Martin Mouly

Approximation de fonction par Algorithme génétique

1. Objectif

L'objectif du projet est d'approximer une fonction de Weierstrass calculant la température d'une étoile à un instant *i* donné. La fonction *t(i)* dépend de trois paramètres *a,b,c* avec *a* un réel compris entre 0 et 1 (exclus) et b et c deux entiers naturels compris entre 1 et 20 (inclus). Théoriquement, l'espace de recherche de ce problème est infini car *a* peut prendre un nombre infini de valeur, mais pour pouvoir réaliser l'algorithme génétique, on approchera *a* à 10^{-2} près. On a donc *a* pouvant prendre 99 valeurs et *b* et *c* peuvent prendre 20 valeurs chacun, ce qui fait un espace de recherche de $99 \times 20 \times 20 = 3{,}96 \times 10^4$.

2. Individu et fonction Fitness

Ma population est une liste composée d'individus sous forme de liste [a,b,c,f]: a, b et c sont les paramètres à trouver et f est la valeur fitness de l'individu a,b,c. Associer f à a,b,c me permet d'associer directement le calcul de la fitness à l'individu sélectionné. J'ai choisi d'utiliser pour les individus le type liste au lieu de tuple pour pouvoir faciliter l'opération de mutation. La génération de la population d'individus s'effectue à l'aide de la fonction **Population**(count) - count étant le nombre d'individus à générer.
Ma fonction fitness **Fitness**(a,b,c) calcule l'écart entre les résultats expérimentaux (donnés dans le fichier temperature_sample) et les résultats de la fonction de Weierstrass **Weier**(a,b,c,i) pour chaque i donné. La fonction retourne ensuite la moyenne de ces écarts. Plus la moyenne se rapproche de 0, plus les paramètres testés se rapprochent des paramètres à trouver.

3. Sélection

Après le calcul de la fonction Fitness, il faut sélectionner les individus que l'on va croiser et muter. La fonction **Selection**(pop,hcount,lcount) permet de trier en ordre croissant la population selon la valeur de la fitness de chaque individu, puis de créer une sous-population comprenant les *hcount* meilleurs individus et *lcount* pires individus de la population. Cette sous-population sera la population où les croisements et mutations vont être effectués.
Initialement, ma fonction de sélection prenait uniquement en compte les meilleurs individus de la population (la première moitié), ce qui avait pour conséquence de faire converger les paramètres a,b et c vers un extremum local. Une autre fonction de sélection que j'ai étudiée est la « roue de la fortune », où pour chaque individu, la probabilité d'être sélectionné est proportionnelle à la valeur de sa fitness. Cependant, implémenter cette fonction en python me paraissait assez complexe et m'aurait pris beaucoup de temps.

4. Croisement

Les individus sélectionnés vont être croisés à l'aide de la fonction **Croisement**(pop). Cette fonction parcourt la population de deux en deux, et à chaque itération, croise deux individus pour en créer deux nouveaux. L'enfant1 récupère le a du parent1 et le b et le c du parent2 et

l'enfant2 récupère le a du parent2 et le b et le c du parent1. L'attribution de parent1 et parent2 se fait de manière aléatoire.

5. Mutation

Après avoir été croisés, on va muter certains individus de la population en utilisant la fonction **Mutation**(pop). Pour muter un individu, on choisit aléatoirement un des 3 paramètres à muter, avec une probabilité pour a de muter supérieure à b et c, car l'espace de recherche de a est 5 fois plus grand que celui de b et c. Initialement, j'avais prévu de muter chacun des individus de la population, mais cela avait pour conséquence la difficulté à converger vers une solution. Au contraire, en mutant peu d'individus, mon algorithme convergeait vers un extremum local. J'ai donc décider d'opter pour la mutation d'⅓ des individus.

6. Résultats

6.1. Taille de la population

Pour faire fonctionner l'algorithme génétique, il faut tout d'abord choisir la taille de la population de base. Une trop grosse population va augmenter le nombre de calculs à effectuer et donc augmenter le temps du programme. Au contraire, une trop faible population risque de retourner un extremum local. Après avoir effectué de nombreux tests, j'ai choisi une population initiale de 20 individus.

