Compte Rendu

Travail Pratique 2

Algorithme multi-objectifs : SPEA2 et NSGA-II

Métaheuristique en Optimisation

8INF852 – Groupe 1

Esmé James - JAME15539504
Wilfried Pouchous - POUW04069501

Sommaire

I - Organisation des fichiers

L'algorithme génétique se trouve dans le dossier Algorithme_TP2. À la racine se trouve :

- Un fichier « *main.m »* qui correspond au point de départ du programme
- Un dossier ***SPEA2*** contenant les étapes de l'algorithme SPEA2
- Un dossier ***NSGA-II*** contenant les étapes de l'algorithme NSGA-II
- Un dossier ***Problemes*** contenant les problèmes à traiter par l'algorithme
- Un dossier ***Operations*** qui contient les fonctions utilisées par les deux algorithmes (input utilisateurs, paramètres des problèmes, sélection, croisement, mutation, …)

II - Fonctionnalités implémentées

1. Fonctionnement principal

1.1. Main

Dans le main, un appel à la fonction *GetUserInput* permet d'afficher dans la console une liste de choix et permet de récupérer les entrées tapées par l'utilisateur. Un contrôle est ensuite effectué sur les choix pour s'assurer que le choix tapé correspond bien à une méthode implémentée.

S'il ne souhaite pas utiliser la fonction, l'utilisateur peut commenter l'appel à la fonction et rentrer les valeurs à la main dans le code dans la partie commentée entre les lignes 28 et 40. Attention cependant, ces variables existent uniquement pour les tests des développeurs et ne sont pas soumises à des vérifications.

Le main fait également appel à la fonction *SetProblemParameters* qui permet, en fonction du problème choisi par l'utilisateur, d'initialiser les bornes et le nombre de variables de décision.

Enfin, le main fait appel à *«SPEA2.m »* ou *« NSGA2.m »* en fonction du choix utilisateur.

1.2. Stockage de la population

La population est stockée dans une structure qui contient à chaque un type de données différentes. Dans les deux algorithmes, on peut retrouver les champs *"Val"*, *"ValObjective"*.

2. SPEA2

Pour l'algorithme SPEA2, nous nous sommes basés sur l'article à propos de SPEA2 et avons décidé de faire un algorithme réel uniquement avec un croisement de type Simulated Binary et une mutation de type Polynomiale.

2.1. Initialisation et début de boucle

Avant le début de la boucle de vie, l'algorithme SPEA2 fait appel à la fonction d'initialisation qui prend en paramètre les choix utilisateurs et les paramètres de l'algorithme et du problème. On génère ensuite une population aléatoire en remplissant la colonne *"Val"* avec n valeurs; n étant le nombre de variables de décision. On remplit également la colonne *"ValObjective"* en appelant le problème choisi.

On initialise également une archive vide et le nombre de génération G à 1. On rentre dans la boucle pour Gmax générations. Au début de chaque boucle, on concatène l'archive et la population précédente dans une nouvelle variable de population P.

2.2. Évaluation

La première étape de l'évaluation est d'effectuer un test de dominance sur les valeurs objectives de la population. On dit i domine j ($i < j$) si :

- Il existe un individu i dont la valeur objective $<$ la valeur objective de l'individu j
- Tout individu i dont la valeur objective $\leq$ la valeur objective de l'individu j

A chaque fois qu'un individu domine l'autre on incrémente sa force S, qui représente le nombre d'individu dominés.

La valeur de fitness finale valant $F = R + D$, il faut ensuite calculer R, la valeur de fitness brute, pour chaque individu. Ceci est fait en sommant la force S en fonction de la dominance.

Pour calculer D, il faut d'abord calculer la distance euclidiennes entre les individus puis les trier. On stocke cette données dans une valeur sigma et on récupère la k-ième valeur; k valant $\sqrt{taillePop + tailleArchive}$. Pour chaque individu, on obtient alors D en effectuant le calcul $D = \frac{1}{sigmaK + 2}$.

