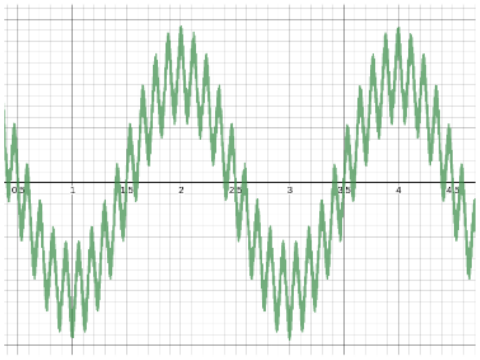
IA : approximation de fonction avec un algorithme génétique

Rapport

Grégoire CAURIER TD B

1 Problème

Des astronomes ont observés une nouvelle étoile inconnue. Leurs premières observations laissent penser que la température à sa surface varie de façon très régulière avec plusieurs périodicités comme illustré ci-dessous.



Les astronomes supposent que l'étoile suit une fonction dîte de Weierstrass[1] avec plusieurs périodicités imbriquées au sens fractale que l'on définie comme suit:

$$t(i) = \sum_{n=0}^{c} a^n \times cos(b^n \pi i)$$

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

$$A = \{\ a \in \mathbb{R} \mid a \in\ ]0, 1[\ \}$$
$$B = \{\ b \in \mathbb{N} \mid b \in\ [1, 20]\ \}$$
$$C = \{\ c \in \mathbb{N} \mid c \in\ [1, 20]\ \}$$

L'objectif est de déterminer la combinaison (a,b,c) capable de rapprocher la fonction de Weierstrass au plus près des observations de températures effectuées à des instants données. Une liste d'observations vous est fournie dans le fichier temperature_sample.csv où chaque ligne représente une observation. Chaque observation est composée de l'instant de l'observation ainsi que de la température observée.

[1] https://en.wikipedia.org/wiki/Weierstrass_function

Notre mission était donc de déterminer la meilleure combinaison (a,b,c) possible de telle manière de s'approcher le plus des valeurs de température récolter dans les échantillons par le biais d'un algorithme génétique. J'ai choisi de coder cet algorithme en python, quand bien même je suis plus à l'aise et préfère le c#, je disposais de plus de ressources avec le module Langage Python pour pouvoir le réaliser. La dernière version de mon algorithme fait un peu plus d'une centaine de ligne. J'explique dans ce rapport mes différents choix.

- Concernant la taille de mon espace de recherche, j'ai choisi de former une population initiale de 200 à 300 individus individus (taille conseillée par monsieur Rodrigues durant le cmo). Cette taille permettait de générer pas mal d'individus possibles différents puisque le choix

d'un individu pouvait se faire dans un espace de recherche de 20x20x10^2(car j'arrondissais a à 2 chiffres après la virgule avec round()) soit 40 000 combinaisons.

- Ma fonction de fitness afin d'évaluer la qualité des individus de mon espace de recherche se fait en plusieurs étapes. J'ai d'abord modélisé la fonction de Weierstrass avec une fonction nommée 't'qui prends en paramètre l'abscisse i et les 3 paramètre a, b et c. La somme est modélisée par une boucle for et une valeur t(i) qui s'incrémente à chaque tour. J'ai vérifié cette modélisation sur une calculatrice.

  J'ai défini ensuite une autre fonction appelée « evaluer » qui pour chaque triplet de ma population :
  -va prendre chaque valeur de i du fichier sample
  -pour chaque valeur de i : calcul l'écart en valeur absolue entre la valeur de la fonction au point i pour le triplet (a,b,c) et la valeur théorique donnée dans le fichier.
  -va faire la somme de tous les écarts obtenus.
  => la valeur obtenue est la qualité de l'individu (a,b,c) que je range dans une liste Fitness
  Plus la valeur obtenue est petite plus l'individu est de qualité.
  -Je répète l'opération pour toute ma population.

- Afin d'augmenter mon espace de recherche, j'ai mis en place des opérateurs sur ma population initiales :
  1) Pour l'opérateur de croisement je choisissais au hasard 2 individus dans toute ma population et je réalisais le maximum de croisement possibles entre eux soit 3*2=6. Je créais un nouvel individu avec le paramètre a du premier parent et le b et c du deuxième parent par exemple. Cet opérateur que je répétais autant de fois qu'il y avait d'individu dans ma population initiales me permettait d'optimiser mes recherches puisque je créer (6*le nombre d'individus de départ) à chaque appel de ma fonction.
  2) Pour l'opérateur de mutation, je choisissais au hasard dans ma population et je remplaçais au hasard un de ses 3 paramètres. J'ajoutais ensuite ce nouvel individu à ma population. Je répétais cette opération autant de fois que j'avais d'individus dans ma population. Cet opérateur me permettait de doubler ma population à chaque appel.
- La taille de ma population de départ est de 200 à 300 individus. C'est un nombre qui me permet d'obtenir un grand nombre d'individus à évaluer. Avec mes opérateurs de croisement et de mutation j'obtenais une population dans les alentours de 1500 individus à tester. A chaque génération je sélectionnais les meilleurs individus en triant ma population et ma liste de fitness simultanément dans l'ordre croissant (tri à bulles vu en langage python) afin de reproduire les individus avec les meilleures qualités entre eux générations d'après. Mon algorithme converge plutôt rapidement vers une solution stable dès la 3ème génération grâce à cette méthode.
- A l'aide de la bibliothèque time j'ai initialisé un timer dans mon programme afin de mesurer le temps qu'il prenait pour trouver une solution stable. En général, l'algorithme trouve une solution stable en 5 secondes pour un temps total d'exécution environ de 14 secondes pour 10 générations.
- J'ai réalisé plusieurs solutions avant d'arriver à une version finale. Dans les premières versions je négligeais le besoin d'une sélection des meilleurs individus pour les reproduire entre eux. Mes algorithmes étaient plutôt lents à exécuter puisque la population était de plus en plus grande ce qui serait devenu problématique si je voulais faire un plus grand nombre de générations. Pour l'opérateur de croisement j'avais d'abord pensé à l'exemple de

la caserne de pompier du td1 en intervertissant a,b et c de chaque attribut mais cette technique ne permettait pas d'obtenir d'individus avec plus de potentiel puisque a n'avait pas le même domaine de définition que b et c. Enfin j'ai voulu optimiser mon opérateur de croisement en étant sûr de ne jamais croiser un individu avec lui-même (optimisation minime puisque l'on a 1/(taille de départ) chance de croiser un individu avec lui-même).