6.2. Convergence de la solution et temps de calcul

L'objectif de la fonction Fitness est de retourner la moyenne des écarts et donc le bon individu est trouvé lorsque la fonction Fitness retourne 0. Vu que l'on cherche à approcher la fonction, on ne pourra pas trouver de valeurs a,b,c tels que la fitness retourne 0. Il faut donc établir un critère de convergence pour pouvoir obtenir un résultat. Lorsque pour un grand nombre n de générations le meilleur individu ne change pas, on peut alors considérer le résultat comme le résultat final. Si n est trop petit, le programme risque de retourner un extremum local ou bien une valeur proche de l'extremum global. Au contraire lorsque est n est trop grand, le programme va mettre beaucoup plus de temps à retourner le résultat final. Après avoir effectué de nombreux tests, j'ai choisi d'opter pour n=100.
Au final la solution converge vers **(0.35,15,2).**
Sur 500 runs testés, 99% convergent vers cette solution, et le temps moyen d'un run est d'environ 0.37 secondes.

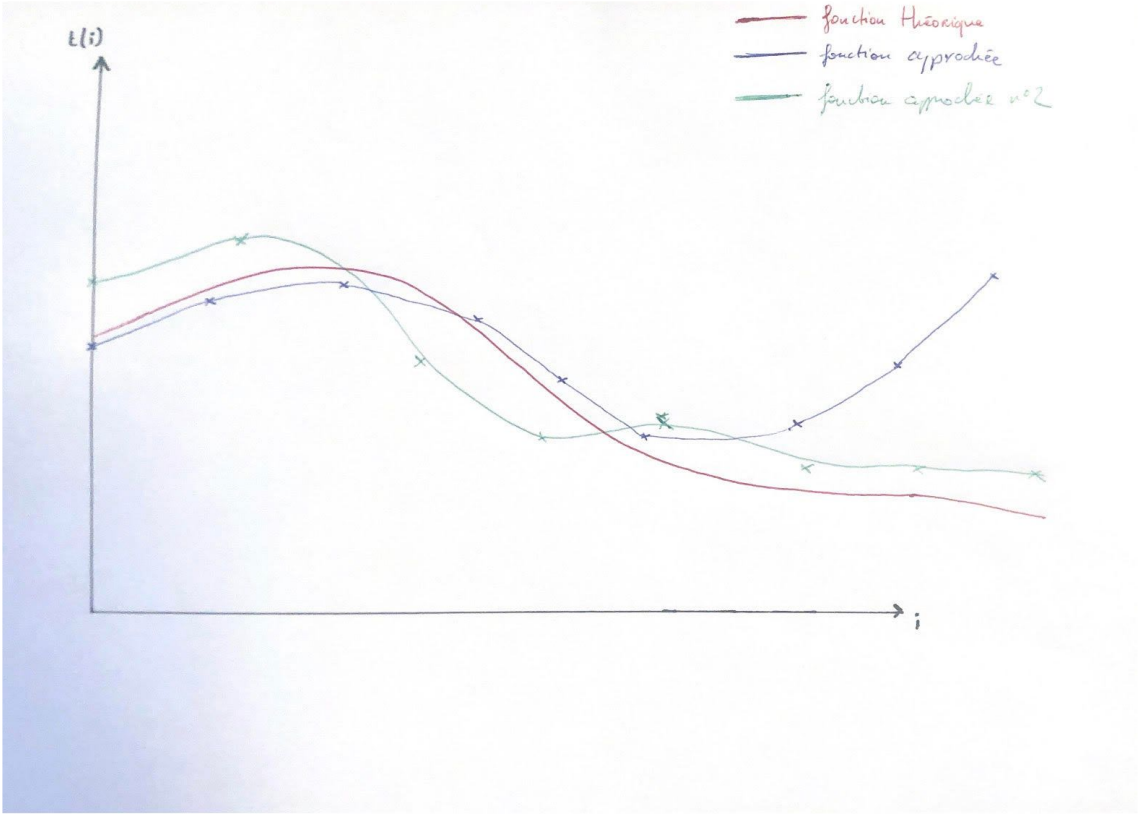
```
In [164]: Algo(500)
RESULTAT
a = 0.35
b = 15
c = 2
TEMPS MOYEN SUR 500 RUNS: 0.37 SECONDES
TAUX DE REUSSITE CONVERGENCE: 99.0%
```

