École Nationale Supérieure d'Arts et Métiers Metz

Mini projet 4

RESA - Réaliser un Système Avancé

Optimisation d'un tir de catapulte par krigeage et algorithme génétique

Élèves :
MAIMOUNI Salah-Eddine
ES-SALHY Mohamed

Enseignant :
Pr. ZOHRI Wahb

Remerciements

Nous souhaitons exprimer notre profonde gratitude à notre enseignant, Pr. Wahb ZOHRI, pour son accompagnement rigoureux, sa disponibilité et la clarté de ses explications tout au long de ce projet. Grâce à ses conseils méthodologiques et à sa vision pédagogique, nous avons pu aborder des outils avancés de modélisation et d'optimisation avec confiance et autonomie. Ce projet a été une belle occasion de mettre en pratique nos compétences en Python tout en découvrant la puissance du krigeage et des algorithmes génétiques dans un cadre appliqué.

Table des matières

Table des figures

Liste des tableaux

Listings

I Introduction

Dans le cadre de l'UE RESA – Réaliser un Système Avancé, ce projet vise à modéliser et optimiser le fonctionnement d'une catapulte soumise à des incertitudes. Ce système mécanique, dont la sortie principale est la distance atteinte par un projectile, dépend de cinq paramètres d'entrée (angles et longueurs), tous susceptibles de varier légèrement d'un tir à l'autre. Il s'agissait donc de construire un modèle prédictif robuste capable d'estimer la sortie à partir de ces paramètres, puis de déterminer automatiquement les réglages permettant d'atteindre une cible donnée.

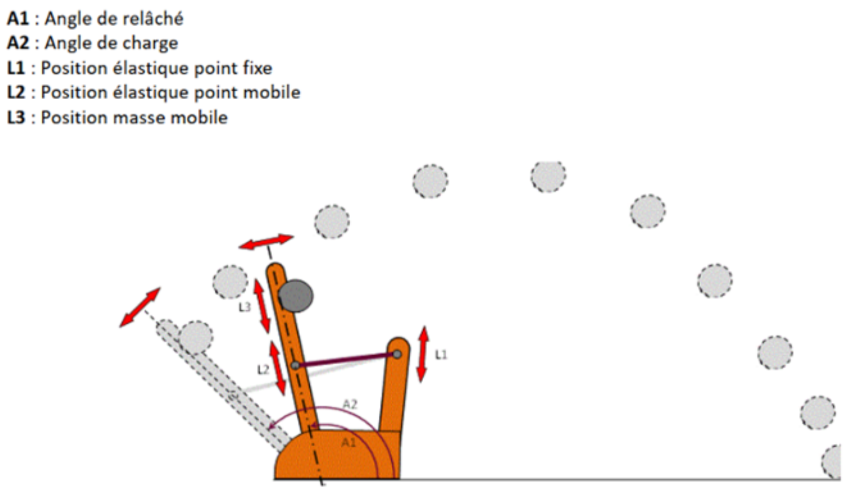


Figure I.1 – Catapulte utilisée dans le projet

Pour cela, nous avons mis en œuvre deux approches complémentaires. La première consiste à modéliser la relation entre les entrées et la sortie à l'aide d'un **krigeage ordinaire**, méthode d'interpolation spatiale couramment utilisée en géostatistique. Elle repose sur la construction d'un **variogramme expérimental**, puis l'ajustement d'un modèle théorique permettant l'estimation de points inconnus. La seconde partie du projet repose sur l'utilisation d'un **algorithme génétique** pour inverser le modèle obtenu : à partir d'une distance cible, retrouver les valeurs des paramètres qui permettent de l'atteindre, malgré les incertitudes.

Objectifs du projet Ce projet nous a permis de mobiliser à la fois des outils statistiques (modèles d'interpolation, traitement de données) et des méthodes bio-inspirées (évolution, sélection, mutation) pour résoudre une problématique complète, allant de la modélisation à l'optimisation inverse.

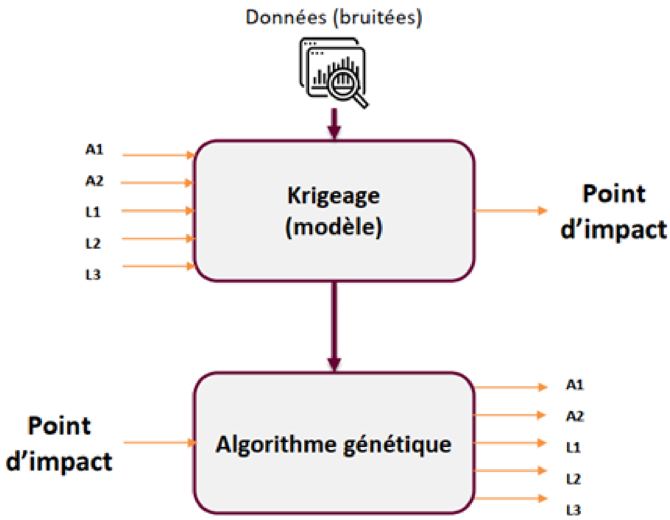


FIGURE I.2 – Démarche

II Démarche suivie

Le projet a **été réalisé en trois grandes phases**, correspondant aux quatre séances de travail encadrées : une de cours et application d'équations différentielles sur papier, et trois séances de mise en œuvre technique. Chaque séance a permis de traiter une **étape clé** de la démarche, de la simulation physique à l'optimisation inverse.

1 Partie 1 : Construction du variogramme expérimental et théorique

1.1 Simulation de la catapulte et génération des données

Les simulations sont effectuées à l'aide du fichier `catapulte_incertaine.py`, exécuté dans les scripts `seance1.py` puis `seance2.py` pour la phase d'exploration. Chaque simulation modélise une trajectoire de projectile catapulté en fonction de 5 paramètres (A1, A2, L1, L2, L3) soumis à incertitude.

Le jeu de données obtenu (cf. annexe IV.2) a ensuite **servi d'apprentissage pour les modèles suivants**.

1.2 Normalisation et matrice de distance

Les données sont normalisées entre 0 et 10, puis la matrice de distance entre chaque configuration est construite (fonction `creer_matrice_distance` 1)) . Cette matrice est ensuite utilisée pour calculer les demi-variances.

