


Algorithme Génétique

Matthieu SEGUI

 [Github du projet](#)



1 Plan d'action

1.1 Quelle est la taille de l'espace de recherche (utiliser une notation scientifique) ?

Les paramètres ne sont pas obligatoirement des entiers. Il sont compris dans un espace allant de $[-100,100]$. De ce fait, il existe une infinité de solutions. Cependant, si nous raisonnons dans le cadre d'un programme fait sous python, alors le type "float" a une limite. Il y a en effet 14 chiffres possibles après la virgule. Nous pouvons donc écrire que le nombre de possibilités est :

$$(200 * 10^{14})^6$$

1.2 Quelle est votre fonction fitness ?

Ma fonction fitness effectue la somme des distance euclidiennes entre ma fonction issue de mes paramètres et l'ensemble des points de position_sample.csv ;

Voici ma méthode de fitness :

```
1 def Fitness(self): #Fitness that sum euclidian distances
2     distance = np.zeros(self.ind)
3     for i in self.training_data :
4         x = self.p1 * np.sin(self.p2*i[0] + self.p3)
5         y = self.p4 * np.sin(self.p5*i[0] + self.p6)
6         distance += np.sqrt((i[1]-x)**2 + (i[2]-y)**2)
7     self.fitness = distance
```

1.3 Décrivez les operateurs mis en œuvre (mutation, croisement) ?

Pour ma mutation, je mute 0 à 6 paramètres des 30% des individus les plus intéressants de ma population en leur ajoutant une valeur aléatoire suivant une loi normale centrée en 0 et d'écart-type σ .

L'écart-type correspond à la variable "par". Voici ma méthode de mutations :

```
1 def Mutations(self): #Mutation method
2     delta=100
3     #Normal law parameter.
4     par = 0.1 #Working at 0.1
5     #Selection rate parameter
6     rate = 0.15 #Working at 0.15
7     upper_percent = 0.3
8     lower_percent = 0.0
9     lower_bond = int(self.ind*lower_percent)
10    upper_bond = int(self.ind*upper_percent)
11    size_sample = upper_bond - lower_bond
12    temp = np.random.random(size_sample) #Randomizing array of float values between 0
    and 1
13    temp = temp<rate #Transforming it into an array of booleans
14    #Applying it to randomize the mutations
15    self.p1[lower_bond:upper_bond] = np.clip(self.p1[lower_bond:upper_bond] + (temp *
    np.random.normal(0,par,size=size_sample)), -delta, delta)
16    temp = np.random.random(size_sample)
17    temp = temp<rate
18    self.p2[lower_bond:upper_bond] = np.clip(self.p2[lower_bond:upper_bond] + (temp *
    np.random.normal(0,par,size=size_sample)), -delta, delta)
19    temp = np.random.random(size_sample)
20    temp = temp<rate
21    self.p3[lower_bond:upper_bond] = np.clip(self.p3[lower_bond:upper_bond] + (temp *
    np.random.normal(0,par,size=size_sample)), -delta, delta)
22    temp = np.random.random(size_sample)
23    temp = temp<rate
```

```

24 self.p4[lower_bond:upper_bond] = np.clip(self.p4[lower_bond:upper_bond] + (temp *
    np.random.normal(0,par,size=size_sample)), -delta, delta)
25 temp = np.random.random(size_sample)
26 temp = temp<rate
27 self.p5[lower_bond:upper_bond] = np.clip(self.p5[lower_bond:upper_bond] + (temp *
    np.random.normal(0,par,size=size_sample)), -delta, delta)
28 temp = np.random.random(size_sample)
29 temp = temp<rate
30 self.p6[lower_bond:upper_bond] = np.clip(self.p6[lower_bond:upper_bond] + (temp *
    np.random.normal(0,par,size=size_sample)), -delta, delta)

```

Pour mes croisements, j'échange les paramètres des 30% des individus suivants entre eux (c'est-à-dire les p1 avec les p1...). J'échange de manière aléatoire en spécifiant un poids de croisement qui avantage les individus les plus intéressants.

Voici ma méthode de croisements :

```

1 def Cross(self) : #Crossover's function
2     upper_percent = 0.6
3     lower_percent = 0.3
4     upper_bound = int(self.ind*upper_percent)
5     lower_bound = int(self.ind*lower_percent)
6     upper = int(self.ind*(0.2))
7     lower = 0
8     size_sample = upper_bound - lower_bound
9     stock = 1/((np.arange(lower,upper)+1)) #Weight to make interesting individuals
    cross more often in order to have better children
10    s = np.sum(stock)
11    keys = np.random.choice(np.arange(lower,upper),size_sample,replace= True, p=stock/
    s) #Random key to achieve the crossover
12    self.p2[lower_bound:upper_bound] = np.take(self.p2,keys) # Apply the keys found
13    self.p1[lower_bound:upper_bound] = np.take(self.p1,keys)
14    self.p3[lower_bound:upper_bound] = np.take(self.p3,keys)
15    keys = np.random.choice(np.arange(lower, upper),size_sample,replace= True, p=stock
    /s)
16    self.p4[lower_bound:upper_bound] = np.take(self.p4,keys)
17    self.p5[lower_bound:upper_bound] = np.take(self.p5,keys)
18    self.p6[lower_bound:upper_bound] = np.take(self.p6,keys)

```

1.4 Décrivez votre processus de sélection.

Comme expliqué précédemment, je génère une première population de manière aléatoire, puis je la trie en fonction de la fitness. Ensuite, je mute les 30 premiers pourcent, je croise les 30 suivants. Pour les 40 derniers, je regenère une nouvelle population d'individus aléatoires. Je trie cette nouvelle population et je recommence.

