d'évolution et on atteint pour 0.5% les 2500 générations, ce qui nous amène au cas exagéré de 5% qui empêche la convergence et ramène la population à des stades non-évolués et on atteint les 10000 générations.

7.2 Maximum d'une fonction

Prenons le cas de la fonction déjà introduite plus haut et où l'on souhaite maximiser $4x(1-x)$, cet exemple met en évidence l'un des deux choix les plus importants dans la création de l'algorithme génétique.

7.2.1 Codage des individus

Si notre individu est un code binaire et que notre but et de trouver un point de l'axe des abscisses qui maximise une fonction, nous devrons impérativement passer par une étape de codage. Le première idée qui nous vient à l'esprit est de coder directement en binaire l'intervalle

$[0, 1[$ sur 8 bits, ce qui s'avère être un trop mauvais choix, en effet, le code de la valeur $\frac{1}{2} = 0.5$ est 10000000 en même temps le code de la valeur $\frac{1}{2} - \frac{1}{256} \simeq 0.4961$ est 01111111, c'est deux codes totalement différents pour des valeurs tout à fait voisines !
Alors pour une fonction ayant son maximum à 0.5 on se retrouve dans la moitié des cas bloqués sur la valeur voisine.
Une solution serait de choisir un codage qui nous évite un tel piège, par exemple dans le cas déjà traité nous avons retrouvé la solution codée en Gray.

7.3 Piles de cartes

Étant donné un ensemble $E = \{1, 2, ..., 10\}$, on veut le partitionner en deux sous-ensembles S et P pour lesquels : $\sum_i s_i = 36$ et $\prod_j p_j = 360$
Pour ce faire on crée une population de 40 individus, pour laquelle chaque individu définit un certain partitionnement de l'ensemble de départ, si le i-ème gène est 0 alors i appartiendra à l'ensemble S (somme), sinon le i-ème gène sera 1 et i appartiendra à l'ensemble P (produit).
Encore un exemple simple et riche, en effet il met en évidence le deuxième choix le plus important lors de la création d'un algorithme génétique, et qui sera la fonction fitness. On peut intuitivement proposer une fonction qui calcule les erreurs puis fait une mise à l'échelle entre 0 et 1 et qui ressemblera à :

$$U(A) = 1 - \frac{1}{2}\left(\frac{|S - 36|}{36} + \frac{|P - 360|}{10\,! - 360}\right)$$

Les résultats convergent mais trop lentement, des centaines voire des milliers de générations sont requises. Ce que nous remarquons pour ce choix c'est que la somme du meilleur individu est maintenu à 36 tandis que son produit approche graduellement la bonne valeur 360, ceci est dû à une mauvaise définition de la fonction utilité qui récompense plus la somme que le produit.
Une meilleur fonction fitness met à l'échelle l'erreur arithmétique de la somme additionnée à l'erreur géométrique du produit comme ceci :

$$U(A) = 1 - \frac{1}{2}\left(\frac{|S - 36|}{36} + \frac{\log(\frac{P}{360})}{\log\frac{10!}{360}}\right)$$

On remarque pour ce cas que les générations convergent aussi bien en somme qu'en produit, ce qui se répercute directement sur le nombre de générations requis pour l'évolution et qui se limite à une ou quelques dizaines générations.

8 Application

8.1 Détails sur le code

Il est a noté que le code est en python (pour des raisons de simplicité) et qu'il est disponible en annexe (je vous invite a le lire il est très lisible).

Pour l'application on a choisi de tester l'algorithme pour la commande en retour d'état d'un quadrirotor en attitude et en position z.

L'entrée de commande du quadrirotor sont les 3 couples suivant *yaw*, *roll* et *pitch* ainsi que le *thrust*.

La fitness value est défini comme pour la commande LQ :

$$fitness(K) = \int_0^{+\infty} (x^T Q x + u^T R u) dt$$

K étant la matrice retour d'état.

Dans notre application un individu représente la matrice gain (du retour d'état), un gène représente un élément de cette matrice. Comme vous pouvez l'apercevoir nos gènes ne sont pas binaire mais ont un valeur réel dans un intervalle permis qu'on a déterminé à partir de nos connaissances.

drone.py

Pour simuler le drone on du créer le modèle du drone (drone.py) dans ce fichier on trouve au début les paramètres du modèle du drone (matrice d'inertie etc) ainsi que des paramètres de simulation (le temps), il est a noté que la résolution de l'équation différentielle d'ordre 1 a été faite avec la méthode d'Euler d'une part car elle n'est pas gourmande en ressources et d'autre part elle nous convient niveau précision.

