**Rapport « IA : Approximation de fonction par Algorithme génétiques »**

**CHUVATIN Alexandra**

**TD P**

# I – Introduction

Des astronomes ont observé une nouvelle étoile inconnue. On souhaite trouver une fonction qui approche au mieux la température de celle-ci. Les astronomes supposent que l'étoile suit une fonction dite de Weierstrass avec plusieurs périodicités imbriquées au sens fractale que l'on définit comme suit:

*Avec t(i) la température de l'étoile à un instant i donné, avec pour ensemble de paramètres:*

On souhaite réaliser un algorithme génétique capable de trouver une bonne combinaison de paramètres expliquant au mieux la température de l’étoile. Pour répondre à ce problème, on définit donc un individu comme un combinaison des 3 paramètres recherchés : le triplet (a, b, c), répondant aux contraintes ci-dessus.

La taille de l’espace de recherche dépend des paramètres a, b et c. De plus, le nombre de chiffre après la virgule pour le paramètre a joue également sur la taille de l’espace de recherche. Dans la version finale de mon programme, le nombre est arrondi à 4 chiffres après la virgule, ce qui donne un espace de recherche de soit possibilités.

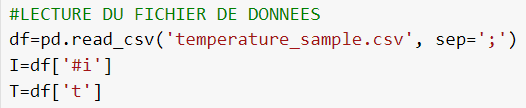
# II – Plan d’action

1. Fonction de Fitness

La première étape de la création de l’algorithme était de créer la fonction de fitness pour ce problème. On souhaite rapprocher la fonction de Weierstrass au plus près des observations de températures effectuées à des instants données. Notre fonction de fitness aura donc la forme suivante :

*Où W(i) est la température approchée obtenue avec les paramètres (a, b, c) à l’instant i appliqués à la fonction de Weierstrass*

*et T(i) la température réelle de l’étoile donnée dans le tableau de données, à l’instant i.*

Le fichier « *temperature\_sample.cvs* » fournit un tableau de données contenant les instants i ainsi que les températures correspondantes pour chaque instant. Dans le programme python, les instants i sont stockés dans la liste « I » et les températures dans la liste « T ».

On cherchera à faire tendre la fonction de fitness vers 0. C’est donc l’individu avec la fitness la plus basse qui définira le meilleur triplet de la fonction de Weierstrass, pour décrire au mieux la température de l’étoile. Cependant, les valeurs données dans le fichier étant bruitées, il n’était pas possible de tomber sur une fitness nulle. Ainsi, pour vérifier le bon fonctionnement de la méthode qui calcule la faction de fitness, j’ai appliqué celle-ci à un tableau de valeurs qui retourne une fitness de 0 pour une certaine bonne combinaison.

1. Opérateurs mis en œuvre

Afin de répondre à ce problème, un algorithme génétique basé sur le principe de mutation, et de croisement dans certains cas, a été mis en place.

L’individu de départ prend un triplet aléatoire répondant aux intervalles données. A chaque itération, un des 3 paramètres a, b ou c sera muté de manière aléatoire. Une fois l’individu muté obtenu, on compare son fitness à l’individu précédent. Si celle-ci est meilleure, on continu les mutations sur ce dernier, sinon, on effectue un croisement entre nos 2 individus. Le croisement s’effectue de manière aléatoire sur l’un des paramètres en le permutant entre les 2 individus. A chaque itération de la méthode de croisement, les 2 meilleurs individus de l’itération précédente sont croisés. La méthode retourne finalement l’individu ayant le meilleur fitness parmi les 2 meilleurs individus de la dernière itération.

 La population initiale est donc un individu généré aléatoirement.

NB : la méthode qui réalise les croisement génère 2 individus croisés et les compare avec les deux individus parents, pour ensuite récupérer le couple d’individus ayants la meilleure fitness. Cela sert à réaliser si besoin un autre croisement juste après à partir de ces 2 meilleurs individus. Cependant, ce code est positionné dans une boucle *while iteration < 1*. Il était donc possible de sélectionner uniquement le meilleur individu des 4 et non pas un couple. Cette boucle était en fait mise en place avec un nombre d’itérations nettement supérieur auparavant. Mais après plusieurs tests, laisser une seule itération pour le croisement (qui intervient lorsqu’une mutation n’est pas « réussie ») était bien plus rapide, et donnait quand même une fitness finale proche de 0.735 (sachant que la méthode principale qui réalise les mutation a quant à elle 400 itérations).

1. Résultats

Etant donné que les données fournies dans le fichier étaient bruitées, la fonction de fitness n’approchait pas de 0 mais d’environ 0.735. Ainsi, la meilleure valeur de fitness que j’ai pu trouver est de 0.7351090824970385 avec les valeurs (0.3528741891 ; 15 ; 2).

Il faudra finalement 400 itérations pour converger vers une solution stable (qui converge vers 0.735). Le programme met en moyenne 586 ms à s’exécuter. Ce temps a été calculé à l’aide de la méthode **%timeit** de Python.

# III – Problèmes rencontrés & améliorations

Avant d’arriver à cet algorithme génétique mettant en œuvre des mutations et des croisements, j’avais mis en place uniquement des mutations. Le problème était dans ma condition d’arrêt, pour laquelle je comparais la fitness à une valeur cible de 0,74 (while fitness > 0,74…) pour obtenir à chaque fois une valeur proche du résultat cherché. Mais cette condition a été trouvée à tâtons, et n’est donc plus valable si les données changent (si par exemple nos données permettent un fitness de 0, le programme s’arrêtera avant d’atteindre cette valeur). Si au contraire la condition d’arrêt était un certain nombre d’itérations, la solution n’était pas du tout stable, et retournait un fitness optimal 1 fois sur 5, voire sur 10 ; d’où la nécessité de fixer un objectif de fitness dans cette ancienne version de l’algorithme.

Une fois une méthode qui réalise des croisements en plus des mutations implémentée, le programme était beaucoup plus efficace et nécessitait uniquement 400 itérations pour converger vers la plus petite fitness.

De plus, ma population initiale ne comprend qu’un seul individu. Après réflexion, je pense qu’il était encore possible d’optimiser mon programme en partant d’une population nettement supérieure. Cela permettrait surement de gagner en temps sur l’exécution du programme car cela évitera de faire autant de boucles qu’actuellement.

Aussi, le choix de réaliser en premier lieu des mutations et de procéder à des croisements dans le cas d’une mutation qui donne un résultat pire est totalement arbitraire. Il aurait été totalement possible de commencer par des croisements puis appliquer des mutations. Cependant, je ne sais pas si cela a un impact sur l’efficacité du programme, étant donné que je n’ai pas eu le temps d’implémenter une version pareille.

Finalement, mon algorithme converge vers une solution optimale en 400 itérations avec un temps d’exécution proche d’une demi-seconde, c’est pourquoi je suis totalement satisfaite du résultat obtenu, bien que de vraies améliorations existent sûrement.