Finalement on obtient la valeur de fitness F, en sommant D et R pour chaque individu.

2.3. Sélection de l'environnement

L'étape de sélection d'environnement permet de faire l'opération de mise à jour de l'archive. Dans celle-ci la première chose à faire est de trouver la population non-dominée, c'est à dire celle où $F < 1$.

Ensuite en fonction de la taille de la population non-dominée, qu'on stocke dans le tableau nextGenArchive, on effectue deux opérations différentes:

- ➔ Si elle est inférieure à la taille max de l'archive, on trie la population par valeur de fitness et on garde les meilleurs pour que nextGenArchive ai la même taille que la taille max de l'archive.
- ➔ Si elle est supérieure à la taille max de l'archive, on tronque nextGenArchive itérativement jusqu'à ce qu'elle ai la même taille que la taille max de l'archive.

Enfin on renvoie nextGenArchive qui devient la nouvelle archive.

2.4. Croisement et mutation

La sélection du mating pool se fait par tournoi binaires. Les individus sont ensuite croisés avec la méthode Simulated Binary et mutés avec la méthode Polynomiale. La sélection, le croisement et la mutation implémentés sont identiques à ceux utilisés lors du TP1.

3. NSGA-II

Pour l'algorithme NSGA-II, nous nous sommes basés sur l'article à propos de NSGA-II et avons décidé de faire un algorithme réel uniquement avec un croisement de type Simulated Binary et une mutation de type Polynomiale.

3.1. Initialisation et évaluation

L'algorithme NSGA-II fait appel à la fonction d'initialisation (NGSA2) qui prend en paramètre les choix utilisateurs et les paramètres de l'algorithme et du problème. On génère ensuite une population aléatoire en remplissant la colonne *"Val"* avec n valeurs; n étant le nombre de variables de décision.

L'étape de l'évaluation, quant à elle, permet de remplir la colonne "*ValObjective*" de chaque membre de la population en se basant sur le problème défini par l'utilisateur.

3.2. Premier tri de la population et boucle

Une fois la première population générée et évaluée, on va la trier. Ce qu'on appelle ici "tri" correspond à la succession des 3 fonctions suivantes : *NonDominatedSorting*, *CrowdingDistance* et *SortPopulation*. Chacune de ces fonctions va être expliquée plus en détails par la suite. Une fois le tri de la première population terminée, on rentre dans la boucle de génération de populations ; cette dernière se terminant quand on a atteint le nombre de générations maximum définies.

3.2.1. NonDominatedSorting

Le but de cette fonction est d'attribuer un rang (ou front) à chaque membre d'une population. Elle se déroule en deux étapes.

La première étape consiste à parcourir chaque membre de la population (boucle for de la taille de la population) et d'évaluer leur domination par rapport aux autres membres de la population (concept de la domination expliquée dans la partie SPEA2). Il y a alors deux possibilités :

- si le membre évalué domine un autre membre de la population, alors on rajoute cet autre membre dans une liste de domination propre au membre évalué (*DominationSet*).
- si le membre évalué est dominé par un autre membre, alors on incrémente la valeur *DominatedCount* également propre au membre évalué.

Cette première étape se finit par la création d'un front pareto ; ce dernier composé des membres n'étant dominé par aucun autre membre (*DominatedCount = 0).* Ces membres forment donc le rang 1.

La deuxième partie consiste à attribuer un rang à tous les membres de la population ne faisant pas partie du rang 1. De plus, on va remplir rang par rang Pour cela, on crée une boucle infinie où :

- on parcourt les membres du dernier rang rempli (au départ il s'agit du rang 1)
    - puis pour chaque membre, on parcourt sa liste de domination (*DominationSet*)
        - puis pour chaque membre de cette liste, on décrémente sa valeur *DominatedCount*
        - et si *DominatedCount* = 0, alors on lui attribue la valeur du dernier rang +1 (2 au début) et on ajoute ce membre dans un tableau Q.
- on teste ensuite la condition de sortie de la boucle infinie qui est : isempty(Q), c'est-à-dire si Q est vide.
- si on est toujours dans la boucle :
    - les membres du tableau Q deviennent les membres du rang suivant
    - et on incrémente la valeur du rang.

