# Recherche de l'architecture optimal

## Méthodes

### Objectif

Dans notre cas l'architecture optimal est l'architecture la plus robuste. Cette architecture ne doit pas changer de comportement lorsque des erreurs de fabrications ont lieu sur les dimensions des différentes barres. De plus il faut supprimer aux maximums les mouvements non-désirer comme un tangage ou un roulis de la plateforme central.

Objectif : trouver les longueurs nominales les plus adaptées pour avoir une architecture final robuste sans tangage et roulis.

Pour valider notre choix final il nous faut une grandeur objective pour comparer les différentes architectures possibles. A l'aide du programme de simulation pour un mécanisme a huit jambes nous pourrons relever les angles maximaux de roulis et de tangage de la plateforme centrale. Nous voulons donc réduire au maximum ces variations qui sont l'expression d'un tangage ou d'un roulis de la plateforme central.

Pour trouver une telle architecture nous allons utiliser un algorithme génétique.

### Algorithme génétique

L'algorithme génétique fonction comme le principe de l'évolutions, nous allons évaluer un grand nombre d'architecture et garder les meilleures. **Une fois évaluées les architectures seront associées à un score**. Nous avons maintenant une génération d'individus il faut maintenant la faire évoluer. Pour cela nous allons passer par quartes étapes :

**La sélection :**

Nous allons construire notre nouvelle génération (génération fils) sur les meilleurs éléments de la générations père. Ainsi les premiers individus de cette nouvelle génération fils sont les architectures qui ont reçu les meilleurs scores de la génération père**. On prend par exemple les 10% avec le score le plus élevé de la génération père.**

**Croissement :**

Pour remplir cette nouvelle génération de manière pertinente nous allons croiser les meilleurs éléments de la générations père. **Il faut donc prendre deux « codes génétiques » (ici les longueurs des barres) et en faire un troisième (A et B font C).** Pour faire ça plusieurs méthodes :

-Soit on fait la moyenne des deux codes sur chaque grandeur (chaque barre)

-Soit pour chaque partie du nouveau code on s'inspire soit de l'espèce A soit de l'espèce B

**Mutation :**

On peut aussi créer un nouvel individu à partir d'un seul autre individu. Pour faire muter cet individu **on modifie chaque partie de son code génétique par un taux de mutation** (5% par exemple). Dans notre cas cela revient à un petit peu changer une longueur nominale de barre.

**Injection :**

Pour éviter que la génération fils soit trop dépendante de la générations père, on **injecte également des individus avec un code génétique complétement défini aux hasards**. Ceci peut permettre d'apporter un caractère génétique majeur pour la génération fils que les meilleurs individus de la génération père n'avaient pas.

Si on reproduit cette idée un grand nombre de fois (pour plusieurs génération) on pourra trouver une solution optimale (au moins localement). On pourra influencer l'évolutions des générations en jouant sur le nombre d'individus par générations, la proportion d'individus sélectionnés, la proportion d'individus croisés ou la proportion d'individus mutés ainsi que le taux de mutation associé.

Après un grand nombre de génération il suffit de sélectionner le code génétique du meilleur élément de la plus jeune génération.

Un algorithme génétique fait évoluer les individus en fonction de la manière dont ont les évalues. Pour cela il faut que l'évaluation des différentes architectures soit en adéquations avec notre objectif.

## Score et algorithme de Monte-Carlo

Si notre évaluation est pertinente plus la solution sera robuste sans tangage et sans roulis et plus le score sera élevé. La solution sera robuste si le comportement ne change pas aux faibles variations, le problème c'est qu'il est impossible de vérifier le comportement pour toutes les petites variations. Il faut donc déterminer le comportement du mécanisme non pas pour des longueurs de tige fixée mais pour un ensemble de longueur de tige environnant certaines longueurs nominales de référence. **Nous allons donc étudier statistiquement la robustesse des architectures pour déterminer le comportement global d'une architecture (Principe de Monte-Carlo).** Nous avons deux variables :