1.5 Quel est la taille de votre population, combien de générations sont nécessaires avant de converger vers une solution stable ?

Ma population comporte 10000 individus. Entre 80 et 150 générations sont nécessaires avant de converger vers une solution stable.

1.6 Combien de temps votre programme prend en moyenne ?

Mon programme prend en moyenne 20ms par générations pour une population de 10000. Donc, je trouve la solution convergente en 2.5s de moyenne.

1.7 Si vous avez testé différentes solutions qui ont moins bien fonctionnées, décrivez-les et discutez-les

Tout d'abord, si vous voulez vous voir mes essais, vous pouvez vous rendre sur le GitHub (cliquable).

J'ai testé 3 IA différentes. En effet, à l'origine j'étais parti sur une IA en utilisant des listes de listes et en utilisant les fonctions de python. J'effectuais mes mutations et croisements sur l'ensemble de ma population mais je stockais les résultats dans une autre list, puis je comparais et gardais les individus intéressants dans ma liste principale. L'IA convergeait et atteignait une valeur intéressante. Cependant, la convergence était lente et je n'étais pas convaincu. J'ai donc attaqué la création d'une nouvelle IA dans le même raisonnement mais cette fois-ci en utilisant la librairie numpy. En effet, celle-ci est beaucoup plus optimisée et puissante que les list classiques lorsqu'on apprend à s'en servir. Cependant, cette IA ne convergeait pas ou alors par de manière certaine. J'ai donc finalement repensé mon algorithme et voici le résultat.

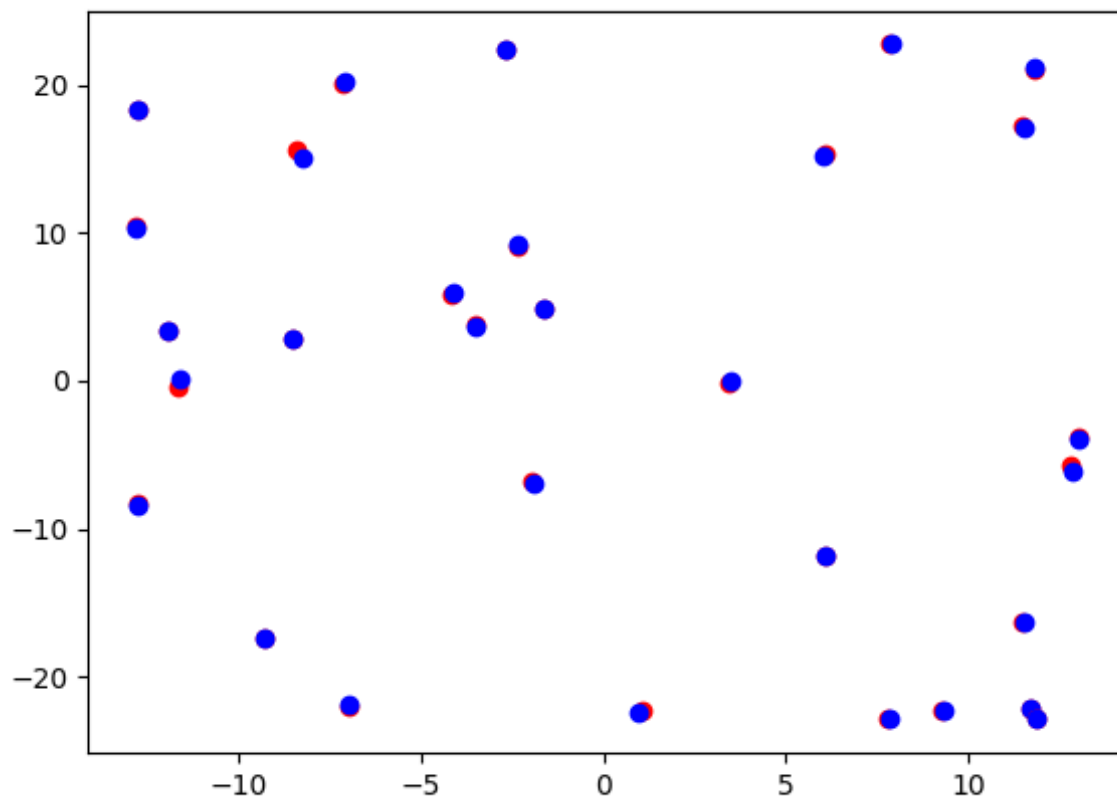


FIGURE 1 – Graphe final
Les points rouges sont les points de position_sample, les bleus mon résultat