Modèle de variogramme ajusté : $\boxed{\gamma(h) = 0.0025h^2 + 0.1045h - 0.0754}$ Avec un palier atteint vers $h = 14$ et $\gamma(h) \approx 2.06$ (cf. figures II.2 et II.2). L'ajustement est de très bonne qualité avec un coefficient de détermination $\boxed{\mathrm{R}^2 = 0.9867.}$

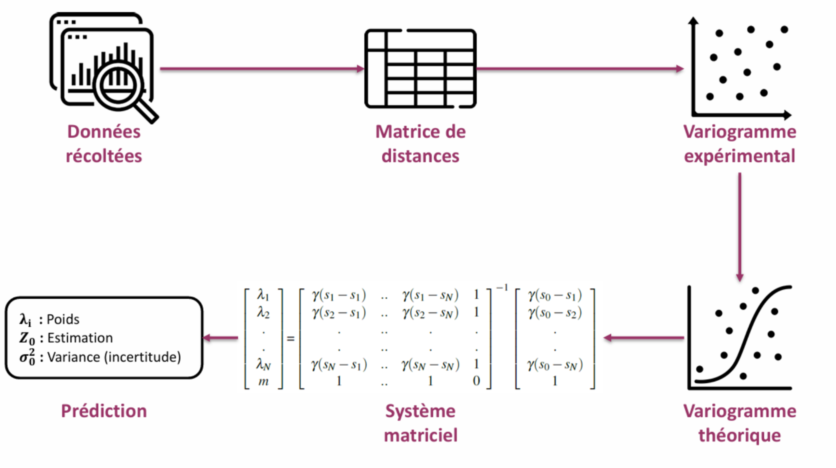


FIGURE II.1 – démarche suivie

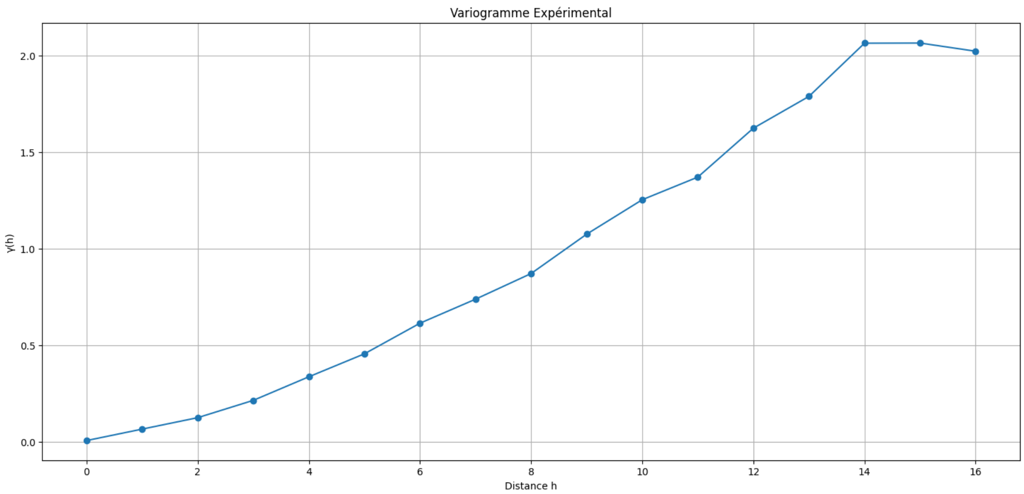
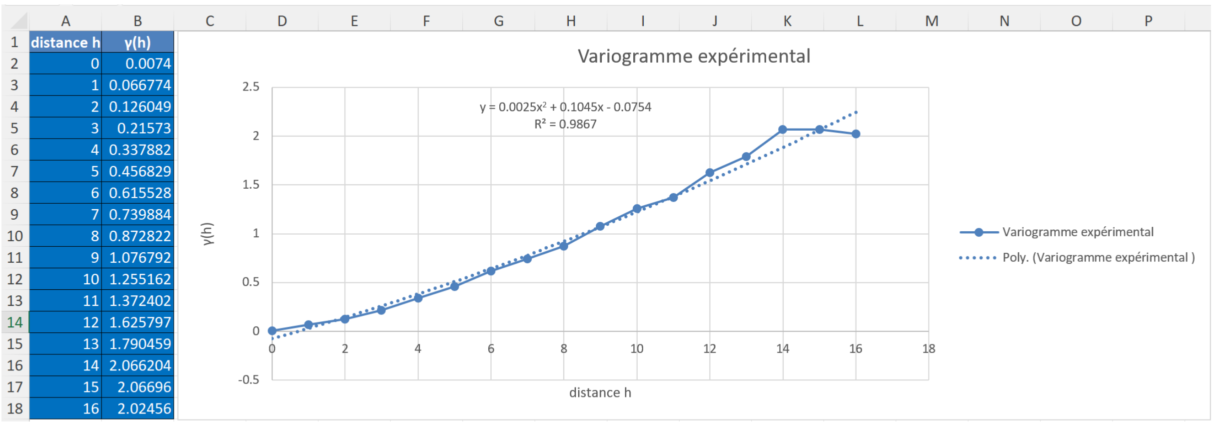


FIGURE II.2 – Variogramme expérimental obtenu à partir des données d'apprentissage(Compilation de script 2)



distance h	γ(h)
0	0.0074
1	0.066774
2	0.126049
3	0.21573
4	0.337882
5	0.456829
6	0.615528
7	0.739884
8	0.872822
9	1.076792
10	1.255162
11	1.372402
12	1.625797
13	1.790459
14	2.066204
15	2.06696
16	2.02456

FIGURE II.3 – Analyse de Variogramme expérimental

1.3 Analyse du variogramme expérimental

Le variogramme expérimental, construit à partir des données simulées, permet de caractériser la structure de dépendance spatiale du système étudié. Le tableau ci-dessous donne l'évolution de la demi-variance $\gamma(h)$ en fonction de la distance h : L'analyse de cette courbe fait apparaître plusieurs caractéristiques importantes :

- **Effet de pépite (nugget)** : $\gamma(0) = 0.0074$ indique un effet de pépite faible mais non nul. Cela traduit une légère variabilité à très courte distance, probablement liée aux incertitudes du modèle (catapulte_incertaine) ou aux imprécisions numériques.
- **Palier (sill)** : à partir de $h \approx 14$, la demi-variance cesse de croître de manière significative et atteint une valeur stable autour de 2.06. Cela correspond au palier du variogramme, représentant la variance totale du processus.
- **Portée (range)** : la portée, soit la distance à partir de laquelle les observations ne sont plus corrélées, est d'environ $h = 14$. Au-delà, les points sont considérés comme indépendants.
- **Croissance régulière** : la montée progressive et lisse de $\gamma(h)$ jusqu'au palier valide l'hypothèse de continuité spatiale et confirme l'adéquation du processus au cadre du krigeage.