**-Roulis : l'angle de roulis maximal prélever lors d'une simulation de l'architecture**

**-Tangage : l'angle de tangage maximal prélever lors d'une simulation de l'architecture**

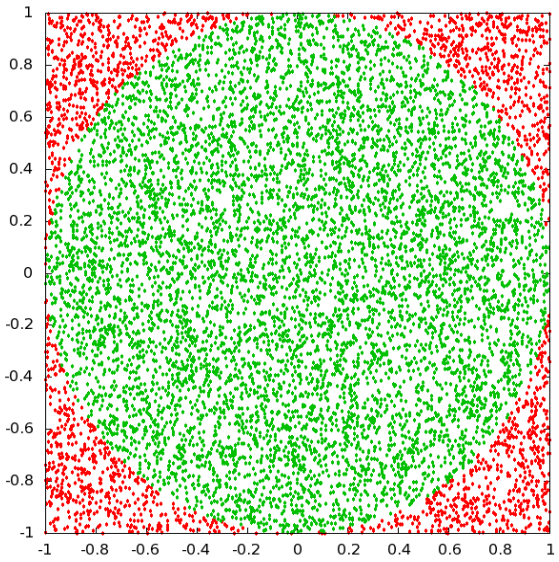
On lance la simulation un grand nombre de fois et on prélève les valeurs de ces deux variables. Pour chaque simulation les longueurs des barres de chaque jambe est redéfini dans l'intervalle d'erreur de fabrication. Plus le nombre de simulation sera grand plus le score sera pertinent. On peut maintenant avoir un score qui prend en compte la robustesse :

$$\mathbf{Score1} = \frac{\mathbf{1}}{\mathbf{1} + \boldsymbol{\sigma}(\mathbf{Roulis}) + \boldsymbol{\sigma}(\mathbf{Tangage})}$$

Plus l'architecture sera robuste plus les écarts-types seront faible et le score élevé. Il va falloir ensuite départager les solutions pour savoir lesquelles tangue et roule le moins toujours à l'aide de la simulation avec ce deuxième score :

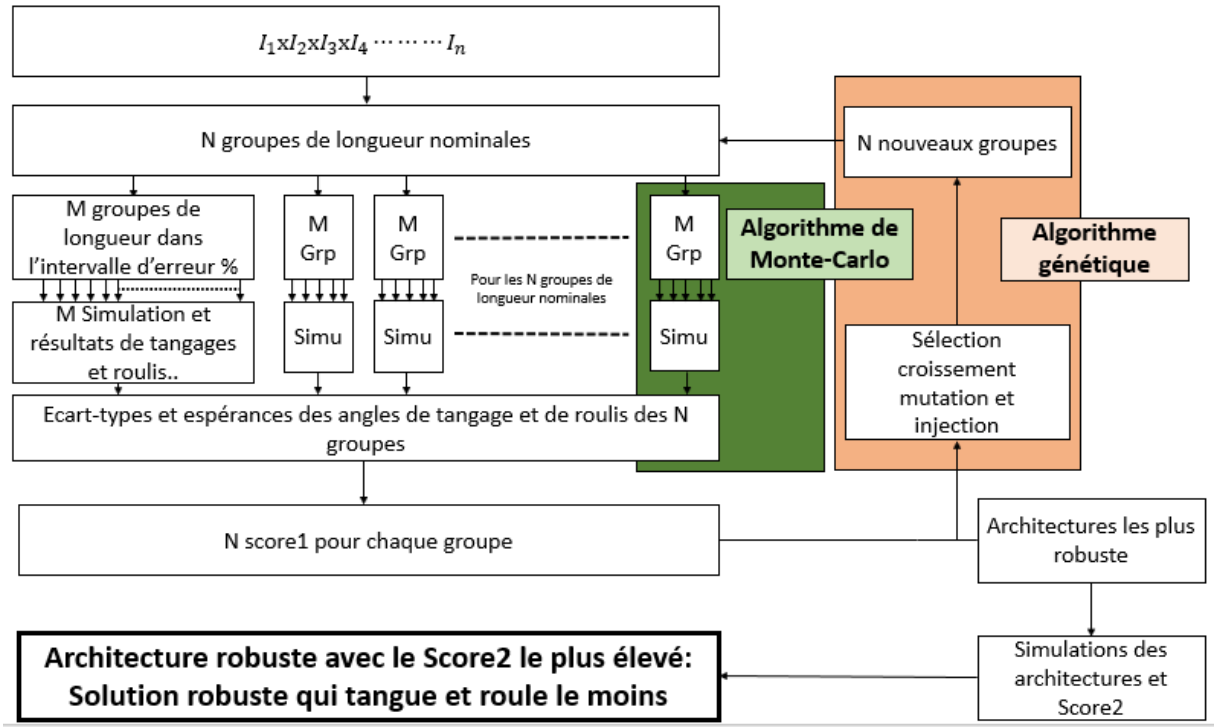
$$\mathbf{Score2} = \frac{\mathbf{1}}{\mathbf{1} + \mathbf{E}(\mathbf{Roulis}) + \mathbf{E}(\mathbf{Tangage})}$$

Après le passage de ces deux filtres nous aurons la solution robuste qui tangue et roule le moins.