Les méthodes de la classe sont :

- Step : permet de faire avancer de dt la simulation, la méthode step prends comme paramètre la commande U. Une fois que le nouvel état est calculé elle enregistre dans une mémoire les signaux de commande et les signaux d'état (qui serviront pour le calcul de la fitness par la suite)
- Control : prends comme paramètre la matrice de gain K (individu) et les signaux de commande pour le retour d'état, il est a noté que les signaux de commandes sont saturé comme en pratique.
- Simulate : permet d'itérer sur notre vecteur temps.
- Show : pour l'affichage des signaux de commande et du vecteur d'état.
- Reset_memory : comme on a la déjà dit les données de simulation sont enregistrer dans une mémoire (pour le calcul de la fitness) mais une fois qu'un individu (un matrice K) a été simulé c'est la fonction reset_memory qui s'occupe de reset le drone en position initiale et d'effacer la mémoire du drone permettant à un autre drone d'être simulé sans que les données se croisent.

genetic_algorithm.py

C'est le code qui gère l'algorithme génétique, au début on peut voir la définition de la matrice Q et R. Pour des raisons de simplicité et de scallability du code on a choisi de créer la classe genetic_algorithm qui possède plusieurs méthodes.

- Individual : permet de créer un individu avec les paramètres dans l'intervalle permis et la forme souhaité (on a supposé un découplage entre les angles donc il y a des 0 dans la matrice.
- Init_pop : permet d'initialiser la population, il est a noté que la population (toutes les matrices de gain K) est enregistré dans l'objet ga.
- Individual_fitness et calculate_fitness : permettent de calculer la fitness. Après avoir simulé les performances de chaque individu sur le drone, les données de simulation sont stocké dans une grande mémoire. Ces deux méthodes calculent avec un intégration trapèze la fitness (LQ) de tous nos individus puis ordonne la population.
- Sélection : c'est l'implémentation de roulette wheel selection, à la sortie de la méthode on les individus sélectionné.
- Mating : permet de créer les enfants de la nouvelle génération connaissant la sélection, à la fin elle ajoute les nouveaux enfants et enlève les mauvais parents.

- Mutation : permet la mutation des gènes (utilisation d'une loi normale), les deux meilleurs parents ne subissent pas la mutation (dans le but de donner une certaine stabilité à l'algorithme).

main.py

Dans le main on crée notre drone pour la simulation ainsi qu'un objet de l'algorithme génétique, dès la création de (ga) la population s'initialise automatiquement.
Le main simule tous les individus sur le drone et enregistre dans une grande mémoire toutes les données en tenant compte qu'il faut reset la mémoire du drone a chaque fois qu'on change d'individu. Une fois la simulation sur le drone de tous les individus d'une génération est terminé, on fait passe la mémoire contenant le résultat que chaque individu a fait sur le drone (vecteur d'état et signaux de commande) à l'algorithme (ga).
(ga) applique les méthodes déjà expliqué dans le document.

8.2 Simulation

Il est a noté que lors de la simulation on a placé le drone dans une position initiale puis on l'a laissé retourner à la position $[\phi, \theta, \psi, z] = [0, 0, 0, 0]$
Après exécution avec 100 individus et 300 itérations on trouve la matrice gain suivante :

$$K = \begin{bmatrix} 3 & 0 & 0 & 0.57 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0.42 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -91.23 & -16.4 \end{bmatrix}$$

