RAPPORT ALGORITHME GENETIQUE

Espace de recherche :

Pour résoudre ce problème et déterminer s'il est intéressant d'utiliser un algorithme génétique (au lieu d'un algorithme « force brute » par exemple), nous devons déterminer l'espace de recherche :

Ici nous avons trois paramètres : « a », « b », « c ».

« b » et « c » sont défini sur l'espace N compris entre 1 et 20 ce qui fait 20 possibilités pour chacun de ces paramètres.

« a » quant à lui est défini sur l'espace R compris entre 0 et 1 ce qui théoriquement nous donne un espace de recherche infini si on ne considère pas de précision maximal. Pour mon code « a » est codé au millième près.

Cela nous fait donc un espace de 20*20*1000=400 000 possibilités comprises dans notre espace de recherche.

Fonction Fitness :

Pour sélectionner les différents individus [a,b,c] d'une population il est nécessaire de les évaluer via une fonction fitness.

Pour ce problème ma fonction fitness est la somme des écarts relatifs entre la valeur de l'individu et la valeur réelle.

$$F = \sum_{i=1}^{N} |w_i(a,b,c) - w_i(a',b',c')|$$

N = nombres de points fournis

W_i = fonction de Weierstrass pour le point i

(a,b,c) = valeur réelle de la fonction que l'on cherche

(a',b',c') = valeur de l'individu à évaluer

Implémentation en Python :

```
def weierstrass(self,i):
  result=0
  for n in range(self.c+1):
    ite = (self.a**n)*(cos((self.b**n)*pi*i))
    result = result+ite
  return result
def distance(self,index,data): #valeur absolue de la distance entre la valeur trouvée et la réalité
  return abs((self.weierstrass(data.iloc[index][0])-data.iloc[index][1]))
def fitness(self,data): # somme des distances (fonction de coût)
  distance_totale=0
  for index in range(len(data)-1):
    distance_totale += self.distance(index,data)
  return distance_totale
```

Opérateurs :

L'algorithme génétique nécessite des opérateurs qui vont permettre d'évaluer, de sélectionner, de muter nos individus.

1) Mutation

Pour apporter un maximum de diversité nous faisons muter des individus de la population. Cela se traduit concrètement par une modification aléatoire de leurs paramètres [a,b,c]. L'augmentation ou la diminution est choisie aléatoirement. Cela peut paraître excessif de modifier l'ensemble des valeurs mais j'observais un fort phénomène d'eugénisme sinon.

```
def mutationbis(c1):
  a=random.randrange(2)
  result=individu()
  if (a==1):
    result = individu(round((c1.a-0.05),3),c1.b+1,c1.c-1)
  if (a==0):
    result = individu(round((c1.a+0.05),3),c1.b-1,c1.c+1)
  if (result.b<1 or result.b>20):
    result.b=random.randint(1,20)
  if (result.c<1 or result.c>20):
    result.c=random.randint(1,20)
  if (result.a<0 or result.a>1):
    result.a=round(random.uniform(0,1),3)
  return result
```

2) Croisement

Il est nécessaire de réaliser des croisements entre les différents individus pour obtenir une nouvelle population avec possiblement des meilleurs résultats. Les variables « b » et « c » sont définis dans le même espace c'est pourquoi j'ai décidé d'inverser leurs valeurs pour générer les nouveaux individus. En ce qui concerne la variable « a », plusieurs options étaient possibles pour croiser deux individus. J'ai opté pour la moyenne des deux parents pour le premier individu et une inversion des deux premières décimales pour le deuxième individu. Ainsi 0.95 devient 0.58 avec ce traitement

```
def croisementbis(c1,c2):
  c3=individu(round(((c1.a+c2.a)/2),3),c1.b,c2.c)
  new_a  = eval(str(c1.a)[:-2]+str(c1.a)[-1]+str(c1.a)[-2])
  if (new_a>1):
    new_a = round(random.uniform(0.01,0.99),3)
  c4=individu(new_a,c2.b,c1.c)
  return c3,c4
```

Sélection des individus :

Une fois évalués et triés selon leurs résultats, les différents individus vont être sélectionnés par le même principe que la sélection naturelle.

Ceux avec les meilleurs résultats seront gardés plus une petite partie de « mauvais » éléments qui apporteront de la diversité nous évitant de tomber dans de l'eugénisme.

```
def evaluatebis(pop):
  return sorted(pop,key=lambda individu : individu.distance_reel)

def selectionbis(pop,n_best,n_worst):
  return pop[:n_best]+pop[len(pop)-n_worst:]
```

J'ai arbitrairement choisi des coefficients d'acceptation des meilleurs et des pires. Dans mon code final : n_best correspond à 35% des « meilleurs » candidats et n_worst au 15% les plus « mauvais ».