*Approche statistique de la valeur de pi à l'aide de l'algorithme de Monte-Carlo*

## Structure algorithmique de la méthode de recherche de l'optimal




*Organigramme des algorithmes de la méthode de recherche de l'architecture optimal*

La recherche de l'architecture optimal se fait donc en deux parties d'abord déterminer les solutions robustes ensuite le score2 permet de trouver la solution robuste qui tangue et roule le moins. Maintenant que la méthode est proprement définie nous passons à l'écriture du programme sous Python.

## Structure du programme

Nous avons eu la volonté de créer un programme qui peut utiliser l'algorithme génétique pour n'importe quel problème et pas uniquement le nôtre. Le programme s'articule sur trois fonctions :

1) **minimize** : Fonction principal de l'algorithme génétique qui va réduire l'écart entre les images des individus et l'objectif.

Entrée :

-***list*** **:** les intervalles de recherche pour les longueurs des barres

-***function*** **:** la fonction de score

-***list*** **:** objectif

Sortie :

-***list*** : meilleur individu

**2) evolution :** *Cette fonction fait évoluer une génération père vers une génération fils.*

Entrée :

**- *list :*** génération fils = [dict : {codes génétiques : list}, dict : {scores : float}, int : numéro de génération]

Sortie :

**- *list*** : génération fils = [dict : {codes génétiques : list}, dict : {scores : float}, int : numéro de génération]

**3) Score :** Fonction qui détermine le score d'un individu d'une génération, cette fonction va utiliser la fonction simulation du programme qui modélise le mécanisme et qui prend en entrée des longueurs nominales de barre)

Entrée :

**- *function*** : Score1 ou Score2

**- *list*** : individu

**- *list*** : objectif

Sortie :

-***float*** : le score de l'individu

Pour pouvoir avoir une influence sur le type d'évolution des générations il y a **des Hyperparamètres** :