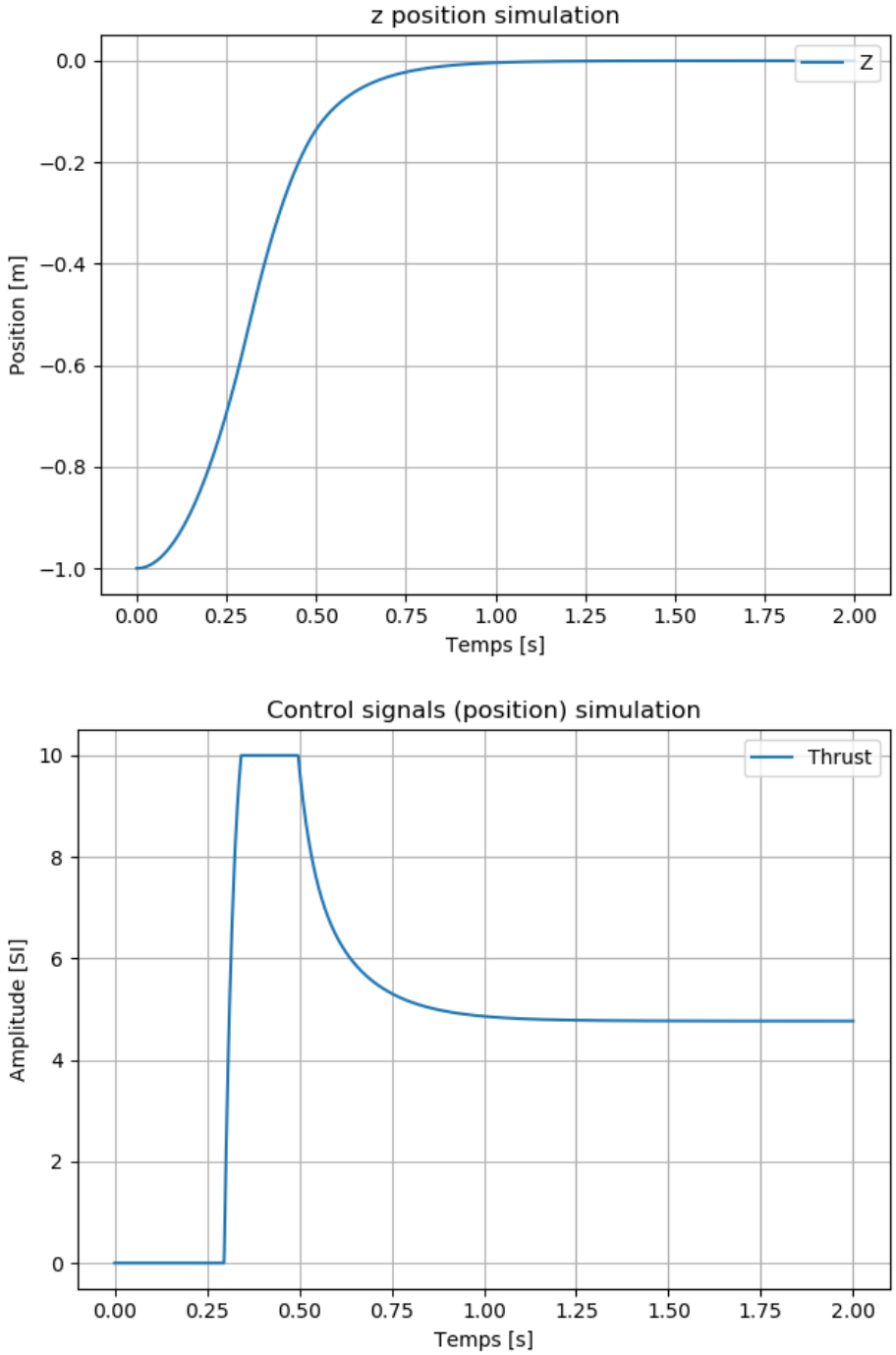

Figure 20 – Réponse en position z (haut) Thrust (bas).