Ainsi à la fin de ces deux étapes, chaque membre de la population appartient à un rang et possède une liste de domination (*DominationSet*).

3.2.2. CrowdingDistance

Le but de cette fonction est de trier les membres appartenant à un même front par rapport à une distance qu'on appelle : Crowding Distance.

Pour cela, on parcourt chaque front et :

- on initialise la valeur de la *CrowdingDistance* de chaque membre du front à 0.
- puis on récupère les valeurs objectives de chaque membre
- et pour chaque valeur objective :
    - on trie ces valeurs par ordre décroissant, puis :
        - la crowding distance du premier et du dernier membre de ce tri vaut *Inf* (infini)
        - et la crowding distance des autres membres vaut la formule suivante : crowdingdistanceMembre + (valMembrePre-valMembreSuiv) / (valPremierMembre-valDernierMembre)

3.2.3. SortPopulation

Ici, on utilise les résultats obtenus à l'aide des deux fonctions précédentes pour trier chaque membre de la population d'abord en fonction de son rang, puis en fonction de sa crowding distance.

3.3. Croisement et mutation

La première étape de la boucle consiste à générer des enfants puis de les muter à partir de la population actuelle. La sélection du mating pool se fait par tournoi binaires. Les individus sont ensuite croisés avec la méthode Simulated Binary et mutés avec la méthode Polynomiale. La sélection, le croisement et la mutation implémentés sont identiques à ceux utilisés lors du TP1.

3.4. Deuxième tri de la population et troncature

Une fois qu'on a obtenu des mutants, on commence par réévaluer la valeur objective de chaque membre car celle-ci peut ne plus être cohérente avec les valeurs des membres.

Ensuite, on rassemble les mutants et la population actuelle dans un tableau nommé *R*.

Puis, on effectue les étapes du tri sur R : on obtient donc un tableau R où tous les membres sont triés en fonction du rang et de la crowding distance.

Enfin, on tronque R en ne gardant que les N premiers membres (N = taille d'un population) pour former la génération suivante P.

3.5. Troisième tri de la population et front pareto

Ayant tronqué R, la liste de domination (*DominationSet)* de chaque membre n'est plus cohérente. Donc pour s'assurer que tout reste cohérent, on effectue une troisième fois les étapes de tri sur la nouvelle génération.

Enfin, on stocke tous les membres de rang 1 dans le front pareto *F* et on incrémente la valeur de la génération avant de retourner au début de la boucle.

III - Résultats

Pour les tests, nous avons utilisé les paramètres suivants :

Taille de population N	*100*
Taille archive NA *(spea2)*	*100*
Gmax	*100*
Proba de croisement pc	*0.7*
Proba de mutation pm	*0.3*

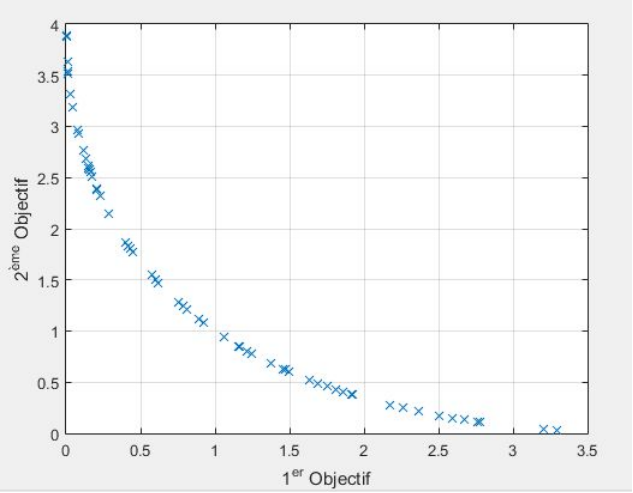
Nous testons les résultats en affichant le front de pareto sur un graphique mettant en lien les deux objectifs du problème.