7. Difficultés et améliorations

Une des grosses difficultés de cet algorithme est de converger vers une solution qui n'est pas un extremum local. Pour cela, il faut bien configurer les opérateurs de croisement et mutation, bien choisir la taille initiale de la population et le critère de validité. Comme expliqué précédemment, j'ai eu recours à plusieurs systèmes de fonction de mutation et de fonction de croisement avant de finalement opter pour les opérateurs actuels. De même, il fallait choisir des paramètres de manière à optimiser les temps de calcul tout en garantissant de ne pas tomber dans un extremum local.

Une autre grande difficulté a été de choisir une fonction Fitness permettant d'évaluer l'approximation de la fonction. Même si je pense que ma fonction Fitness, qui calcule la moyenne des écarts entre les valeurs tests et les valeurs expérimentales, est une bonne manière d'évaluer l'approximation d'une fonction, il existe sûrement un moyen plus optimisé. En effet il est possible que certaines valeurs soient des valeurs "aberrantes", c'est-à-dire dont l'écart entre la valeur test et expérimentale est élevé. La moyenne des écarts est alors très légèrement impactée, mais la fonction n'est alors pas parfaitement approchée.

Exemple:
Sur ce graphique, le graphe bleu représente la fonction approchée et le graphe rouge la fonction théorique. Malgré deux valeurs aberrantes, la moyenne des écarts reste relativement faible, mais le graphe bleu n'est au final pas l'approximation optimale de la fonction théorique. En revanche les points du graphe vert sont en moyenne plus distancés mais ne prennent aucune valeur aberrante, ce qui fait du graphe vert une meilleure approximation de la fonction théorique.



J’ai également testé ma fonction fitness sur le sample 2, où M.Rodrigues trouve a=0.14, b=19, c=5. Sur le sample 2, mon algo retourne les valeurs 0.14, 19 et 2. J’ai alors fait un programme pour voir laquelle des 2 approximations se rapprochaient le plus des valeurs expérimentales, pour chaque i donné.

```
In [175]: TestApprox()
MARTIN 0.0019199345796437894
M.RODRIGUES 0.010747157873585378
M.RODRIGUES 0.01697840779409865
MARTIN 0.0052179236699796405
MARTIN 0.005174164369537992
MARTIN 0.011024677191982746
M.RODRIGUES 0.007806116578779276
MARTIN 0.0064171546801483625
M.RODRIGUES 0.0019973956102184998
M.RODRIGUES 0.0013335208744489346
M.RODRIGUES 0.010115329612612856
MARTIN 0.005731767241349162
MARTIN 0.002901116658317715
MARTIN 0.00033444481412603366
M.RODRIGUES 0.02108371156333616
MARTIN 0.0067826884188249
MARTIN 0.0036798335924899117
MARTIN 0.0007608653703937707
MARTIN 0.0028181570653686983
MARTIN 0.011375891492820145
/////
SCORE FINAL: MARTIN 13 - 7 M.RODRIGUES
```

Sur les 20 valeurs, mon approximation est plus proche pour 13 valeurs, cependant on observe que lorsque M.Rodrigues a une meilleure approximation, l’écart est plus grand.

On retrouve donc le cas cité précédemment à l’aide du graphique: Ma fonction approchée approche très bien la valeur théorique pour un certain intervalle, mais est moins bien approchée sur un ou plusieurs petits intervalles.

Pour avoir une approximation optimale, il aurait donc fallu que je prenne en compte les écarts élevés et leur applique un coefficient dans le calcul de la moyenne.