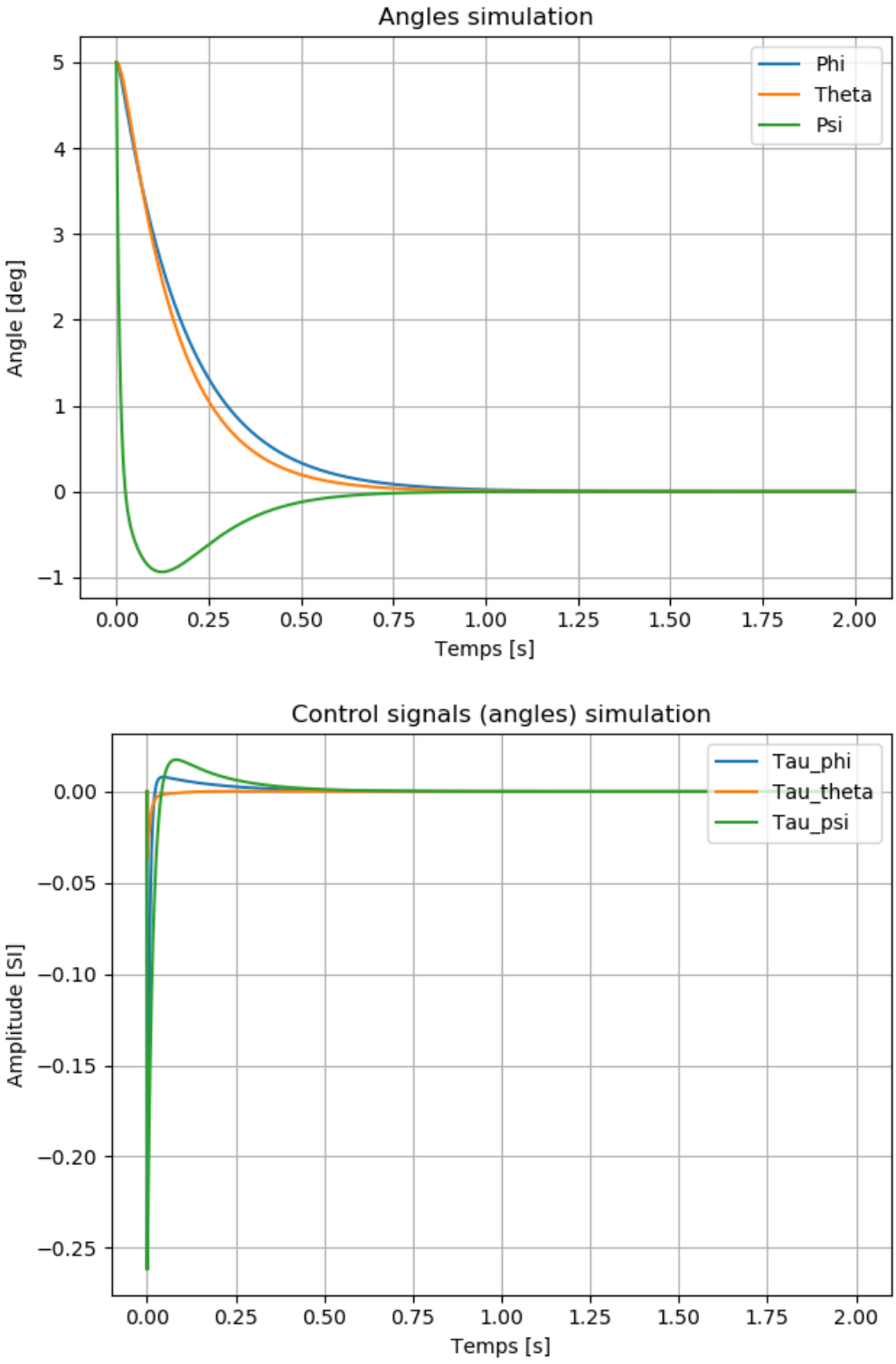

Figure 21 – Réponse en angle ϕ, θ, ψ (haut), signaux de commande des angles (bas).

9 Référence

1. Jean-Marc ALLIOT, Thomas SCHIEX, Pascal BRISSET, Frederick GARCIA, Intelligence artificielle et Informatique théorique, 2e édition.
2. Nicolas Durand, Algorithmes Génétiques et autres méthodes d'optimisation appliqués à la gestion de trafic aérien, Institut National Polytechnique de Toulouse, 2004.
3. Toward data science (medium) : `https://towardsdatascience.com/continuous-genetic-algorithm-from-scratch-with-python-ff29deedd099`
4. Toward data science (medium) : `https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3`
5. Practical Genetic Algorithms, 2nd Edition. Randy L. Haupt, Sue Ellen Haupt.
6. Code source algorithm génétique disponible sur le lien : `www.github.com/GhilesNC/AlgorithmesGenetiques`
7. Code source LQR drone control using genetic algorithm : `www.github.com/HoussemMEG/Genetic-algorithm-drone-LQR-control`