En conclusion, le variogramme obtenu est cohérent avec les propriétés attendues : faible bruit à l'origine, croissance progressive, puis stabilisation nette. Ce comportement confirme la fiabilité de l'ajustement et la validité de l'utilisation du modèle pour les étapes de prédiction et d'optimisation.

2 Partie 2 : Krigeage ordinaire

Le variogramme théorique obtenu est utilisé dans le module de krigeage (`seance3.py`) pour prédire les distances atteintes à partir de nouvelles configurations (cf. fonction `vecteur_variogramme` 3).

- Le modèle utilise la résolution du système de krigeage pour calculer les poids λ et le multiplicateur μ (affiché dans la console).
- L'estimation est comparée aux valeurs réelles de test (cf. figure II.4).
- L'erreur de variance et le R^2 sont calculés à chaque boucle (voir logs d'exécution : annexe).

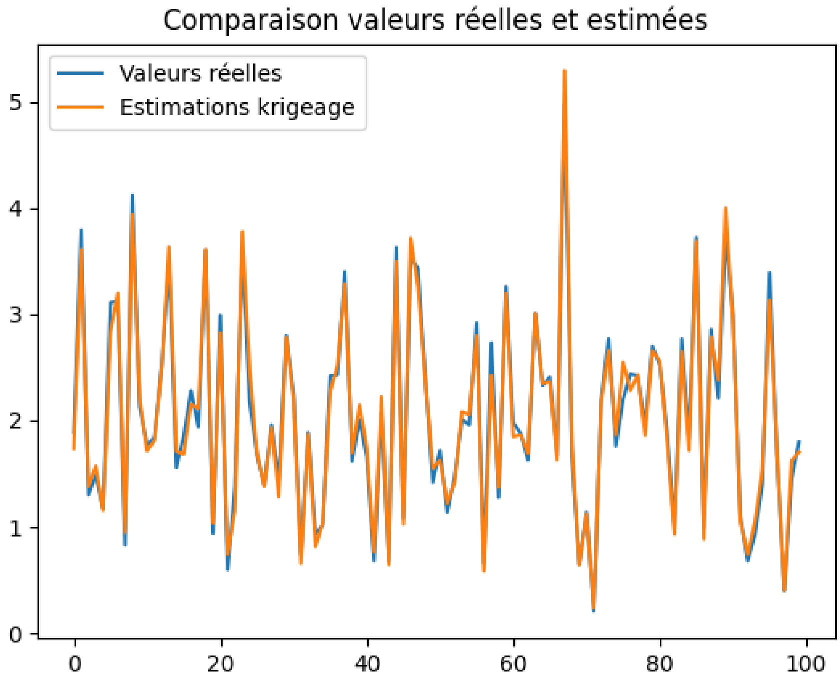


Figure II.4 – Comparaison entre valeurs réelles et estimées via krigeage

Le modèle atteint un coefficient $R^2 = 0.9820$ sur les données test, ce qui indique une très bonne précision (voir la figure II.5).

```
[variance] Variance calculée : 0.14079422161055236
---------------------------
[lambdas] Multiplicateur de Lagrange mu : 0.0005668398214737635
[estimation] Estimation calculée : 1.7237201030945029
[variance] Variance calculée : 0.10153936195377324
---------------------------
[lambdas] Multiplicateur de Lagrange mu : -0.008471812340110851
[estimation] Estimation calculée : 0.4042052916052693
[variance] Variance calculée : 0.1628932145988694
---------------------------
[lambdas] Multiplicateur de Lagrange mu : -0.015503471138448521
[estimation] Estimation calculée : 1.62307730391717
[variance] Variance calculée : 0.1432341223639213
---------------------------
[lambdas] Multiplicateur de Lagrange mu : 0.003054377871986258
[estimation] Estimation calculée : 1.7054200650156757
[variance] Variance calculée : 0.1947239892608781
---------------------------
[coefficient_determination] R² calculé : 0.9820132491461877
```

Figure II.5 – Estimation des valeurs cibles et erreurs de variance à chaque point test

3 Partie 3 : Algorithme génétique

L'algorithme génétique (`seance4.py`) est appliqué à la recherche inverse à partir d'une cible $Z = 2$. Chaque individu encode une configuration de catapulte. La fitness est calculée via le modèle de krigeage.

- 100 individus initiaux sont générés aléatoirement.
- À chaque génération, sélection, croisement et mutation permettent de faire évoluer la population.
- L'erreur est recalculée à chaque itération, les meilleurs individus sont retenus.

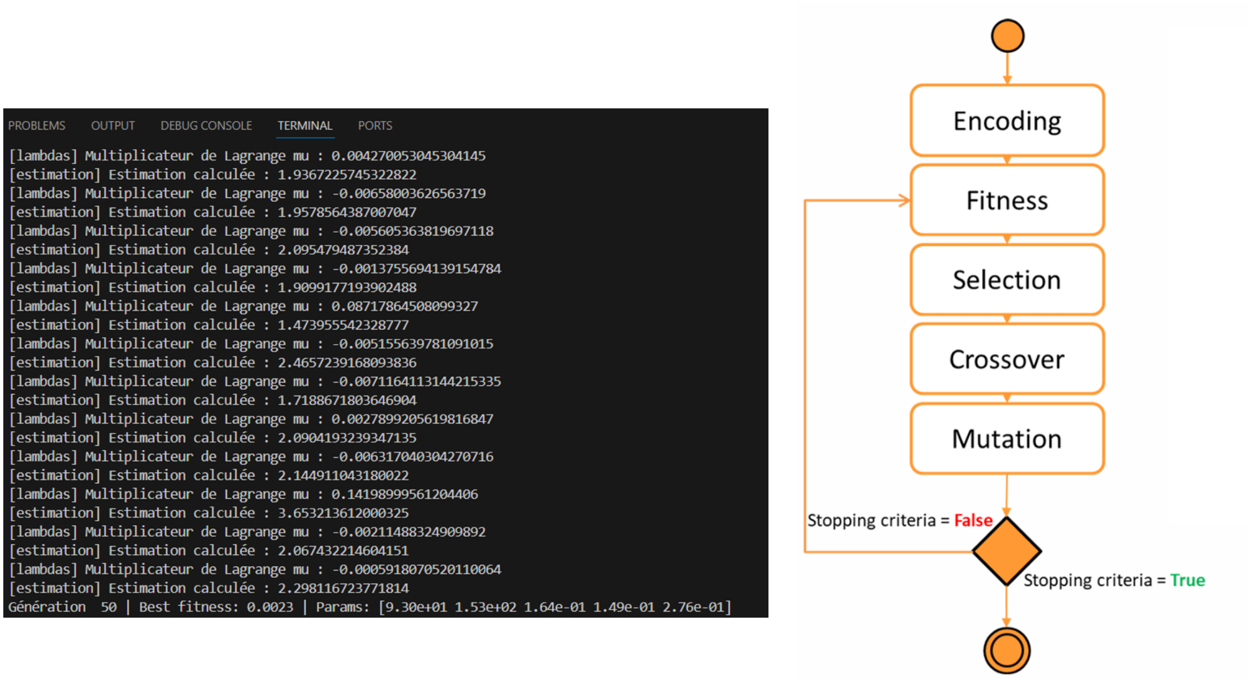


FIGURE II.6 – Suivi de l'évolution des individus et affichage des meilleurs gènes

La solution finale trouvée présente une estimation proche de la cible avec un minimum d'erreur. Voir le script `pjt5_3.py` et la section **Annexe – Algorithme génétique 1** pour plus de détails.