**-Ngen :** Nombre total de génération

**-Nind :** Nombre d'individu par génération

**-N_MC :** Nombre de simulation dans la fonction ***Score*** avec l'algorithme de Monte-Carlo.

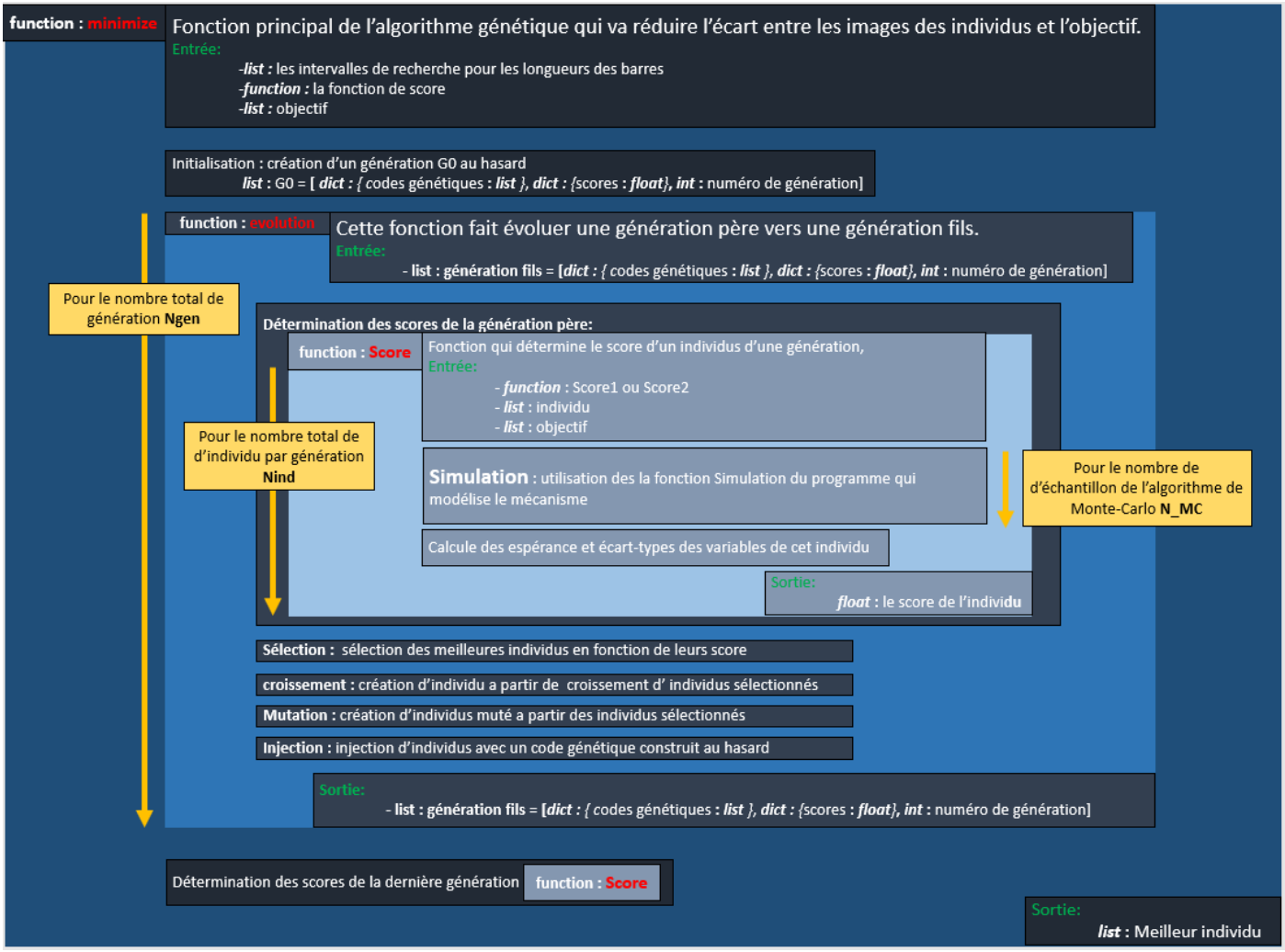
**-quantite_selectionne** : pourcentage de la population total (Nind) qui est sélectionné pour « survivre » jusqu'à la génération suivante

**-quantite_croissement, quantite_mute** : idem

**-taux_mutation** : taux de mutation des individus crées par mutation

**-intervalles** : intervalles de recherche sous forme de ***list*** pour les longueurs de barres

-**Sigma** : valeur qui représente l'erreur de fabrication qui aura une répartition normale ici.




*Structure du programme python pour la recherche de l'architecture optimale*

Il faut rajouter à ça les lignes de codes qui servent au suivie des données de la recherche (valeurs, courbes...)

Maintenant que le programme existe il faut s'assurer qu'il fasse le travail correctement.

## Validation du processus de recherche

Pour valider le processus de recherche nous l'avons d'abord utilisé sur une fonction mathématique polynomiale :

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} (x-3)^2 + (y-7)^2 \\ x + z - 3 \end{pmatrix}$$

Après ce premier essai nous avons corriger notre algorithme de tri des individus en fonction des scores. En effet dans certains cas particuliers d'égalité de score et un nombre d'individu par génération faible on se retrouvé avec des générations vides d'individus.

Pour mettre à l'épreuve le programme avec un problème un peu plus complexe nous avons travailler avec Antoine Amsaleg et son projet qui nécessiter de trouver le minimum d'une fonction escalier. C'est donc une fonction discontinue (contrairement à une fonction polynomiale) avec des parties constantes ce qui implique un grand nombre de points qui auront le même score.

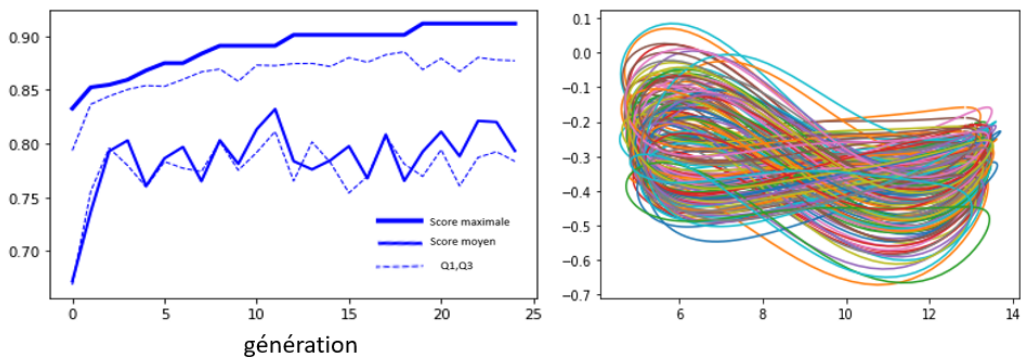
Les résultats du programme semblent être cohérent avec les recherche d'Antoine ce qui atteste de l'efficacité de l'algorithme même pour des problèmes relativement complexe comme le sien.