10 Annexe

drone.py

```python
from numpy import tan, sin, cos, rad2deg
import numpy as np
import matplotlib.pyplot as plt

# Initialising the model parameters
Ix = 3.8278e-3
Iy = 3.8278e-3
Iz = 7.6566e-3
Kf_ax = 5.567e-4
Kf_ay = 5.567e-4
Kf_az = 6.354e-4
g = 9.81
m = 0.486

# Simulation parameters
init_time = 0
final_time = 5
dt = 0.01
time = np.arange(init_time, final_time + dt, dt)


class Drone:
    def __init__(self, init_state=[0]*8, ref_state=[0]*8):
        # state = [phi, theta, psi, P, Q, R, Z, Z_dot]
        self.init_state = init_state
        self.ref_state = ref_state
        self.state = init_state
        self.state_memory = [self.state]
        self.control_memory = [[0]*4]

    def step(self, U):
        # Updating the new_state using the old state (not using the new calculations)
        new_state = [0]*8
        new_state[0] = self.state[0] + dt * (self.state[3] + tan(self.state[1]) *
                                             (sin(self.state[0])*self.state[4] + cos(self.state[0])*self.state[5]))
        new_state[1] = self.state[1] + dt * (cos(self.state[0])*self.state[4] - sin(self.state[0])*self.state[5])
        new_state[2] = self.state[2] + dt * (sin(self.state[0])/cos(self.state[1])*self.state[4] +
                                             cos(self.state[0])/cos(self.state[1])*self.state[5])
```

```
        new_state[3] = self.state[3] + dt * ((Iy-Iz)/Ix*self.state[4]*self.state[5] + U[0]/Ix
                                            - Kf_ax/Ix*(self.state[3])**2)
        new_state[4] = self.state[4] + dt * ((Iz-Ix)/Iy*self.state[3]*self.state[5] + U[1]/Iy
                                            - Kf_ay/Iy*(self.state[4])**2)
        new_state[5] = self.state[5] + dt * ((Ix-Iy)/Iz*self.state[3]*self.state[4] + U[2]/Iz
                                            - Kf_az/Iz*(self.state[5])**2)
        new_state[6] = self.state[6] + dt * (self.state[7]) if (self.state[6] + dt * (self.state[7])) <= 0) else 0
        new_state[7] = self.state[7] + dt * (g - U[3]/m*cos(self.state[0])*cos(self.state[1]))
        self.state = new_state
        # saving the new state and control signals in the memory in order to compute the fitness value
        self.state_memory.append(new_state)
        self.control_memory.append(U)
        return new_state

    def control(self, K):
        K = np.array(K)
        # State feedback with a small change made
        control = -K.dot(np.array(self.state)-self.ref_state) + \
                  [0, 0, 0, m*g/(cos(self.state[0])*cos(self.state[1]))]
        # Permitted control values
        control[0] = max(-1, min(control[0], 1))
        control[1] = max(-1, min(control[1], 1))
        control[2] = max(-1, min(control[2], 1))
        control[3] = max(0, min(control[3], 10))
        return control

    def simulate(self, K):
        for i in range(len(time) - 1):
            self.step(self.control(K))

    def position_show(self):
        plt.figure()
        plt.title("z position simulation")
        ax = plt.axes()
        plt.grid(b=True)
        ax.plot(time, [row[6] for row in self.state_memory], label='Z')
        plt.legend(loc='upper right')
```

```
76          ax.set(xlabel='Temps [s]', ylabel='Position [m]')
77          plt.show()
78
79      def angle_show(self):
80          plt.figure()
81          ax = plt.axes()
82          plt.title("Angles simulation")
83          plt.grid(b=True)
84          plt.plot(time, [rad2deg(row[0]) for row in self.
    state_memory], label="Phi")
85          plt.plot(time, [rad2deg(row[1]) for row in self.
    state_memory], label="Theta")
86          plt.plot(time, [rad2deg(row[2]) for row in self.
    state_memory], label="Psi")
87          plt.legend(loc='upper right')
88          ax.set(xlabel='Temps [s]', ylabel='Angle [deg]')
89          plt.show()
90
91      def control_angle_show(self):
92          plt.figure()
93          ax = plt.axes()
94          plt.title("Control signals (angles) simulation")
95          plt.grid(b=True)
96          plt.plot(time, [row[0] for row in self.control_memory],
    label="Tau_phi")
97          plt.plot(time, [row[1] for row in self.control_memory],
    label="Tau_theta")
98          plt.plot(time, [row[2] for row in self.control_memory],
    label="Tau_psi")
99          plt.legend(loc='upper right')
100         ax.set(xlabel='Temps [s]', ylabel='Amplitude [SI]')
101         plt.show()
102
103     def control_thrust_show(self):
104         plt.figure()
105         ax = plt.axes()
106         plt.title("Control signals (position) simulation")
107         plt.grid(b=True)
108         plt.plot(time, [row[3] for row in self.control_memory],
    label="Thrust")
109         plt.legend(loc='upper right')
110         ax.set(xlabel='Temps [s]', ylabel='Amplitude [SI]')
111         plt.show()
112
113     def reset_memory(self):
114         self.state_memory = [self.init_state]
115         self.control_memory = [[0]*4]
116         self.state = self.init_state
```

main.py

```
from drone import Drone
from genetic_algorithm import *
from math import pi


def deg2rad(list1):
    list1[0] = list1[0] / 180 * pi
    list1[1] = list1[1] / 180 * pi
    list1[2] = list1[2] / 180 * pi
    return list1


# Initialisation and algorithm object creation
steps = 200
population_number = 100
ga = GA(population_number)

# Creating, initialising and giving to the drone the reference
# The angles are in degrees, and : state = [phi, theta, psi, P, Q, R, Z, Z_dot]
init_state = deg2rad([5, 5, 5, 0, 0, 0, -1, 0])
ref_state = deg2rad([0, 0, 0, 0, 0, 0, 0, 0])
drone = Drone(init_state=init_state, ref_state=ref_state)

for generation in range(steps):
    all_state_memories = []
    all_control_memories = []
    if generation % 10 == 0:
        print("Generation :", generation)
    for index in range(population_number):
        # Resetting the drone memory
        drone.reset_memory()
        # Simulating the drone for that gain
        drone.simulate(ga.population[index])
        # Storing the memory for the fitness calculation
        all_state_memories.append(drone.state_memory)
        all_control_memories.append(drone.control_memory)
    # Selecting individual
    selection = ga.selection(all_state_memories, all_control_memories)
    # Mating and mutation
    ga.mating(selection)
    ga.mutation()
    # Printing and showing graphs part
    if step == steps - 1:
        print('\ngain :', ga.population[index])
```

```
45          drone.position_show()
46          drone.angle_show()
47          drone.control_angle_show()
48          drone.control_thrust_show()
```