Pour la vérification des graphiques, nous avons comparé nos résultats à ceux du wiki suivant : https://en.wikipedia.org/wiki/Test_functions_for_optimization.

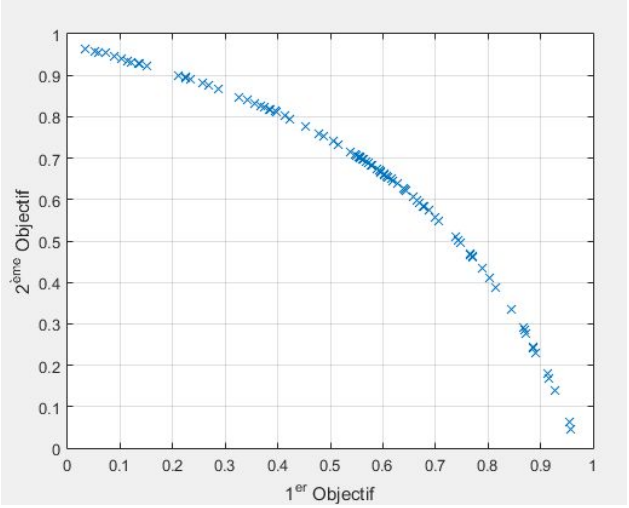
1. SPEA2

Ci dessous les résultats de l'algorithme SPEA2 pour chacun des problèmes, avec les paramètres listés plus haut.