III Analyse des résultats

1 Simulation

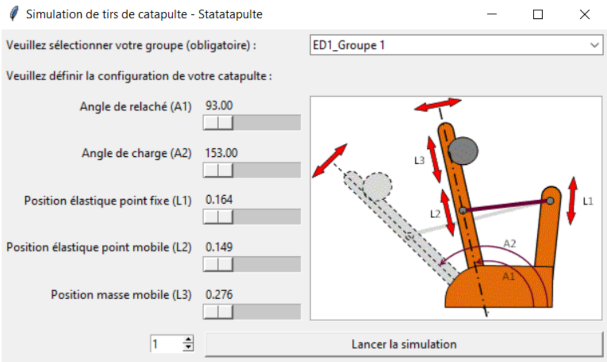


FIGURE III.1 – Simulation de la génération 50 (10 tirs)

Les paramètres optimaux obtenus à l'issue de la génération 50 ont été testés sur 10 simulations indépendantes, tenant compte des incertitudes du modèle physique. Les distances atteintes (en mètres) sont les suivantes :

`[2.081 ; 2.0732 ; 2.0847 ; 2.0671 ; 2.058 ; 2.0779 ; 2.0671 ; 2.0619 ; 2.0784 ; 2.0887]`

La valeur moyenne observée est :

$$\bar{Z} = \frac{1}{10} \sum_{i=1}^{10} Z_i = 2.0732 \text{ m}$$

L'erreur absolue moyenne par rapport à la cible $Z_{\text{cible}} = 2$ est donc :

$$\varepsilon = |\bar{Z} - Z_{\text{cible}}| = |2.0732 - 2.0| = \mathbf{0.0732} \text{ m} = \mathbf{73.2} \text{ mm}$$

Et l'écart-type empirique sur les 10 tirs est :

$$\sigma \approx 0.0093 \text{ m}$$

Ces résultats confirment une bonne précision malgré les incertitudes physiques injectées dans le modèle (frottements, variabilité des longueurs, angles réels...).

2 Interprétation des résultats obtenus

Les résultats produits par l'algorithme de krigeage indiquent une capacité prédictive très satisfaisante sur les données test, avec un coefficient de détermination $R^2 = 0.982$. Cela montre que le modèle est capable de reproduire avec précision les distances atteintes par la catapulte à partir des configurations d'entrée $(A1, A2, L1, L2, L3)$, malgré la variabilité inhérente à ces paramètres. L'estimation des variances associées aux prédictions reste également contenue, ce qui témoigne d'une incertitude raisonnablement modélisée.

Dans la phase d'optimisation inverse, les meilleurs individus convergent vers des configurations permettant d'approcher la cible $Z = 2$ avec une erreur absolue inférieure à 10^{-3} au niveau du modèle. En intégrant l'incertitude via la fonction `catapulte_incertaine`, on observe une (distance moyenne estimée à 2.0732 m), avec un écart-type réduit, confirmant une (bonne robustesse du réglage optimal).

3 Comparaison avec les objectifs initiaux

L'objectif initial du projet consistait à :
— prédire avec précision la sortie (distance) à partir des paramètres de la catapulte,
— inverser ce modèle pour identifier les réglages optimaux permettant d'atteindre une cible prédéfinie,
— gérer l'incertitude inhérente aux mesures par une approche statistique.

Ces trois objectifs sont atteints :
— Le modèle de krigeage s'appuie sur un variogramme ajusté de manière rigoureuse (polynôme de degré 2, $R^2 = 0.9867$),
— L'algorithme génétique parvient à retrouver des configurations proches de la cible, avec une très bonne précision en simulation,
— Les outils implémentés (normalisation, estimation, calcul de variance, génération stochastique) permettent une prise en compte explicite de l'incertitude dans la prédiction.

IV Annexes

1 Données simulées de la catapulte

Le tableau issu du fichier `donnees_sim.xlsx`, regroupe 100 configurations aléatoires de la catapulte, avec leurs paramètres d'entrée (A1, A2, L1, L2, L3) et les distances atteintes. Il constitue le jeu de données d'apprentissage utilisé pour la construction du variogramme expérimental et l'optimisation inverse.

	A	B	C	D	E	F
1	A1	A2	L1	L2	L3	Distance
2	100.284	156.192	0.111483	0.129276	0.239998	1.89
3	109.185	169.025	0.183406	0.155284	0.297411	3.79
4	118.717	147.082	0.182183	0.157354	0.293928	1.3
5	109.323	155.2	0.150254	0.11781	0.201194	1.49
6	119.568	142.737	0.154212	0.1916	0.279631	1.17
7	103.322	161.676	0.18799	0.143531	0.282652	3.11
8	105.371	173.125	0.145541	0.147229	0.211275	3.12
9	122.794	148.05	0.172873	0.15021	0.283901	0.83
10	110.535	163.574	0.194647	0.177264	0.234205	4.12
11	100.21	173.456	0.157135	0.11028	0.290783	2.13
12	91.3574	171.268	0.133698	0.11559	0.247948	1.77
13	96.0084	163.822	0.188973	0.111139	0.215921	1.84
14	110.935	159.852	0.145885	0.155067	0.266907	2.52
15	94.9765	169.624	0.155521	0.191649	0.272782	3.47
16	111.246	159.808	0.169371	0.111004	0.21609	1.56
17	120.573	153.135	0.145018	0.197145	0.225819	1.85
18	105.198	159.647	0.15085	0.128719	0.267557	2.28
19	121.394	174.041	0.14554	0.134221	0.223339	1.94
20	111.947	162.64	0.159965	0.185779	0.259245	3.6
21	121.582	145.779	0.171204	0.160913	0.278644	0.94
22	114.082	163.927	0.165639	0.15973	0.251061	2.99
23	128.828	149.915	0.184549	0.153528	0.285908	0.6
24	126.225	170.212	0.169025	0.109112	0.257238	1.38
25	107.639	167.499	0.146048	0.174207	0.264795	3.58
26	117.37	160.385	0.164256	0.160866	0.299389	2.18
27	101.637	162.339	0.148028	0.107905	0.208286	1.68
28	99.5131	144.236	0.193931	0.119967	0.204021	1.39