genetic_algorithm.py

```
import numpy as np
from numpy.random import randint
from random import random, gauss, uniform
from drone import dt
from math import isnan

# Fitness calculation variables (LQR)
R = [350, 35, 28000, 0.36]
Q = [8, 0.1, 8, 0.1, 5, 0.00000003, 3400, 50]


class GA:
    def __init__(self, pop_size, genes_nb=8, genes_upper_lim=0, genes_lower_lim=2, z_enable=True):
        self.pop_size = pop_size
        self.genes_nb = genes_nb
        self.genes_upper_lim = genes_upper_lim
        self.genes_lower_lim = genes_lower_lim
        self.z_enable = z_enable
        self.population = []
        self.fitness = []
        self.init_pop()
        self.mutation_variance = 0.8
        self.reproduction_rate = 0.6

    def init_pop(self):
        for _ in range(self.pop_size):
            self.population.append(self.individual())

    def individual(self):
        # Creating individual
        individual = []
        for i in range(3):
            temp = [0]*8
            temp[i] = round(random() * (self.genes_upper_lim - self.genes_lower_lim) + self.genes_lower_lim, 2)
            temp[i+3] = round(random() * (self.genes_upper_lim - self.genes_lower_lim) + self.genes_lower_lim, 2)
            individual += [temp]

        # z part
        individual += [[0]*8]
        individual[-1][-2] = round(random() * -100, 2)
        individual[-1][-1] = round(random() * -100, 2)
        return individual

```

```
    def individual_fitness(self, response, control_signal):
        # Calculating fitness
        # Squaring response part
        phi_squared = [Q[0]*element[0]**2 for element in response]
        phi_dot_squared = [Q[3]*element[1]**2 for element in
response]
        theta_squared = [Q[1]*element[2]**2 for element in response
]
        theta_dot_squared = [Q[4]*element[3]**2 for element in
response]
        psi_squared = [Q[2]*element[4]**2 for element in response]
        psi_dot_squared = [Q[5]*element[5]**2 for element in
response]
        z_squared = [Q[6]*element[6]**2 for element in response]
        z_dot_squared = [Q[7]*element[7]**2 for element in response
]

        # Squaring control signal part
        tau_phi_squared = [R[0]*element[0]**2 for element in
control_signal]
        tau_theta_squared = [R[1]*element[1]**2 for element in
control_signal]
        tau_psi_squared = [R[2]*element[2]**2 for element in
control_signal]
        thrust_squared = [R[3]*element[3]**2 for element in
control_signal]

        # Adding the weighted parts
        total = [phi_squared[i] + phi_dot_squared[i] +
theta_squared[i] + theta_dot_squared[i] +
                 psi_squared[i] + psi_dot_squared[i] + z_squared[i
]+z_dot_squared[i] +
                 tau_phi_squared[i] + tau_theta_squared[i] +
                 tau_psi_squared[i] + thrust_squared[i] for i in
range(len(response))]
        integral = dt * ((total[0]+total[-1])/2 + sum(total[1:-1]))
        if isnan(integral):
            return 9999999
        else:
            return integral

    def calculate_fitness(self, state_memories, control_memories):
        # Calculate fitness for each individual and sorting them
        fitness = [[round(self.individual_fitness(state_memories[i
], control_memories[i]), 2), i]
                   for i in range(len(state_memories))]
        fitness_sorted, sorted_index = zip(*sorted(fitness, reverse
=True))
```

```
78          self.population = [self.population[i] for i in sorted_index
    ]
79          print(fitness_sorted, " = ", round(sum(fitness_sorted)))
80          self.fitness = fitness_sorted
81
82      def selection(self, state_memories, control_memories):
83          # Roulette wheel selection
84          self.calculate_fitness(state_memories, control_memories)
85          chance = gauss(self.reproduction_rate, 0.11)
86          while chance > 1 or chance < 0:
87              chance = gauss(self.reproduction_rate, 0.11)
88          # Reversing the fitness so the ones who has the lowest
    fitness have a higher chance to get selected
89          fitnesses = [round(abs(self.fitness[i]-1.1*max(self.fitness
    )), 2) for i in range(len(self.fitness))]
90          sum_fitness = sum(fitnesses)
91          fitness_normalised = [fitness/sum_fitness for fitness in
    fitnesses]
92          # Selecting
93          fitness_cumsum = list(np.cumsum(fitness_normalised))
94          fitness_cumsum.append(chance)
95          selection_index = sorted(fitness_cumsum).index(chance)
96          selection = list(range(selection_index, self.pop_size))
97          while len(selection) < 2:
98              selection = self.selection(state_memories,
    control_memories)
99          else:
100             if len(selection) % 2 == 1:
101                 return selection[1:]
102             else:
103                 return selection
104
105     def mating(self, selection):
106         for i in range(0, len(selection), 2):
107             pivot_point = randint(1, self.genes_nb)
108             child_one = []
109             child_two = []
110             for j in range(4):
111                 if self.z_enable and j == 3:
112                     pivot_point = 7
113                 child_one += [self.population[selection[i]][j][0:
    pivot_point] +
114                               self.population[selection[i+1]][j][
    pivot_point:]]
115                 child_two += [self.population[selection[i+1]][j][0:
    pivot_point] +
116                               self.population[selection[i]][j][
    pivot_point:]]
```

```
            self.population[i] = child_one
            self.population[i+1] = child_two

    def mutation(self):
        # Mutation for all genes of all individuals except the 2 best individual
        for j, person in enumerate(self.population[:-2]):
            for k, genes in enumerate(person):
                for i, gene in enumerate(genes):
                    if uniform(0, 1) <= 0.3 and self.population[j][k][i] != 0:
                        if k == 3 and (i == 6 or i == 7):
                            self.population[j][k][i] = max(-100, min(round(gauss(gene, self.mutation_variance), 2), 0))
                        else:
                            self.population[j][k][i] = max(self.genes_lower_lim, min(round(gauss(gene, self.mutation_variance), 2), self.genes_upper_lim))
```