# Résultats

## Hyperparamètres et Interprétation

On lance notre première recherche avec sans mettre trop de génération ou d'individu pour voir comment le programme s'adapte au problème et surtout pour se rendre compte du temps que pourrais prendre les recherche (en termes de temps d'exécution du programme).

| Test 1 | |
|---|---|
| Ngen | 25 |
| Nind | 40 |
| Quantite_selectionne (%) | 20 |
| Quantite_croisse (%) | 30 |
| Qauntite_mute (%) | 20 |
| Taux_mutation | 0,1 |
| N_MC | 15 |
| Sigma | 0,02 |
| temps exécution (h) | 0,88666667 |




*Test 1 (sur la figure de droite les valeurs sont en centimètre)*

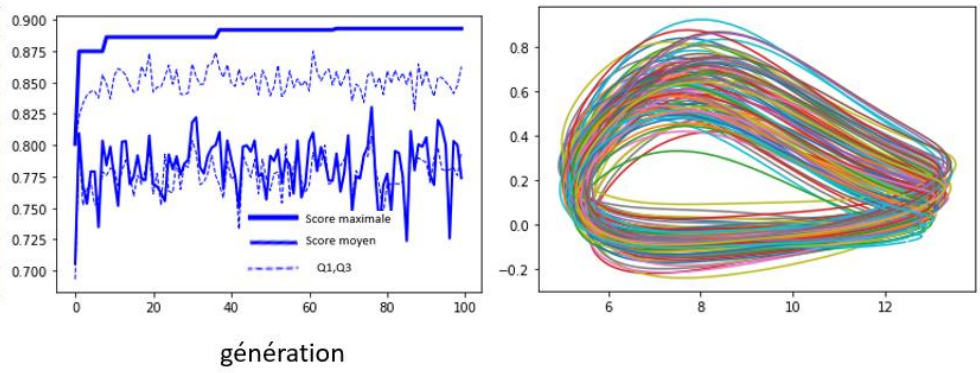
*Longueurs nominales = [1.21, 3.88, 4.20, 4.03, 3.78, 5.61, 3.90, 3.79, 6.57, 5.06, 4.93, 6.10, 0.85]*

Les scores affichés sur les graphiques seront ceux de score1 donc le score qui ne prend en compte uniquement la robustesse.

Pour savoir si la solution est robuste on regarde les déplacements des pieds du mécanisme sur un grand nombre de simulation (figure de droite). On peut comme cela constaté rapidement s'il y une instabilité aux petites variations.

Le premier retour sur ce test c'est que la solution n'est pas vraiment robuste puisqu'il y a une vraie dispersion entre les cycles de jambe. De plus le graphique ne montre pas vraiment que le score à fini de converger.il faut donc augmenter le nombre de simulation pour cela deux possibilités soit on augmente le nombre de générations soit le nombre d'individus par génération. Que ce soit l'un ou l'autre les calcules risque d'être très long puisque le test 1 a déjà pris pratiquement 1 heure.

| Test 2 | |
|---|---|
| Ngen | 100 |
| Nind | 40 |
| Quantite_selectionne (%) | 20 |
| Quantite_croisse (%) | 30 |
| Qauntite_mute (%) | 20 |
| Taux_mutation | 0,1 |
| N_MC | 15 |
| Sigma | 0,02 |
| temps exécution (h) | 3,88222222 |