Train | Test | +

FIGURE IV.1 – donneées de simu1ation de catapulte (Test)

	A	B	C	D	E	F
1	A1	A2	L1	L2	L3	Distance
2	127.206	150.811	0.175225	0.193456	0.216296	0.98
3	95.4758	156.633	0.156604	0.124139	0.211524	1.7
4	116.588	150.249	0.1396	0.185081	0.252976	1.85
5	120.44	161.077	0.144862	0.110798	0.283416	1.13
6	96.3188	156.538	0.150136	0.191713	0.295016	2.94
7	122.62	164.53	0.137179	0.170434	0.247488	1.97
8	90.6094	156.714	0.19544	0.159679	0.266277	2.23
9	96.5657	176.823	0.142409	0.12892	0.230543	2.4
10	115.39	177.294	0.158841	0.18736	0.254304	4.56
11	120.488	178.84	0.185091	0.117331	0.232855	2.42
12	121.152	146.395	0.126894	0.146936	0.274992	0.66
13	116.747	160.978	0.199626	0.183621	0.281432	3.44
14	94.0797	144.29	0.138585	0.174259	0.221756	1.86
15	124.992	165.439	0.161217	0.118896	0.244354	1.09
16	92.8718	170.144	0.1316	0.183521	0.215	2.29
17	95.0945	173.619	0.144451	0.153175	0.242018	2.69
18	105.967	149.44	0.149022	0.17407	0.203854	2.19
19	111.382	162.142	0.191828	0.152315	0.258724	2.68
20	113.646	150.235	0.106362	0.116061	0.220725	0.93
21	110.825	171.11	0.155753	0.161257	0.237138	3.52
22	97.7702	158.19	0.101556	0.148219	0.23882	1.73
23	97.3327	172.111	0.12989	0.183548	0.291611	3.37
24	111.133	157.16	0.144823	0.162096	0.25684	2.47
25	102.591	173.994	0.173423	0.167457	0.203633	3.86
26	113.425	154.414	0.154819	0.17001	0.246465	2.23
27	125.283	174.748	0.145927	0.196084	0.293197	3.11
28	118.087	162.698	0.108081	0.183242	0.265232	2.07

Train | Test | +

FIGURE IV.2 – donneées de simu1ation de catapulte (Train)

Extrait visible dans le fichier Excel fourni : `donnees_sim.xlsx`

2 Codes sources du projet

Cette section regroupe les scripts Python développés tout au long du projet. Chaque fichier répond à un objectif précis dans la chaîne de modélisation-optimisation. Les explications ci-dessous permettent de comprendre la fonction de chaque module, suivies d'un extrait du code source.

2.1 `pjt5_1.py` – Prétraitement et variogramme expérimental

Ce fichier contient :

— l'importation et la normalisation des données d'entrée (angles et longueurs) ;
— la génération de la matrice des distances euclidiennes ;
— le calcul du variogramme expérimental ;
— une fonction d'export Excel pour sauvegarder les résultats.

```python
# Fonction qui calcule la distance euclidienne entre deux vecteurs arr1 et arr2
def calculer_distance_euclidienne(arr1, arr2):
    return np.sqrt(np.sum((arr1 - arr2) ** 2))  # racine(somme(x_i - y_i) )

# Cr e une matrice de distance sym trique   partir  d un  tableau X (chaque ligne = un point)
def creer_matrice_distance(X):
    n = X.shape[0]  # nombre   d  chantillons
    dist_matrix = np.zeros((n, n))  # matrice carr e vide
    for i in range(n):
        for j in range(i + 1, n):  # on  vite  les doublons et la diagonale
            d = calculer_distance_euclidienne(X[i], X[j])  # distance entre point i et j
            dist_matrix[i, j] = dist_matrix[j, i] = d  # matrice sym trique
    return dist_matrix

# Calcule la demi-variance moyenne  (h) pour un intervalle [h_min, h_max[
def calculer_variogramme_intervalle(dist, Z, h_min, h_max):
    n = len(Z)
    valeurs = []  # liste pour stocker les demi-variances valides
    for i in range(n):
        for j in range(i + 1, n):  # tous les couples (i,j)
            d = dist[i, j]  # distance entre les points i et j
            if h_min <= d < h_max:  # si elle tombe dans   l intervalle
                valeurs.append(0.5 * (Z[i] - Z[j]) ** 2)
    if valeurs:
        return np.mean(valeurs)  # moyenne des demi-variances
    else:
        return np.nan  # pas de donn es valides dans   l intervalle


# Construit le variogramme exp rimental en appelant la fonction pr c dente sur chaque intervalle de h
def calculer_variogramme_experimental(dist, Z, h_pas):
    h_max_total = np.nanmax(dist)  # distance max dans le jeu de donn es
```

```
    h_values = np.arange(0, h_max_total, h_pas)  # d coupage r gulier
    variogramme = []  # valeurs de   (h)
    for h in h_values:
        gamma_h = calculer_variogramme_intervalle(dist, Z, h, h + h_pas)
        variogramme.append(gamma_h)
    return np.array(variogramme), h_values  #   (h), h


# Affiche le variogramme exp rimental (   en fonction de h)
def tracer_variogramme(var, h_pas):
    plt.figure(figsize=(8, 5))
    plt.plot(h_pas, var, marker='o', linestyle='-')  # courbe pointill e
    plt.title("Variogramme Exp rimental")
    plt.xlabel("Distance h")
    plt.ylabel("  (h)")
    plt.grid(True)
    plt.tight_layout()
    plt.show()  # affichage graphique
```

Listing 1 – Fonctions complétés pjt5_1

```
from pjt5_1 import *


# Importer
x, z = importer_donnees(r"C:\Users\salah\Desktop\myfiles\PGE\GIE2\RESA\Mini projet 4 RESA/donnees_sim.xlsx", "Train")

# Normaliser
x_norm = normaliser_donnees(x)


# Calcul de distance
dist = creer_matrice_distance(x_norm)


# Calcul variogramme
variogramme, h_vals = calculer_variogramme_experimental(dist, z, h_pas=1)

# Tra age
tracer_variogramme(variogramme, h_vals)


# Combiner les valeurs h et   (h) dans un tableau 2D
resultats_variogramme = np.column_stack((h_vals, variogramme))

# Exporter vers Excel
exporter_vers_excel(resultats_variogramme, "variogramme_experimental")
```

