Projet Tatooine

Rapport de Marius Ortega

1. Taille de l'espace de Recherche

Dans ce problème nos recherchons les coefficients p_1, p_2, p_3, p_4, p_5 et p_6 permettant de modéliser la trajectoire du satellite dans la bordure extérieure de Tatooine, ce dernier se mouvant, d'après l'énoncé du problème, en suivant une orbite de Lissajous. On peut exprimer une orbite de Lissajous comme suit :

$$\begin{cases} x(t) = p_1 \times \sin(p_2 \times t + p_3) \\ y(t) = p_4 \times \sin(p_5 \times t + p_6) \end{cases}$$

En nous appuyant sur l'énoncé du problème, nous décidons de limiter arbitrairement l'espace des solutions possibles tel que :

$$\begin{cases} p_1 \in [-100\,;100] \\ p_2 \in [-100\,;100] \\ p_3 \in [-100\,;100] \\ p_4 \in [-100\,;100] \\ p_5 \in [-100\,;100] \\ p_6 \in [-100\,;100] \end{cases}$$

Dès lors, on peut conclure que notre vecteur solution S est tel que :

$$S = [p_1, p_2, p_3, p_4, p_5, p_6] \in [-100; 100]^6$$

2. Définition de la fonction de fitness

Nous a été fourni, avec le sujet du problème, un ensemble d'observations du satellite problématique. Soit une observation o est un vecteur tel que :

$$o = [x, y, t]$$

Avec x la position selon la direction x, y la position selon la direction y et t le temps auquel ces informations ont été recueillies.

Ces observations définissent des points. Par ailleurs, appellera e (de la même forme que les observations) les points expérimentaux calculés à partir des coefficients de Lissajous trouvés par notre algorithme.

De ce fait, on définit notre fonction de fitness f d'un vecteur e comme la somme des distances euclidiennes entre e et les observations fournies o.

$$f_e = \sum_{i=1}^{n} \left(\sqrt{\left(y_e - y_{o_i}\right)^2 + \left(x_e - x_{o_i}\right)^2} \right)$$

3. Description des opérateurs

Pour permettre la mise en place de notre algorithme génétique, nous avons besoin de définir plusieurs opérations. L'opérateur de mutation ainsi que l'opérateur de croisement.

- **Opérateur de mutation :**
  Pour cet opérateur, nous avons décidé d'opérer des mutation ponctuelle sur nos individus. Pour un vecteur S, on tire aléatoirement entre 0 et 5 un nombre de mutation que nous allons lui appliquer. Ensuite, nous générons un nombre entre -100 et 100 qui définira la mutation du coefficient de cette solution et donc sa nouvelle valeur.

- **Opérateur de croisement :**
  Le croisement s'appuie de nouveau sur l'aléatoire. Pour recontextualiser, la fonction de croisement prend en paramètre deux individu i_1 et i_2, et retourne deux individus croisés c_1 et c_2. Dans un premier temps, on tire aléatoirement un nombre k entre 0 et 5 (parmi les indices d'un vecteur S). On créer alors nos deux nouveaux individus comme tel :

$$\begin{cases} c_1 = i_1[:k] + i_2[len(i_2) - k:] \\ c_2 = i_2[:k] + i_1[len(i_1) - k:] \end{cases}$$

4. Processus de sélection

Enfin, notre opérateur de sélection nous permet, à chaque génération d'individus, de sélectionner une certaine partie de la population. On effectue ici une sélection par élitisme comportant une légère variante.

Soit $k\ et\ l \in \mathbb{N}$ avec $l < len(S)$. A chaque génération, on a choisi arbitrairement de sélectionner les k meilleurs individus et les l individus les moins performants. Pour ce faire, on trie la population actuelle sur la base de la fitness de ses individus, puis on applique la sélection comme détaillé plus haut. Par ailleurs, la prise en compte d'individus peu performants permet d'éviter à notre solution de stagner dans des minimums locaux de fitness.

