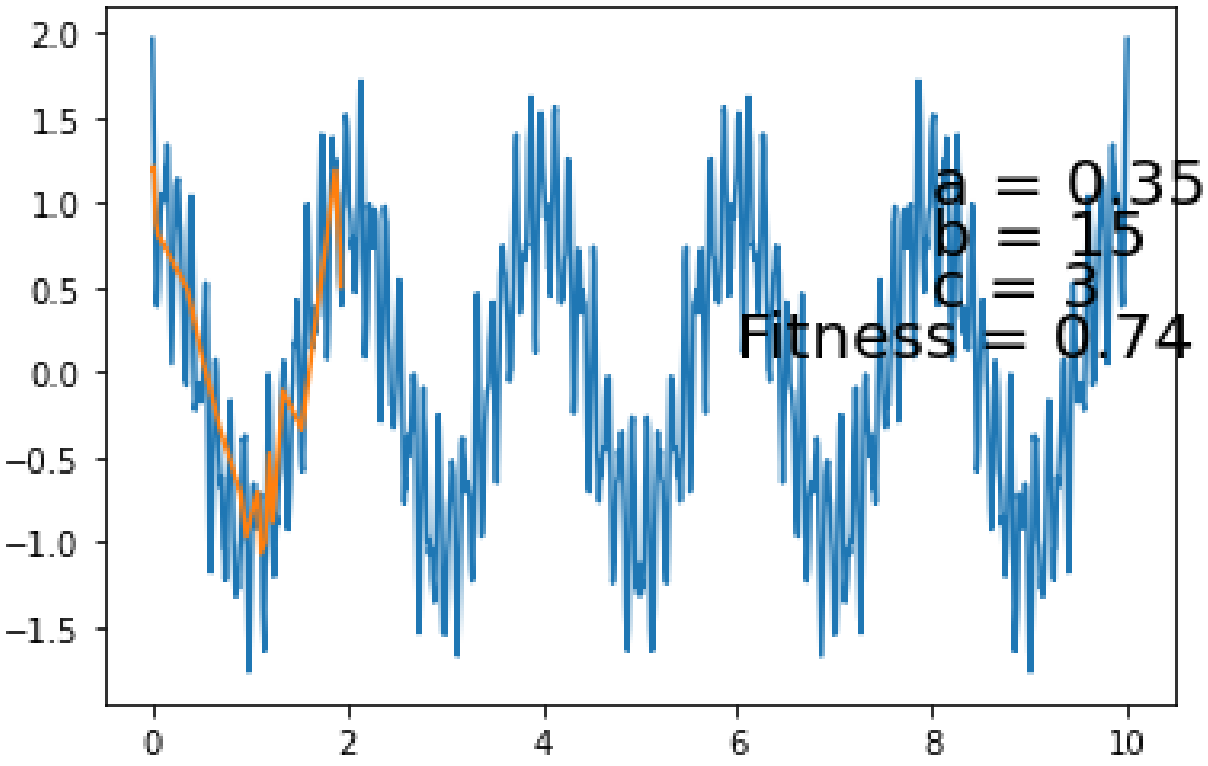
Rapport algorithme génétique

Quentin Kaufman

2020/2021



Introduction

Dans le cadre du cours d'intelligence artificielle de 3ème année à l'esilv. J'avais pour but de coder un programme capable de retrouver des coefficients d'une fonction. La fonction dite de *Weierstrass*. Le programme devait fonctionner sur le principe de l'algorithme génétique. Qui consiste à sélectionner et croiser les meilleurs individus d'une population de départ pour avoir le meilleur individu (ici les individus sont des triplets de 3 coefficients).

Le vif du sujet

L'algorithme génétique (et tous les algorithmes d'intelligence artificielle en général) est utilisé lorsque qu'il est trop compliqué de résoudre un problème mathématiquement, notamment de par le coup en ressources machine necessaire à la résolution du problème. En effet le problème posé est de trouver les coefficients a, b et c tels que : $A = \{ a \in \mathbb{R} \mid a \in]0,1[\}$ $B = \{ b \in \mathbb{N} \mid b \in [1,20] \}$ $C = \{ c \in \mathbb{N} \mid c \in [1,20] \}$

Quand on multiplie les intervalles en eux, on remarque que le nombre monte très vite. Surtout que l'intervalle de A est continu, donc un nombre infini de possibilités.