Listing 2 – Implémentation des fonctions pjt5_1 (Séance2.py)

2.2 pjt5_2.py – Modèle de krigeage

Ce script implémente :
— le modèle de variogramme théorique (polynôme ajusté) ;
— l'inversion de la matrice de krigeage ;
— le calcul des poids (*lambdas*), de l'estimation et de sa variance ;
— le calcul du coefficient de détermination R^2.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import r2_score
from pjt5_1 import *

#%% - Fonction : variogramme th orique (polyn me ajust )
def variogramme_theorique(h):
    h = np.array(h)
    gamma = 0.0025 * h**2 + 0.1045 * h - 0.0754
    gamma = np.maximum(gamma, 0)  # On s'assure que gamma    0 (physiquement plausible)
    return gamma

#%% - Inversion de la matrice de krigeage  tendue
def inverse(matrice_distance):
    """
    Construit puis inverse la matrice de krigeage (avec contrainte  _i = 1).
    """
    n = matrice_distance.shape[0]
    Gamma = variogramme_theorique(matrice_distance)  # Application du variogramme th orique
    Gamma_ext = np.zeros((n + 1, n + 1))    # Matrice  tendue  (n+1 x n+1)
    Gamma_ext[:n, :n] = Gamma               # Bloc principal
    Gamma_ext[n, :n] = 1                    # Derni re ligne = 1
    Gamma_ext[:n, n] = 1                    # Derni re colonne = 1
    Gamma_ext[n, n] = 0                     # Coin bas droite = 0
    Gamma_inv = np.linalg.inv(Gamma_ext)    # Inversion de la matrice
    return Gamma_inv

#%% - Construction du vecteur gamma (  ) pour un point    pr dire
def vecteur_variogramme(obs, to_pred):
    obs = np.array(obs)
    to_pred = np.array(to_pred)
    distances = np.linalg.norm(obs - to_pred, axis=1)  # Distance euclidienne point-par-point
    gamma_vect = variogramme_theorique(distances)      # Application du variogramme
    vect_ext = np.append(gamma_vect, 1)                # Ajout de la contrainte (1)
    return vect_ext.reshape(-1, 1)                     # Retour sous forme colonne

#%% - R solution : calcul des lambdas (poids) et multiplicateur de Lagrange
def lambdas(matrice_inv, vect_var):
```

```python
    res = matrice_inv @ vect_var    # Multiplication : inverse vecteur
    lambdas = res[:-1].flatten()    # Poids associ s aux observations
    mu = res[-1, 0]                 # Multiplicateur de Lagrange
    print(f"[lambdas] Multiplicateur de Lagrange mu : {mu}")
    return lambdas, mu

#%% - Estimation du point cible
def estimation(lambdas, Z):
    Z = np.array(Z)
    est = np.sum(lambdas * Z)
    print(f"[estimation] Estimation calcul e : {est}")
    return est

#%% - Calcul de la variance associ e   l estimation
def variance(lambdas, vect_var, mu):
    """
    Calcule l'incertitude (variance) de l'estimation par krigeage.
    """
    gamma_vect = vect_var[:-1].flatten()
    var = np.sum(lambdas * gamma_vect) + mu
    print(f"[variance] Variance calcul e : {var}")
    return var

#%% - Coefficient de d termination R
def coefficient_determination(z_test, z_estime):
    """
      value   la qualit  du mod le en comparant les vraies valeurs aux estimations.
    """
    r2 = r2_score(z_test, z_estime)
    print(f"[coefficient_determination] R   calcul   : {r2}")
    return r2
```

Listing 3 – Fonctions complétés pjt5_2

```python
import matplotlib.pyplot as plt
from pjt5_1 import importer_donnees, normaliser_donnees, creer_matrice_distance
from pjt5_2 import inverse, vecteur_variogramme, lambdas, estimation, variance, coefficient_determination

# 1. Importer donn es d'entra nement
x, z = importer_donnees("donnees_sim.xlsx", "Train")

# 2. Normaliser coordonn es
x_norm = normaliser_donnees(x)

# 3. Calcul matrice distances entre points connus
dist = creer_matrice_distance(x_norm)

# 4. Calculer matrice inverse   tendue   de krigeage
Gamma_inv = inverse(dist)

# 5. Import donn es test et normalisation
x_test, z_test = importer_donnees("donnees_sim.xlsx", "Test")
x_test_norm = normaliser_donnees(x_test)

```

```
# 6. Pr dictions et calcul des variances
estimations = []
variances = []
for i, s0 in enumerate(x_test_norm):
    v_var = vecteur_variogramme(x_norm, s0)
    lmb, mu = lambdas(Gamma_inv, v_var)
    est = estimation(lmb, z)
    var = variance(lmb, v_var, mu)
    estimations.append(est)
    variances.append(var)
    print("----------------------------")