maximum_fonction.m

```
% Maximum d'une foction %
clc
clear

% Compteur de generation
Compteur = 0;

% Fonction a maximiser
Fonction = @(x) 4*(x.*(1-x));

% Taille de la population
Taille = 5;
% Initialisation de la population
Population = logical(decimalToBinaryVector(randperm(256,Taille)-1,8)');
DecGray = binaryVectorToDecimal(Population')';
Dec = gray2bin(DecGray,'fsk',256);
% Initialisation de l'utilite
Utilite = Fonction( Dec/256 );
UtiliteMax = max(Utilite);

fprintf('> Generation : %d \n',Compteur);
fprintf('> Meilleure Utilite : %f \n',UtiliteMax);
pause

while UtiliteMax<1
    % Selection
    Reproduction = cumsum(Utilite)/sum(Utilite);
```

```
    Random = rand(Taille,1);
    I = 1+sum(Random>Reproduction,2);
    S1 = Population(:,I);
    Reproduction = cumsum(Utilite)/sum(Utilite);
    Random = rand(Taille,1);
    I = 1+sum(Random>Reproduction,2);
    S2 = Population(:,I);

    % Croisement
    I = logical(randi([0 1],8,Taille));
    C = S1;
    C(I) = S2(I);

    % Mutation
    I = (randi(10,8,Taille)==1);
    M = C;
    M(I) = not(M(I));

    % Nouvelle Generation
    Compteur = Compteur + 1;
    Population = M;
    DecGray = binaryVectorToDecimal(Population')';
    Dec = gray2bin(DecGray,'fsk',256);
    Utilite = Fonction( Dec/256 );
    UtiliteMax = max(Utilite);

    fprintf('\n');
    fprintf('> Generation : %d \n',Compteur);
    fprintf('> Meilleure Utilite : %f\n',UtiliteMax);

end
```

touver_le_message.m

```
% Trouver le message %
clc
clear

% Compteur de generation
Counter = 0;

% Characteres disponibles
CharSet = [32,46,65:90,97:122];
% Phrase cible
Target = 'Message Cible.';
% Longueur de Phrase
Length = length(Target);