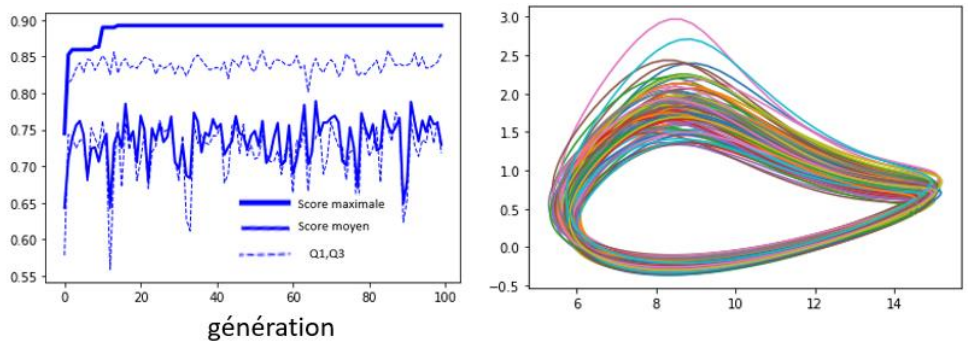



*Test 2 (sur la figure de droite les valeurs sont en centimètre)*

*Longueurs nominales = [1.13, 3.79, 4.21, 4.14, 3.94, 5.52, 3.64, 3.89, 6.71, 4.91, 5.17, 6.31, 0.89]*

Sur ce deuxième test nous avons mis quasiment 4h à trouver cette solution, la convergence des scores est déjà plus évidente d'après le deuxième graphique. Quant à la robustesse de l'architecture il y un progrès. Cependant même si le cycle semble plus adapté à une marche régulière il y a encore de la dispersion. A ce stade pour nous le problème venait d'un minimum local en robustesse, pour éviter ce problème nous somme dit qu'il fallait éviter les croisements entre individus de la générations père puisque cela pouvait nous laissez bloquer dans un minimum local. Nous avons supprimé le croissement ce qui a donc augmenter le nombre de nouveau individus intégrés à chaque génération et pousse à une évolution plus rapide du code génétique des individus. Attention cela peut devenir être un problème puisque si le code génétique ne stabilise jamais il peut rater des solutions

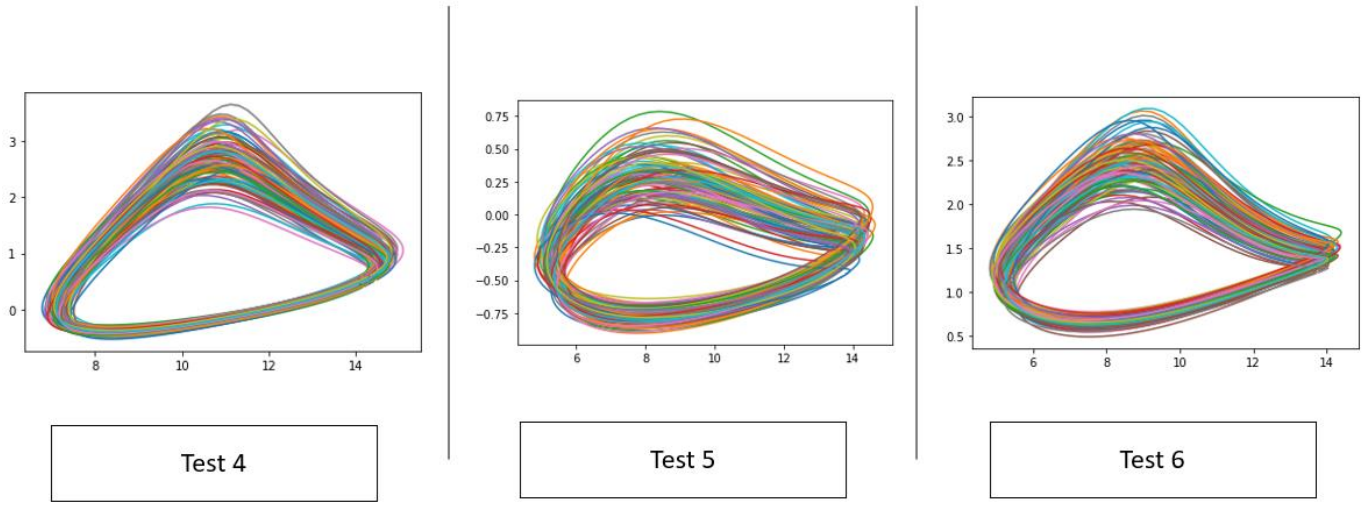
| Test 3 | |
|---|---|
| Ngen | 100 |
| Nind | 40 |
| Quantite_selectionne (%) | 20 |
| Quantite_croisse (%) | 0 |
| Qauntite_mute (%) | 20 |
| Taux_mutation | 0,1 |
| N_MC | 15 |
| Sigma | 0,02 |
| temps exécution (h) | 2,86527778 |