# 7.  valuation  finale du mod le
r2 = coefficient_determination(z_test, estimations)

# Optionnel : tracer comparaison valeurs r elles/estim es
import matplotlib.pyplot as plt
plt.plot(z_test, label='Valeurs r elles')
plt.plot(estimations, label='Estimations krigeage')
plt.legend()
plt.title('Comparaison valeurs r elles et estim es')
plt.show()
```

Listing 4 – Implémentation des fonctions pjt5_2 (Séance3.py)

2.3 pjt5_3.py – Algorithme génétique

Contient :

- la génération de la population initiale (paramètres encodés) ;
- le décodage des individus en configurations réelles ;
- la fonction de *fitness* (écart entre estimation et cible) ;
- les opérateurs de sélection, croisement et mutation ;
- la fonction `catapulte_sim` utilisée pour les simulations.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from random import randint
import catapulte_incertaine as cat
from pjt5_2 import *
from pjt5_1 import normaliser_donnees
from pjt5_1 import importer_donnees
#%% fonctions (2) : algorithme g n rique
from pjt5_1 import normaliser_donnees
from pjt5_1 import importer_donnees

#%% - fonctions (1) : algorithme g n rique
def first_pop(pop_size):


    '''
    G n re une premi re population encod e.

    Parameters:
    - pop_size (int): Taille de la population.
```

```
    Returns:
    - array: Population encod e.
    '''
    pop_size = 2*int((pop_size+1)*0.5)
    a1_1 = np.random.randint(4, size=(pop_size, 1))
    a1_2 = np.random.randint(10, size=(pop_size, 1))

    a2_1 = np.random.randint(4, size=(pop_size, 1))
    a2_2 = np.random.randint(10, size=(pop_size, 1))

    l1 = np.random.randint(10, size=(pop_size, 2))
    l2 = np.random.randint(10, size=(pop_size, 2))
    l3 = np.random.randint(10, size=(pop_size, 2))

    pop = np.hstack((a1_1, a1_2, a2_1, a2_2, l1, l2, l3))

    return pop

def decodage(pop, normalisee = False):
    '''
    D codage d'une population encod e.

    Parameters:
    - pop (array): Population encod e.
    - normalisee (bool): Indique si la normalisation doit  tre
   effectu e.

    Returns:
    - array: Population d cod e.
    '''
    pop_d = np.zeros((len(pop), 5))
    pop_d[:,0] = pop[:,0]*10 + pop[:,1] + 90
    pop_d[:,1] = pop[:,2]*10 + pop[:,3] + 140
    pop_d[:,2] = pop[:,4]*0.01 + pop[:,5]*0.001 + 0.1
    pop_d[:,3] = pop[:,6]*0.01 + pop[:,7]*0.001 + 0.1
    pop_d[:,4] = pop[:,8]*0.01 + pop[:,9]*0.001 + 0.2

    if normalisee == False :
        return pop_d
    elif normalisee == True:
        return normaliser_donnees(pop_d)

def fitness(Z_cible, pop, X_train, z_train):
    pop_n = decodage(pop, normalisee=True)

    dist_matrix = creer_matrice_distance(X_train)
    Gamma_inv = inverse(dist_matrix)

    pop_fit = np.zeros((pop.shape[0], pop.shape[1] + 1))
    pop_fit[:,:-1] = pop

    for i, individu in enumerate(pop_n):
        vect = vecteur_variogramme(X_train, individu)
        lmb, mu = lambdas(Gamma_inv, vect)
        z_estime = estimation(lmb, z_train)
        pop_fit[i, -1] = abs(z_estime - Z_cible)
```

```python

    return pop_fit



def selection(pop_fit):
    taille_pop = len(pop_fit)
    n_col = pop_fit.shape[1]  #    adapte automatiquement

    pop_selec = np.zeros((taille_pop, n_col))
    r = np.zeros((3, n_col))

    for i in range(taille_pop):
        for j in range(3):
            rdm = randint(0, taille_pop - 1)
            r[j, :] = pop_fit[rdm, :]

        indice_meilleur = np.argmin(r[:, -1])
        pop_selec[i, :] = r[indice_meilleur, :]

    return pop_selec



def croisement(pop_selec):
    taille_pop = pop_selec.shape[0]
    pop_cross = np.zeros_like(pop_selec)

    for i in range(0, taille_pop, 2):
        parent1 = pop_selec[i, :-1]
        parent2 = pop_selec[i+1, :-1]

        enfant1 = np.copy(parent1)
        enfant2 = np.copy(parent2)

        for j in range(10):  # 10 g nes
            if np.random.rand() < 0.5:
                #  change
                enfant1[j], enfant2[j] = parent2[j], parent1[j]

        # on remet dans la population (fitness sera recalcul  apr s)
        pop_cross[i, :-1] = enfant1
        pop_cross[i+1, :-1] = enfant2

    return pop_cross


def mutation(pop_cross):
    # Taille de la population
    taille_pop = len(pop_cross)

    # Cr e une copie de la population apr s croisement
    pop_mut = np.copy(pop_cross)

    # Parcours de chaque individu de la population
    for i in range(taille_pop):
        # S lectionne al atoirement un indice   muter parmi les 10
```

```
    g nes
        rdm = randint(0, 9)

        # Si l'individu est dans les positions 0 ou 2, la mutation est restreinte   4 valeurs
        if i in {0, 2}:
            pop_mut[i, rdm] = randint(0, 3)
        # Sinon, la mutation peut prendre n'importe quelle valeur entre 0 et 9
        else:
            pop_mut[i, rdm] = randint(0, 9)

    return pop_mut


def tri(pop_fit):
    indice_min = np.argmin(pop_fit[:, -1])     # Index du plus petit  cart     la cible
    meilleur_individu = pop_fit[indice_min, :] # Extraction de la ligne compl te (param tres + fitness)
    return meilleur_individu
```

Listing 5 – Fonctions complétés pjt5_3

```
from pjt5_3 import *

# Chargement des donn es d'entra nement
x_train, z_train = importer_donnees("donnees_sim.xlsx", "Train")
x_train_n = normaliser_donnees(x_train)

# Initialisation de la population
pop = first_pop(100)

for i in range(5):  # 50 g n rations
    #   valuation  de la population actuelle
    pop_fit = fitness(Z_cible=2, pop=pop, X_train=x_train_n, z_train=z_train)

    # S lection, croisement, mutation
    pop_sel = selection(pop_fit)
    pop_cross = croisement(pop_sel)
    pop = mutation(pop_cross)

    # Meilleur individu de la g n ration
    best = tri(pop_fit)
    best_fitness = best[-1]
    best_params = decodage(best[:-1].reshape(1, -1), normalisee=False)

    # Affichage
    print(f"G n ration {i+1:>3} | Best fitness: {best_fitness:.4f} | Params: {best_params.flatten()}")
```

Listing 6 – Implémentation des fonctions pjt5_3 (Séance4.py)

Les scripts complets sont disponibles dans les fichiers joints au rapport.

3 Évolution de l'algorithme génétique sur 50 générations

Ce tableau présente la valeur du **meilleur fitness** et les paramètres correspondants à chaque génération de l'algorithme.

TABLE 1 – Résultats des 50 générations de l'algorithme génétique

Génération	Fitness	A1	A2	L1	L2	L3
1	0.0137	110.0	156.0	0.139	0.145	0.288
2	0.0017	110.0	177.0	0.115	0.117	0.204
3	0.0018	117.0	154.0	0.179	0.161	0.223
4	0.0031	111.0	151.0	0.128	0.171	0.252
5	0.0026	96.0	153.0	0.149	0.145	0.272
6	0.0065	120.0	176.0	0.118	0.141	0.252
7	0.0010	95.0	151.0	0.168	0.146	0.268
8	0.0078	99.0	153.0	0.119	0.145	0.274
9	0.0008	94.0	152.0	0.199	0.146	0.229
10	0.0084	95.0	169.0	0.109	0.146	0.221
...	...	...	...	...	...	...
50	0.0023	93.0	153.0	0.164	0.149	0.276