```

```
% Fonction a maximiser
Function = @(Phrase) sum(Phrase==Target,2)/Length;

% Pourcentage de mutation
MutPercent = 5;

% Taille de la population
Size = 50;
% Initialisation de la population
Population = char(CharSet(randi(54,Size,Length)));
% Initialisation de l'utilite
Fitness = Function( Population );
[MaxFitness,MaxFitnessI] = max(Fitness);
fprintf('> Generation : %d ',Counter);
fprintf('> Meilleure Utilite : %f ',MaxFitness);
fprintf(' %s ',Population(MaxFitnessI,:));

while MaxFitness<1
%   Selection
    Reproduction = cumsum(Fitness)/sum(Fitness);
    Random = rand(1,Size);
    I = 1+sum(Random>Reproduction);
    Parent1 = Population(I,:);
    Random = rand(1,Size);
    I = 1+sum(Random>Reproduction);
    Parent2 = Population(I,:);

%   Croisement
    Random = logical(randi([0 1],Size,Length));
    Children = Parent1;
    Children(Random) = Parent2(Random);

%   Mutation
    I = (randi(1000,Size,Length)<=10*MutPercent);
    Mutant = char(CharSet(randi(54,sum(I(:)),1)));
    Children(I) = Mutant;

%   Nouvelle Generation
    Counter = Counter + 1;
    Population = Children;
    Fitness = Function( Population );
    [MaxFitness,MaxFitnessI] = max(Fitness);
    fprintf('\n');
    fprintf('> Generation : %d \n',Counter);
    fprintf('> Meilleure Utilite : %f \n',MaxFitness);
    fprintf('%s',Population(MaxFitnessI,:));

end
```

piles_cartes.m

```
% Piles de cartes %
clc
clear

% Compteur de generation
Counter = 0;

% Entiers disponibles
Length = 10;
Integers = 1:Length;

% Somme/Produit cible
TargetSum = 36;
TargetProduct = 360;

% Fonction a maximiser
Function = @(S,P) 1 - ( abs(36-S)/36 + abs(log(P/360))/log(10080) )/2;

% Taille de la population
Size = 40;
% Initialisation de la population
Decimal = randperm(2^Length,Size)-1;
Population = logical(decimalToBinaryVector(Decimal,Length));
% Initialisation de l'utilite
Sum = CalcSum(not(Population));
Product = CalcProduct(Population);
Fitness = Function( Sum , Product );
[MaxFitness,MaxFitnessI] = max(Fitness);

Display(Counter,MaxFitness,Sum(MaxFitnessI),Product(MaxFitnessI),Population(
   MaxFitnessI,:));
pause

while MaxFitness<1
%   Selection
    Reproduction = cumsum(Fitness)/sum(Fitness);
    Random = rand(1,Size);
    I = 1+sum(Random>Reproduction);
    Parent1 = Population(I,:);
    Random = rand(1,Size);
    I = 1+sum(Random>Reproduction);
    Parent2 = Population(I,:);

%   Croisement
    Random = logical(randi([0 1],Size,Length));
    Children = Parent1;
```

```
46      Children(Random) = Parent2(Random);
47
48  %   Mutation
49      I = (randi(10,Size,Length)==1);
50      Mutant = not(Children(I));
51      Children(I) = Mutant;
52
53  %   Nouvelle Generation
54      Counter = Counter + 1;
55  %   AllGenes = [Children;Population];
56  %   Sum = CalcSum(not(AllGenes));
57  %   Product = CalcProduct(AllGenes);
58  %   Fitness = Function( Sum , Product );
59  %   [~,I] = sort(Fitness,'descend');
60  %   AllGenes = AllGenes(I,:);
61  %   AllGenes = unique(AllGenes,'rows','stable');
62  %   Population = AllGenes(1:Size,:);
63      Population = Children;
64      Sum = CalcSum(not(Population));
65      Product = CalcProduct(Population);
66      Fitness = Function( Sum , Product );
67      [MaxFitness,MaxFitnessI] = max(Fitness);
68      Display(Counter,MaxFitness,Sum(MaxFitnessI),Product(MaxFitnessI),Population
            (MaxFitnessI,:));
69
70  end
71
72  function Y = CalcSum(X)
73  Integers = 1:size(X,2);
74  Y = X*Integers';
75  end
76
77  function Y = CalcProduct(X)
78  Integers = 1:size(X,2);
79  Y = X.*Integers;
80  Y(not(Y)) = 1;
81  Y = prod(Y,2);
82  end
83
84  function Display(C,MFF,MFS,MFP,MF)
85  fprintf('> Generation : %d \n',C);
86  fprintf('> Meilleure Utilite : %f ',MFF);
87  fprintf('  Somme : %d ',MFS); S = find(not(MF));
88  fprintf('= %d',S(1)); fprintf(' + %d',S(2:end));
89  fprintf('  Produit : %d ',MFP); P = find(MF);
90  fprintf('= %d',P(1)); fprintf(' x %d',P(2:end));
91  end
```