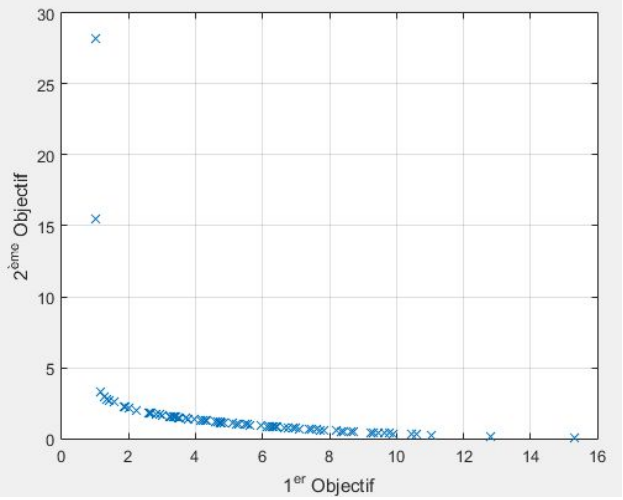
- **Fonction SCH (Schaffer) :**



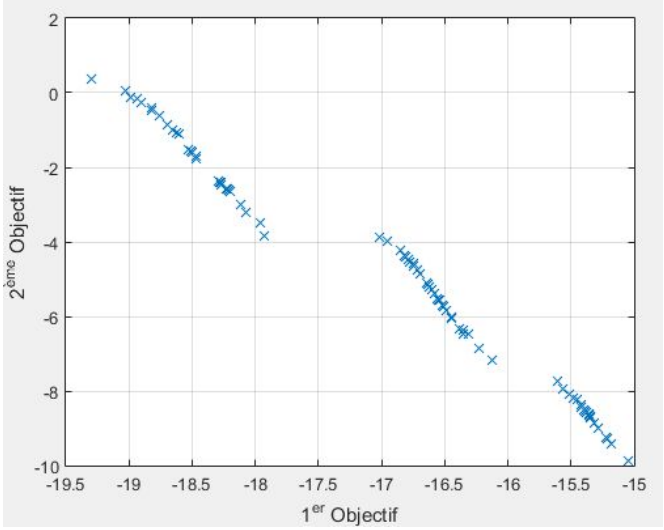
- **Fonction FON (Fonseca-Fleming) :**



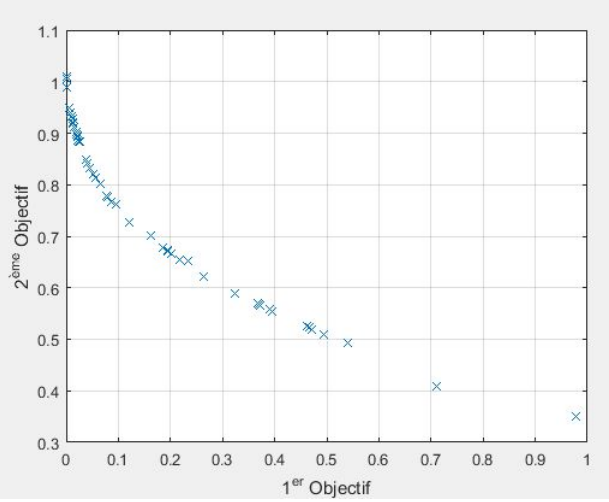
- **Fonction POL (Poloni) :**



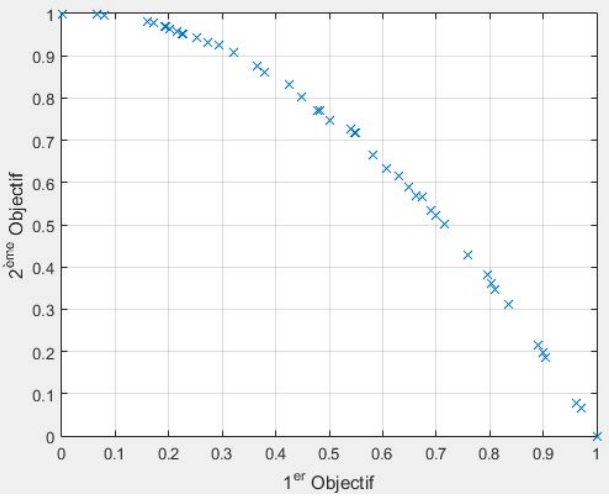
- **Fonction KUR (Kursawe) :**



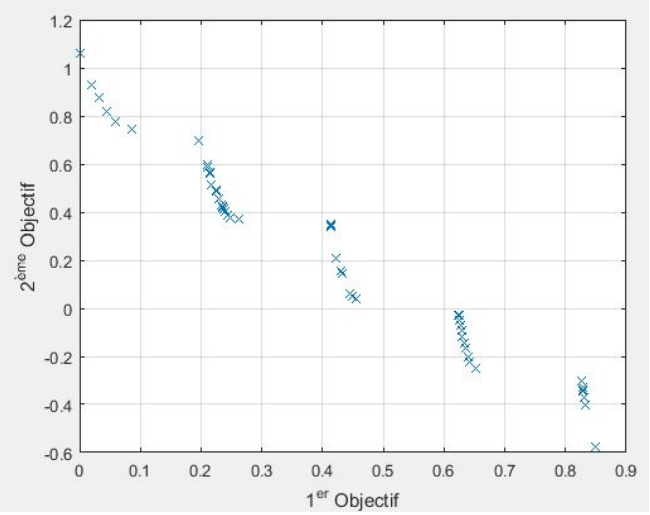
- **Fonction ZDT1 (Zitzler–Deb–Thiele 1) :**