On se rend donc rapidement compte qu'il ne sera pas possible de tester tous les triplets et de repérer celui qui rempli le mieux les critères. En parlant de critères !

Comment noter un individu ? Comment peut-on savoir si un individu est meilleur qu'un autre ? Il faut trouver un moyen de noter les individus pour pouvoir les classer. C'est ce qu'on appelle la *fitness* de l'anglais fit « ajuster ». C'est une fonction que l'on retrouve dans tous les algorithmes génétiques qui permet de sélectionner un individu plutôt qu'un autre. Elle est très importante car c'est elle qui sélectionnera les individus. Ma fonction fitness est plutôt simple. C'est la somme des valeurs absolues des différences entre l'ordonné de la fonction de Weierstrass et l'ordonnée lu dans le fichier échantillon (j'avais dit simple, mais c'est tout de même une phrase compliquée ! Voici donc le code)

```
def fitness(ind):
    fitTot = 0
    for i in range(len(temps)):
        fit = abs(W2(temps[i],ind)-temperature[i])
        fitTot += fit
    return fitTot
```

C'est une somme des écrats aux échantillons pour faire plus simple. Cela nous renvoie une valeur décimale, cette dernière doit être la plus petite possible pour avoir une fonction qui suit au maximum les échantillons.

La fitness m'a donnée un peu de fil à retordre, surtout pour l'implémenter en Python, j'ai donc fait plusieurs essais de fonction fitness avant de coder quelque chose que je considère comme intuitif et qui fonctionnait avec ma fonction de Weierstrass déjà implémentée. Ci-dessous mes autres essais de fonction fitness (non retenus dans le code final).

```python
def fitness(ind):
    tempTheory = map(W,temps)
    out = np.vectorize(W)
    tempExp = temperature
    #tempTheory = out(temps,ind)
    result = sum(abs(tempTheory-tempExp))
    return result

def fitness2(ind):
    truc = lambda x: W2(x,ind)
    tempTheory = map(truc,temps)
    tempExp = temperature
    result = sum(abs(tempTheory-tempExp))
    return result

def fitness3(ind):

    tempTheory = [W2(t,ind) for t in temps]
    tempExp = temperature
    result = sum(abs(tempTheory-tempExp))
    return result
```

L'algorithme génétique ressemble comme on l'a vu en tout points à l'évolution d'une espèce d'Êtres vivants. Cela signifie qu'il faut ajouter le caractère des croisements du patrimoine génétique. Il y a aussi une partie de mutation des individus qui peut arriver, on doit donc l'implémenter. Les mutations peuvent paraitre négligeables, mais elles sont très importantes, tant dans le règne vivant que dans notre algorithme. Elles fournissent une diversité génétique et peuvent faire apparaitre des caractères que l'on n'aurait pas eu par simples croisements. On aurait atteint un « maximum local », c'est quand il n'y a plus assez de brassage génétique et que l'évolution stagne.

Ci-dessous ma fonction de croisement

```python
def croisement(ind1,ind2):
    ind3 = ind1.copy()
    ind3[0]  = ind2[0]
    ind4 = ind2
    ind4[0]  = ind1[0]

    ind5 = ind1.copy()
    ind5[1]  = ind2[1]
    ind6 = ind2
    ind6[1]  = ind1[1]

    ind7 = ind1.copy()
    ind7[2]  = ind2[2]
    ind8 = ind2
    ind8[2]  = ind1[2]
    return [ind3,ind4,ind5,ind6,ind7,ind8]
```

On remarque que le croisement ne se fait pas de manière aléatoire, je fais un simple échange bidirectionnel des coefficients. J'ai essayé d'appliquer du hasard à ma fonction de croisement. Malheureusement cela devenait compliqué et il y avait pleins d'exceptions à gérer. J'aurais aussi pu échanger plusieurs paramètres, mais cela devenait compliqué de gérer toutes les combinaisons linéaires, à partir de 2 individus j'en génère déjà 6, c'est déjà beaucoup. C'est néanmoins un point améliorable car si le brassage se fait lentement, on mettra plus de temps à obtenir une solution.