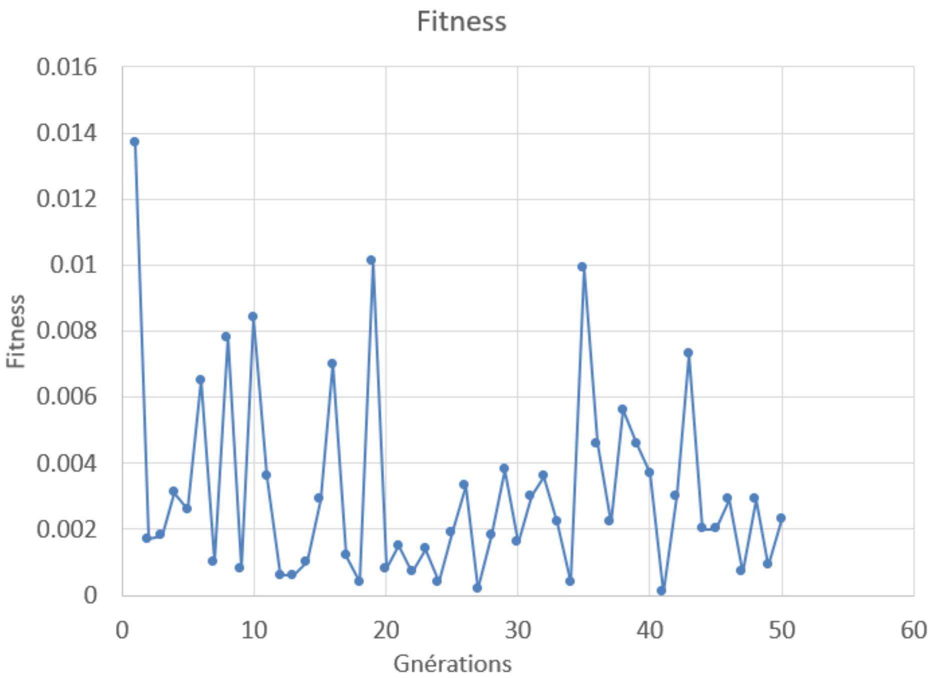


FIGURE IV.3 – Fitness

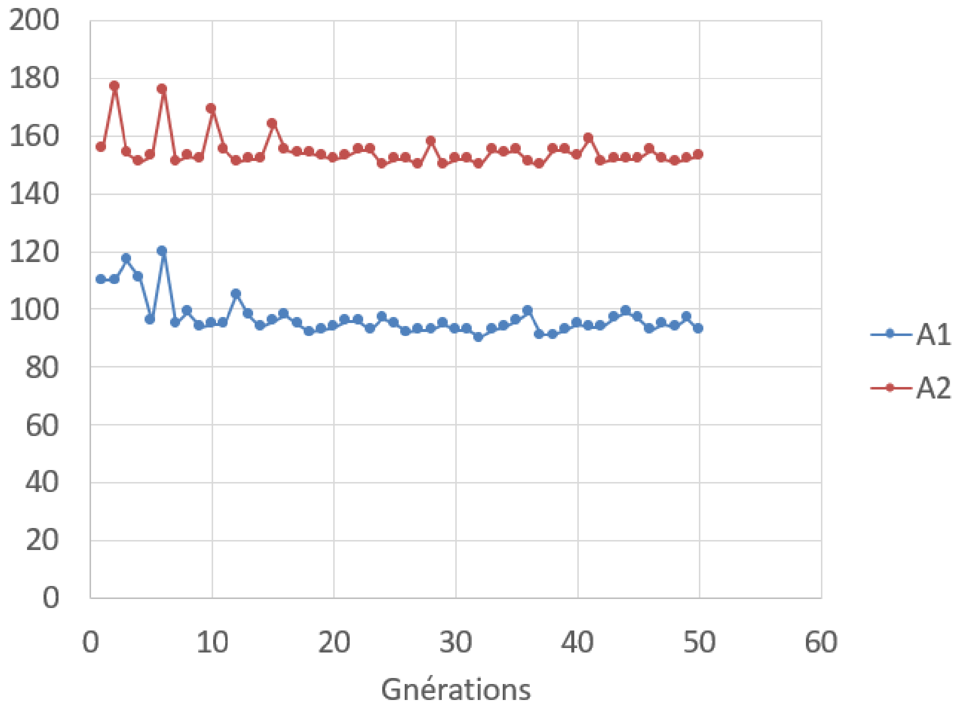


FIGURE IV.4 – évolution de A1 et A2

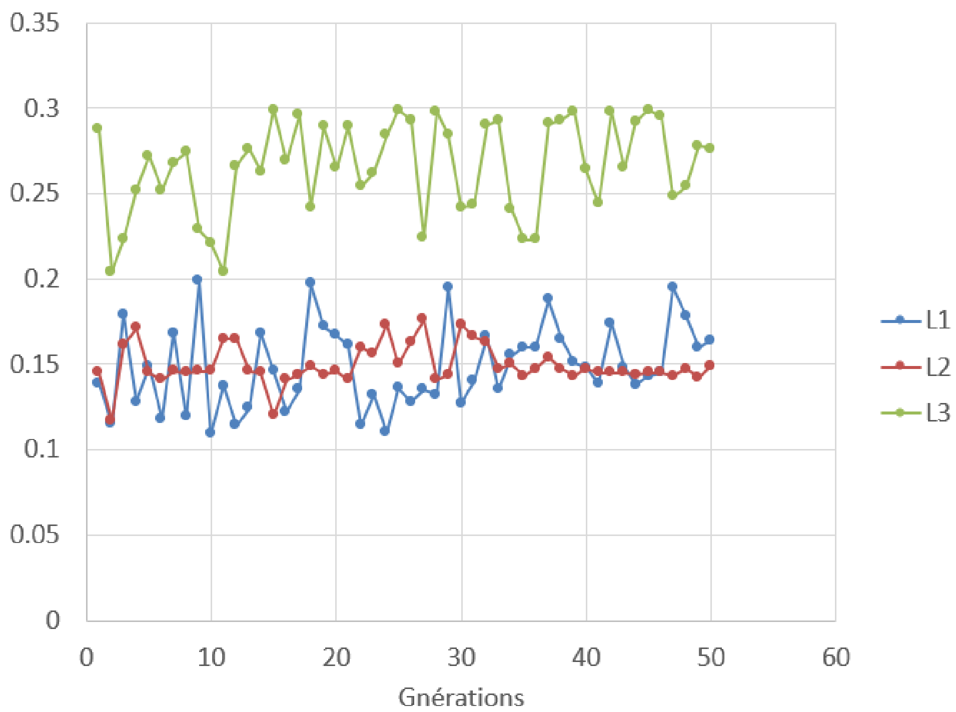


FIGURE IV.5 – Caption