- **Fonction ZDT2 (Zitzler–Deb–Thiele 2) :**



- **Fonction ZDT3 (Zitzler–Deb–Thiele 3) :**

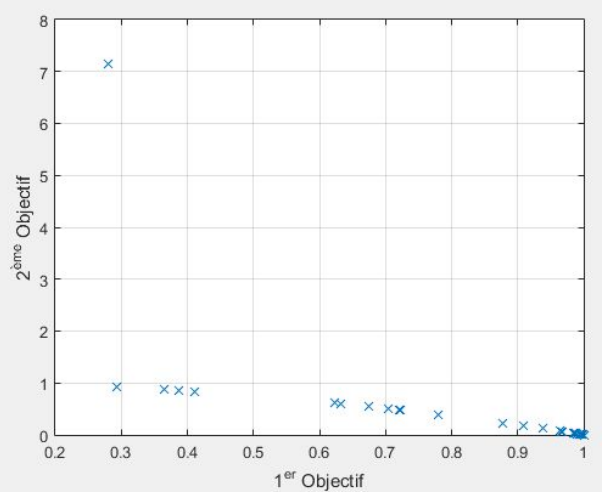


- **Fonction ZDT4 (Zitzler–Deb–Thiele 4) :**

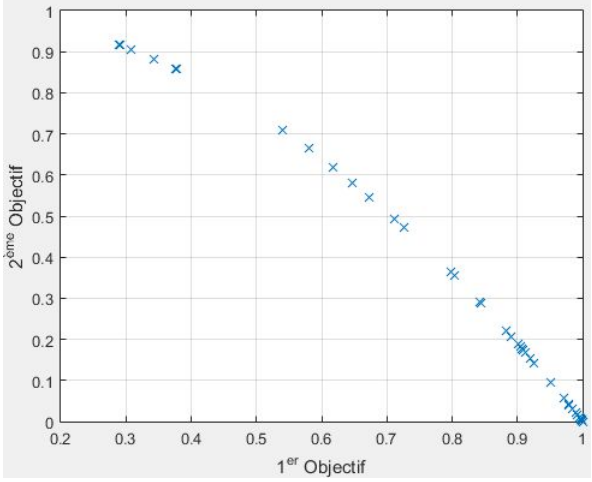
Pour cette fonction, nous avons eu un problème de partie réelles et imaginaires. Nous n’avons pas trouvé d'où venait le problème.

- **Fonction ZDT6 (Zitzler–Deb–Thiele 6) :**

Pour ce problème il faut beaucoup plus de générations pour obtenir un résultat cohérent. Ci-dessous le front après 100 générations :



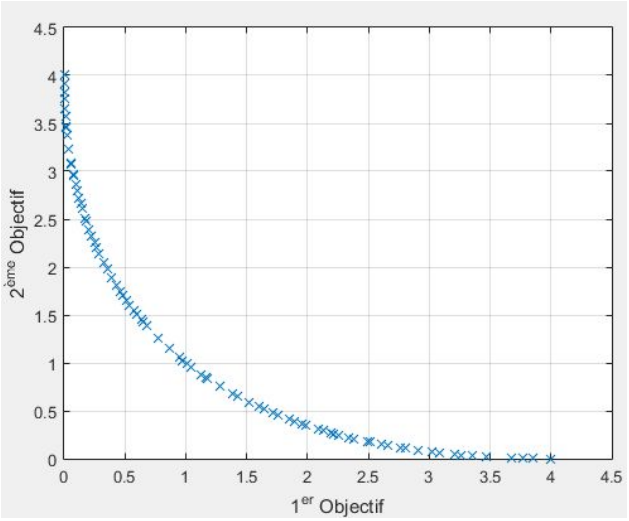
Et en comparaison, après 200 générations :



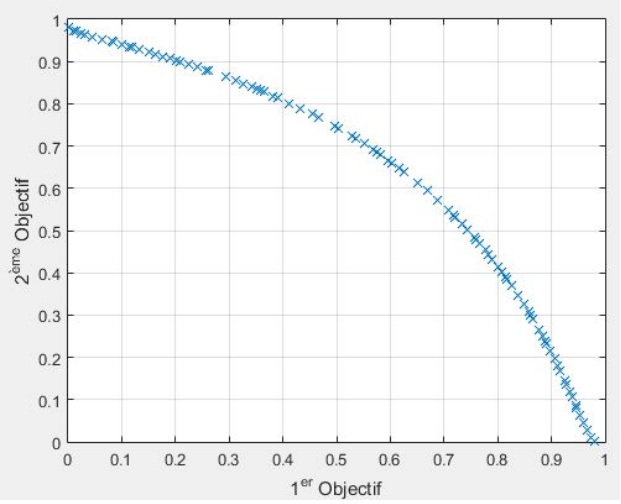
2. NSGA-II

Ci dessous les résultats de l'algorithme NSGA-II pour chacun des problèmes, avec les paramètres listés plus haut.