*Test 3 (sur la figure de droite les valeurs sont en centimètre)*

*Longueurs nominales = [1.30, 3.60, 4.23, 3.91, 4.02, 5.52, 4.12, 3.89, 6.73, 4.74, 4.81, 6.07, 0.94]*

Retirer les croissements semble avoir été la solution de notre problème. La solution ici semble très robuste et la convergence est claire. On remarque que la solution a été trouvée dés le début de l'évolution des espèces. Avec ces paramètres il ne semble pas indispensable de faire autant de simulation et 25 générations aurait pu suffire. Il est difficile de conclure de manière certaines cependant car il s'agit ici d'un seul essai un coup de chance peu être possible. Nous avons donc relancé avec les mêmes hyperparamètres :




*Test 4 | Test 5 | Test 6*

Les 5 derniers résultats montrent que deux types d'architectures ressortent de l'algorithme de recherche. D'abord les Test 2 et 5 sont des solutions avec un score de robustesse plus faible que les trois autres solutions et avec des « enjambés » qui ne monte jamais plus haut qu'un centimètre. De l'autre coté les trois autres résultats beaucoup plus robustes mais avec des « enjambés » qui monte quartes fois plus haut.

## Tangages et Roulis

Il faut maintenant comparer les deux types d'architectures robuste par rapport tangages et au roulis pour cela j'utilise le Score2.

Architecture Type I (petites enjambés) :

[1.13, 3.79, 4.21, **4.14**, 3.94, 5.52, 3.64, 3.89, 6.71, **4.91**, **5.17**, 6.31, 0.89]

[1.24**,** 3.80, 3.98, **4.17**, 3.82, 5.42, 4.21, 3.94, 6.54, **5.08**, **5.13**, 6.25, 0.77]

Score2 : 0.144 et 0.154

Architecture Type II (grandes enjambés) :

[1.30, 3.60, 4.23, **3.91**, 4.02, 5.52, 4.12, 3.89, 6.73, **4.74**, **4.81**, 6.07, 0.94]

[1.19, 3.80, 4.04, **3.77**, 4.22, 5.83, 4.19, 3.79, 6.32, **4.71, 4.99**, 6.24, 1.11]

[1.43, 3.99, 3.97, **3.56**, 3.73, 5.29, 4.36, 3.90, 6.19, **4.71**, **4.83**, 5.86, 0.78]

Score2 : 0.144, 0.156 et 0.139

Les deux architectures on des Score 2 similaires. **Comme les architectures de type II sont les plus robuste ce sera ce type de mécanisme qui sera optimal pour avoir une architecture robuste qui ne tangue pas et ne roule pas. Au final le tangage maximal est d'environ 5° et le roulis maximal de 0.5° pour les deux architectures.**

## Les limites du modèle de recherche d'optimal

La première limite c'est le temps de calcul je pense que les résultats aurait était beaucoup plus facile d'interprétation si notre programme de recherche prenait autant de temps. Il faut savoir qu'au bout d'un certain temps de calcul l'exécuteur Python risque de s'arrêté, en tous cas le problème est déjà arrivé plusieurs fois de notre côté.

Une autre limite vient du faite que nous visons deux objectifs en même temps tangage et roulis et robustesse. Nous avons donc choisi de d'abord réglé le problème de robustesse pour ensuite régler le problème des angles. Rien ne garantit qu'il ne fallait pas commencer par résoudre le problème d'angle en premier pour ensuite se concentrer sur la robustesse du système pour répondre au problème le plus efficacement possible.

Nous avons choisi de ne pas prendre en compte la vitesse d'avance du mécanisme dans notre recherche. Cependant pour réaliser jusqu'au bout le projet il va bien falloir s'assurer que le mécanisme avance à une vitesse décente.

Pour augmenter la vitesse de calcul il a fallu réduire le nombre d'échantillon dans l'algorithme de Monte-Carlo. Et au final nous avons fait nos tests avec seulement 15 échantillons par architectures. Selon nous cette valeur est vraiment juste 30 ou 60 échantillon par architecture nous aurait assurer de la qualité du score avec seulement 15 échantillon une architecture peut se retrouver mal jugé et donc mal classé (attention dans notre cas 15 reste correcte, mais limite).