5. Taille de Population et Convergence vers une solution stable

Après expérimentation, nous avons adopté une taille initiale de population $N_0 = 10$. En comptant l'ajout des mutations, des croisements et des nouveaux individus, nos tailles de population $(N_i)_{i\in[2,n]}$ sont telle que :

$$(N_i)_{i\in[2,n]} = N_{selection} + N_{mutation} + N_{croisement} + N_{nouveaux} = 17$$

Nous détaillerons dans la partie 7 les tests qui ont mené à ce choix empirique. Pour rester succinct, cette taille de population apparaissait comme le compromis idéal entre une population grande avec une vitesse de convergence rapide mais un grand temps calcul pour chaque génération et une population trop restreinte induisant un temps de calcul rapide pour chaque génération mais une vitesse de convergence trop faible.

Dans son état actuel, notre programme s'arrête et nous renvoie le meilleur individu de la population actuelle si la fitness du meilleur individu est inférieur à 150. On a pu noter par des observations préalables qu'à cette valeur de fitness, notre solution converge de manière solidement stable.

Afin de déterminer la vitesse de convergence de notre programme, nous nous sommes basés sur 10 compilations de ce dernier. De ces expériences nous avons tiré plusieurs informations à propos de nos sets de données.

Tout d'abord les coefficients p_3 et p_6 sont très instables dans le temps. Par conséquent, nous avons fait abstraction de ces derniers pour caractériser la vitesse de convergence de notre algorithme.

De cette analyse des données, nous avons tiré que notre algorithme converge vers une solution stable à en moyenne 1622 générations. Après un tel nombre de générations, nos coefficients p_1, p_2, p_4, p_5 ne varient plus ou peu. On peut également relever que la convergence est notable dès que notre fonction de fitness $f < 200$.

6. Temps de compilation

En basant notre explication sur 10 compilations de notre programme, nous obtenons les données suivantes. On rappelle que notre condition d'arrêt est $f < 150$ pour notre meilleur élément :

Essai	Temps de compilation (s)	Erreur
1	16.86	65.3813
2	15.7	70.4258
3	121.48	98.4115
4	53.17	99.1491
5	255.87	67.6767
6	67.58	95.4246
7	59.72	95.5724
8	102.21	96.6123
9	63.51	99.9789
10	133.95	99.6456

Dès lors, on peut conclure sur un temps de compilation moyen $\bar{T}_{exec}$:

$$\bar{T}_{exec} = 88.998 \approx 89s$$

7. Autres Solutions testées et Difficultés

- **Taille de Population :**
  Initialement, j'avais initialisé ma taille de population à 25. Toutefois, après plusieurs tests j'ai remarqué que le temps de compilation pour chaque population était relativement long. C'est pour cela que j'ai décidé de diminuer la taille de population initiale à 10. De fait, la compilation est en moyenne plus rapide même s'il arrive que l'algorithme soit bloqué plus longtemps dans des minimums locaux que ses homologues ayant une taille de population plus grande. Pour conclure, une petite population de 10 individus permet à l'algorithme de converger rapidement vers des solutions avec des fitness avoisinant 150 mais montre ses limites lors de calculs très précis (fitness inférieure à 100 voire 20).

- **Fonction de fitness :**
  Lors de sa toute première version notre algorithme ne calculait pas la distance euclidienne entre les points observés et les points générés aléatoirement pour calculer une fitness. En effet, nous calculions la somme de la différence entre les abscisses et de la différence entre les ordonnées. Dans cette configuration, l'algorithme fonctionne très mal. La fitness du meilleur élément descendait péniblement sous 400. Après une séance de Debug, j'ai compris que le problème de cette approche se posait lorsque des points avaient des coordonnées négatives ce qui engendrait des distances négatives entre les points et faussait complètement le calcul de la fitness qui consiste en la somme de ces distances. Par conséquent, j'ai pensé à la norme euclidienne qui résout complètement ce problème.