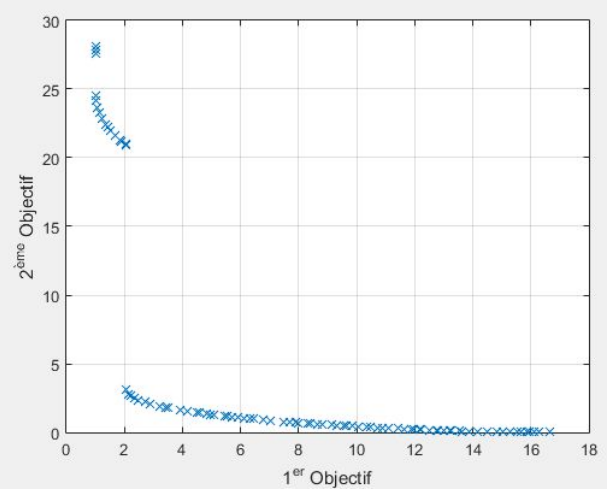
- **Fonction SCH (Schaffer) :**



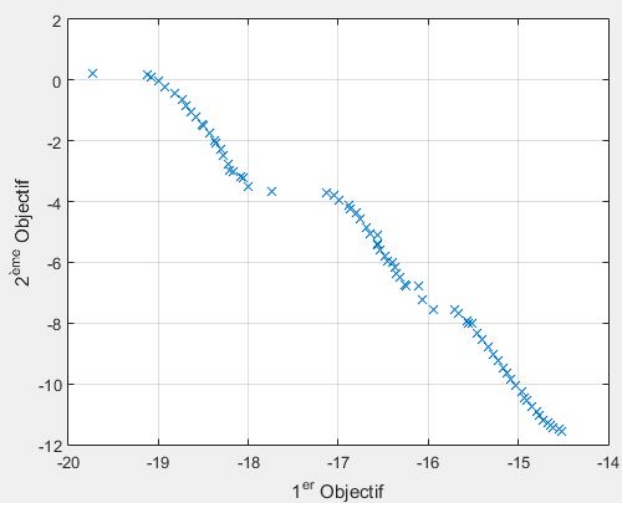
- **Fonction FON (Fonseca-Fleming) :**



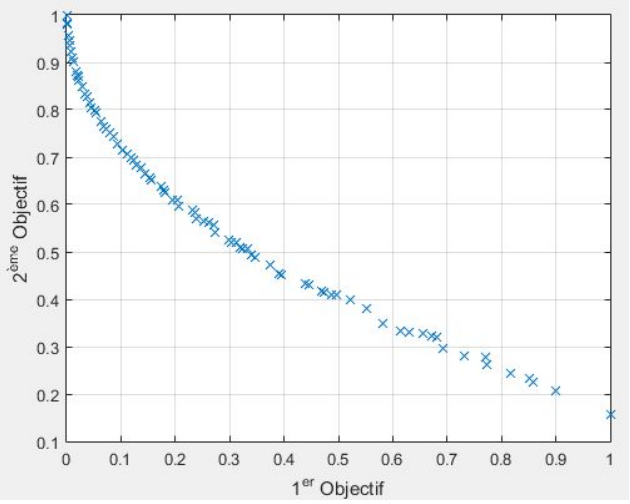
- **Fonction POL (Poloni) :**



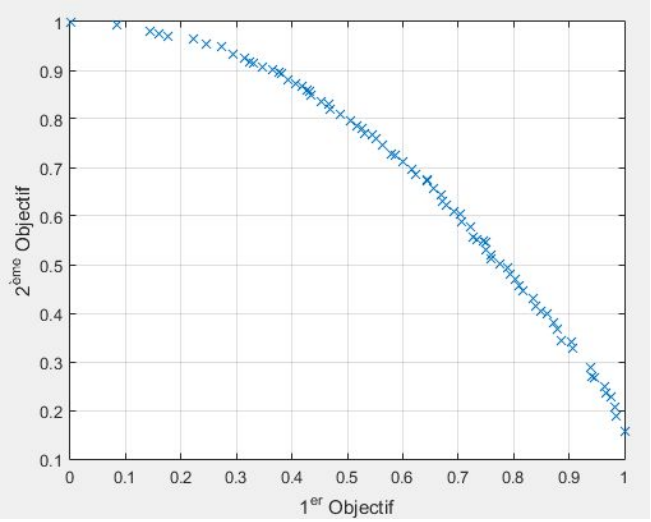
- **Fonction KUR (Kursawe) :**



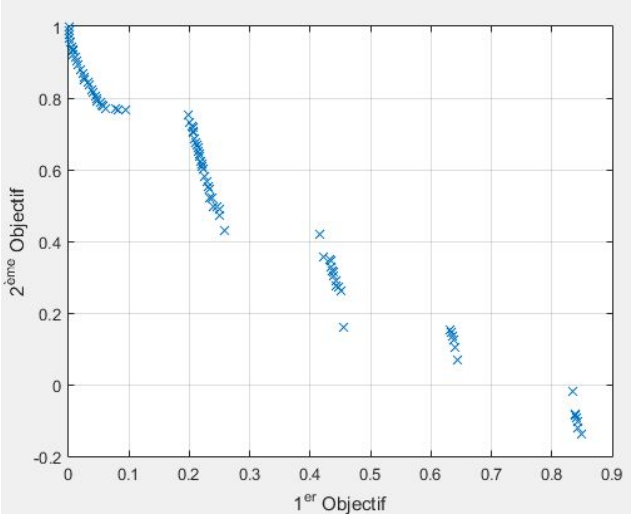
- **Fonction ZDT1 (Zitzler–Deb–Thiele 1) :**



- **Fonction ZDT2 (Zitzler–Deb–Thiele 2) :**



- **Fonction ZDT3 (Zitzler–Deb–Thiele 3) :**

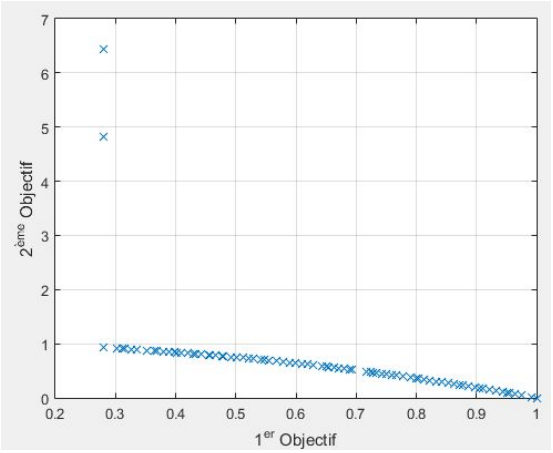


- **Fonction ZDT4 (Zitzler–Deb–Thiele 4) :**

Pour cette fonction, nous avons eu un problème de partie réelles et imaginaires. Nous n'avons pas trouvé d'où venait le problème.

- **Fonction ZDT6 (Zitzler–Deb–Thiele 6) :**

On retrouve bien la courbe de ZDT6 entre les points (0,1) et (1,0). Cependant, il y a du bruit en plus causé par des points (0,5) ou (0,7).



IV - Comparaison

Quand on compare les deux algorithmes, on constate que NSGA-II trouve un front de pareto acceptable au bout de moins de générations que SPEA2. Ci-dessous un tableau résumant le nombre de générations après lesquelles on obtient un “bon” front pour chaque algorithme et les premiers problèmes :

	SPEA2	NSGA-II
SCH	30	18
FON	30	20
POL	35	22
KUR	100	50

On remarque cependant que le temps de traitement de chaque génération est beaucoup plus long avec NSGA-II que SPEA2. Enfin, on constate que l’allure de la courbe est plus proche du résultat attendu avec NSGA-II qu’avec SPEA2.