Taille de population et nombre de générations :

Une fois tous les opérateurs définis il est nécessaire d'implémenter une boucle qui les répétera le nombre de générations souhaités. Des paramètres doivent aussi être défini par tâtonnement comme la taille de la population et les générations. Après plusieurs tests j'ai opté pour une population de départ de 10 individus et 20 générations maximum.

Pour une précision abordable il faut 5-10 générations pour converger.

A la suite d'une génération la population augmente légèrement ce qui peut diminuer la vitesse du programme sans pour autant apporter des bonnes solutions. Si la population dépasse un certain nombre, on ne garde que les 30 meilleurs individus.

Il n'est pas rare de tomber sur une bonne solution avant l'ensemble des générations mais l'algorithme peut d'abord tomber dans un minimum local. Pour contrer cela, si la moyenne des fitness des meilleurs éléments ne dépasse pas une certaine valeur (soit un minimum local) on regénère une nouvelle population. C'est pourquoi il y a autant de générations car il faut laisser le temps à la nouvelle population le temps de converger.

Temps d'exécution :

Pour une distance globale de 1 (soit 0,05 de différence de moyenne sur les 20 points), le programme a une moyenne de : 400ms (250ms s'il trouve tout de suite et jusqu'à 2 secondes en cas de contre-temps dans un minimum local).

Pour une distance globale de 0.2 (soit 0,01 de différence de moyenne sur les 20 points), le programme a une moyenne de : 3 secondes (250 ms s'il trouve dans les 3 premières générations et jusqu'à 5 secondes pour les pires cas).

Ces temps d'exécution et précisions sont issues du deuxième set de données.

Différentes solutions envisagées

Lors de la réalisation de ce projet j'ai rencontré certaines difficultés.

1) Premièrement il a fallu définir les bons critères de croisement et précisément celui de « a ». En effet il fallait assurer une grande diversité : la moyenne a tendance à converger trop rapidement donc le deuxième individu était d'autant plus essentiel. Premièrement je lui associais la valeur d'un parent mais je tombais dans de l'eugénisme. J'ai donc opté pour l'inversion des valeurs des décimales.

2) Deuxièmement j'ai rencontré des difficultés avec ma boucle final. J'ai d'abord commencé par développer une boucle qui agissait comme une compétition. Le principe était de partir avec une population très importante (2 000 environs) puis de sélectionner les meilleurs éléments, les croiser etc… jusqu'à arriver à un individu unique. Malheureusement, je me retrouvais avec une population trop uniforme qui donnait des résultats en demi-teinte et une vitesse d'exécution trop faible. J'ai donc revu mon modèle en optant pour une population bien plus faible mais stable en nombre.

3) Enfin, j'ai passé un certain temps à déterminer les différents coefficients « arbitraires » comme les pourcentages d'individu à garder lors de la sélection. Je pouvais par exemple sous-estimer la population de « mauvais » individu à garder.

Approche du bruit

J'ai remarqué que selon les sets de données fournis, mon algorithme n'arrivait pas à s'approcher de la valeur initiale avec la même précision. Pour le deuxième je n'ai aucun problème à trouver des valeurs extrêmement similaires de l'ordre d'un centième de différence. Les résultats réels sont : 0,14 19 et 5 et j'obtiens à multiple reprises : 0,139 19 8. En revanche je n'arrive pas à approcher cette précision pour le premier set.

Les données peuvent être plus ou moins bruitées dans ce cas ou dans la réalité ce qui rend difficile le choix arbitraire d'une précision optimale.

Pour y remédier j'ajoute une dernière boucle qui va continuer de tourner tant que la précision maximale n'est pas atteinte. Je l'initialise très haut et la décrémente tant que l'algorithme trouve des solutions plus précises.

```
precision=2 #précision de départ
stop_def=False #stop définitif du programme
stop_compteur=0
winner_list=[]
while (stop_def==False):
  result,win=boucle_finalv2(precision)
  if (win==True): #précision battue on peut continuer à minimiser notre fonction de fitness
    precision -=0.2
    stop_compteur=0
    winner_list.append(result)
  else:
    stop_compteur+=1
    if (result.distance_reel <evaluatebis(winner_list)[0].distance_reel): # cas où on ne bat pas la précision mais on s'en approche
      winner_list.append(result)
  if (stop_compteur==3): #trois boucles sans amélioration on arrête la recherche
    stop_def=True
print("Precision atteinte :",precision )
print("Best : ",evaluatebis(winner_list)[0])
print("Fitness : ",evaluatebis(winner_list)[0].distance_reel)
```