Ci-dessous ma fonction de mutation :

```python
def mutation(ind):
    param = random.randint(0,2)
    a = round(random.uniform(0,1),2)
    b = random.randint(1,20)
    c = random.randint(1,20)
    if(param == 0):
        ind[0] = a
    elif(param == 1):
        ind[1] = b
    else:
        ind[2] = c
    return ind
```

La mutation elle a été bien mieux implémentée car comme vous pouvez le voir on choisit aléatoirement le paramètre à modifier, et on calcul aléatoirement les coefficients qui vont se faire remplacer. On aurait même pu choisir aléatoirement le nombre de paramètres à modifier (ici un seul paramètre est modifié à la fois), pour avoir une mutation la plus aléatoire possible, mais ce résultat est déjà satisfaisant et je manquais de temps.

Ci-dessous al fonction de sélection :

```python
def selection(pop, hcount, lcount):
    liste = pop[:hcount].copy()+ pop[len(pop)-lcount:len(pop)].copy()
    return liste
```

La fonction de sélection est pour le coup très simple, on créé une liste qui contient un nombre défini des meilleurs et des pires individus pour une génération. Cela permet de faire des croisements plsu ciblés et non sur une population gigantesque. On fait de la sélection artificielle.

Après cela on crée une population finale constituée des parents, des enfants, des enfants avec mutation et de nouveaux individus créés aléatoirement (pour rajouter encore un peu de hasard). Cela fait une grande population qui dépend qui dépend de pleins de paramètres.

```python
endPop = popSelect + popCroise + popMute + newPop
```

Après avoir fait la boucle, on commence à avoir des résultats.

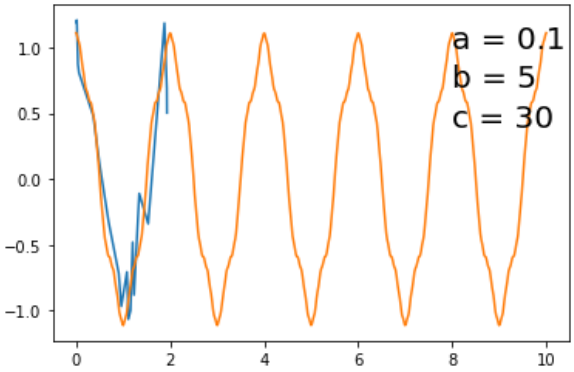
J’ai obtenu des résultats plutôt étonnants, en effet, j’arrive à trouver un individu qui remplit presque parfaitement les critères, mais sans avoir une convergence de la fitness. Plusieurs problèmes sont possibles :

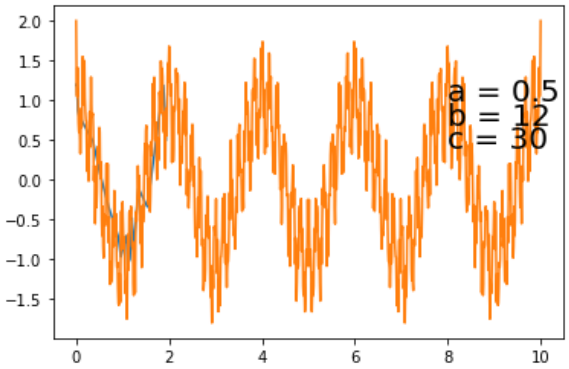
- Les croisements se font mal donc le patrimoine génétique des meilleurs n’est pas conservé
- Il y a trop de mauvais individus gardés à chaque génération ce qui « plombe la moyenne »
- Il y a trop de mutations et d’individus créés au hasard qui eux aussi plombent la moyenne
- Un mélange des trois

J’ai éliminé l’hypothèse d’une fonction fitness qui ne fonctionne pas car je l’ai bien plus testée que les autres et tous les résultats semblent correspondre.

Je ne converge donc pas vers une valeur précise, mais je pense qu’il y a quand même suffisamment de brassage et beaucoup de mutations qui me permettent de trouver assez rapidement une valeur précise. Pour un run j’ai par exemple la valeur de 730 générations

Je me suis beaucoup aidé des graphiques pour visualiser les fonctions, et quels étaient les impacts des paramètres.





J’ai aussi créé une liste des meilleurs individus de chaque génération que je triais ensuite pour voir quels étaient les meilleurs de tous.

```
#ind1 = [0.1,10,3]
#print(fitness(ind1))
a = createRandomCoeff(15)
#affichePop(a)
print()
b=evaluate(a)
#affichePop(b)
print()
c=selection(b, 2, 1)
#affichePop(c)
ind1 = [0.1,10,1]
ind2 = [0.2,20,2]
#affichePop(croisement(ind1, ind2))
#afficheInd(mutation(ind1))
```