Chapitre 6

Algorithmes évolutionnaires

Le concept de l'évolution en biologie est centré sur la capacité des individus à s'adapter aux changements environnementaux. Les plus aptes sont retenus grâce à des processus tels que la sélection naturelle, la reproduction, la mutation, la compétition, la symbiose ou l'immigration (chapitres 4 et 5). Les algorithmes évolutionnaires utilisent une population d'individus gouvernés par les opérateurs traduisant ces mécanismes (pages 34, 82 et 85). Le choix et l'ordre dans lequel ces opérateurs s'effectuent a donné naissance à plusieurs algorithmes évolutionnaires. Par exemple, combinaisons, mutations et sélections ont donné naissance à l'algorithme génétique (GA), à la programmation génétique (avec une représentation différente de l'individu par rapport au GA), à la programmation évolutive (simulant le comportement adaptatif phénotypique), aux stratégies évolutionnaires (simulant les règles d'évolution ou l'évolution de l'évolution) et à l'évolution différentielle. À cette liste on pourra rajouter l'optimisation basée sur la biogéographie qui simule la répartition géographique des espèces via la migration et l'émigration. La co-évolution qui s'inspire de l'évolution de deux populations (donc deux fonctions objectifs différentes) en relation de symbiose (coopération) ou prédateur-proie (compétition) représente une autre classe d'algorithme évolutionnaire [7]. Dans ce chapitre, nous avons choisi de présenter l'algorithme génétique, l'évolution différentielle et l'optimisation basée sur la biogéographie afin d'illustrer les similitudes entre des méthodes basée sur les mêmes opérateurs.

6.1 Algorithme génétique (GA)

L'algorithme génétique (genetic algorithm, GA) est un algorithme d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle : croisement, mutation et sélection. L'algorithme génétique a

déjà une histoire relativement ancienne puisque les premiers travaux de John Holland sur les systèmes adaptatifs remontent à 1962 [14], et de nombreuses variantes de cet algorithme ont été développées par la suite.

6.1.1 Étapes de l'algorithme génétique

Au début de l'algorithme génétique, une population de N parents est générée aléatoirement et leurs qualités sont évaluées par le biais de la fonction objectif. Par analogie avec le monde du vivant, les variables de décisions sont appelées « chromosomes » et un parent /individu transmet un ensemble de chromosomes donc de caractères à une descendance. Le GA est un algorithme itératif évolutionnaire reposant sur les étapes suivantes :

1. *Combinaison / croisement* : des parents choisis aléatoirement sont recombinés (croisés, crossover) pour donner une nouvelle génération de N éléments. Le nombre de descendants peut être différent de N.
2. *Mutation* : ces nouveaux éléments sont pseudo-aléatoirement mutés selon des lois de probabilité définies.
3. *Sélection* : les N meilleur éléments des parents et des enfants sont choisis et deviennent les nouveaux parents. Cette approche est qualifiée d'élitiste. Une sélection uniquement dans la population des enfants est non élitiste.

Les opérateurs combinaison, mutation et sélection ont été présentés pages 86, 84 et 35. Les opérateurs supplémentaires de recherche locale (page 96) et antistagnation (page 95) peuvent être intégrés à la méthode, suivant la topologie de la fonction à minimiser. Les sorties de l'espace de recherche sont traitées par application de l'opérateur décrit Sec. 2.2, page 32.

Les seuls paramètres exogènes des méthodes décrites ici sont le nombre $N = 30$ de vecteurs de variables de décision, le nombre maximum $T = 100 \times D$ d'itérations permises. Les paramètres endogènes sont la dimension D et les bornes du domaine de recherche S ; ils sont définis pour chaque fonction de test (Sec. 3.2.2, page 53), comme la valeur acceptable VA (page 52) qui, si elle est atteinte, permet d'arrêter la boucle d'optimisation avant d'atteindre T itérations.

6.1.2 Variantes de l'algorithme génétique

La combinaison, la mutation et la sélection sont programmés sous forme de fonctions décrivant chaque opérateur. Ajuster leurs paramètres d'entrée permet de sélectionner les méthodes. Les taux de succès obtenus pour chacune des fonctions de test sont donnés pour chaque variante et pour $D = 2$ dans

un premier temps. On indiquera l'incertitude sur le taux de réussite entre parenthèses (page 44). Nous proposons trois variantes de la métaheuristique génétique (GA1, GA2 et GA3). Le code de GA3 est donné.

GA1 combinaison par échange de gènes, croisement (crossover) :

```
[xc1,xc2]=OperateurCombGen(x,x,0*x,N,D,[],[],1,'
    ea1',[],[],[],[]);
```

Deux vecteurs sont produits, chacun est muté et évalué. La mutation de toutes les variables de décision est réalisée en diminuant linéairement l'amplitude de mutation ($'t'$) (ou suivant une loi sigmoïde ($'s'$) décrite page 83, avec $k = 10$) et suivant deux lois de probabilités (uniforme ($'U'$) et normale ($'N'$)).

```
  xm1=OperateurMutGen(xc1,N,D,1,1,'U','s',t,T);
  xm2=OperateurMutGen(xc2,N,D,1,1,'N','t',t,T);
[xm1,sd]=GestionSortieDomaine(endogenes,xm1, ...
        N,D,sd);
[xm2,sd]=GestionSortieDomaine(endogenes,xm2, ...
        N,D,sd);
  fm1=feval(FonctionObjectif,xm1,N,D);
  fm2=feval(FonctionObjectif,xm2,N,D);
  neval0=neval0+2*N;
[x,f,mx,mf]=OperateurSelection([xm1;xm2], ...
        [fm1;fm2],N,mx,mf,'t');
```

Le taux de réussite est de 100% (ou voisin) pour les fonctions F_0 à F_7, F_9, F_{10} et F_{12}. Par contre, il est proche de 84%(7) pour F_8 mais seulement de 9(7) % pour F_{11} et de 1%(2) pour F_{13}. Ces résultats montrent que la méthode est peu efficace pour trouver un optimum global « caché » dans un puits (F_{11} et F_{13}). Elle est moyennent efficace pour sortir des puits d'optimums locaux (F_8).

GA2 la mutation porte sur des variables de décision tirées aléatoirement.

```
xm1=OperateurMutGen(xc1,N,D,0,1,'U','s',t,T);
xm2=OperateurMutGen(xc2,N,D,0,1,'N','t',t,T);
```

Le taux de réussite passe à 100% pour F_{11} et F_{13}. Par contre, il est fortement dégradé pour F_1 et F_8 : 0%, Ces résultats montrent que GA2 est plus efficace que GA1 pour trouver un optimum global « caché » dans un puits, mais elle est inefficace pour sortir des puits d'optimums locaux (F_8) et pour trouver un optimum global situé dans un large puits (F_1).

GA3 utilise le même paramétrage que GA2 et les opérateurs anti-stagnation et de recherche locale sont ajoutés.

```
1 function [mf,mx,sd,neval0,Conv,t]= ...
```

```
2            GA3Operateur(endogenes,exogenes)
3  FonctionObjectif=endogenes.FonctionObjectif;
4  VA=endogenes.VA; D=endogenes.D;
5  S=[endogenes.BornesInf; endogenes.BornesSup];
6  N=exogenes.N; T=exogenes.maxIterations;
7  test2D=exogenes.test2D;
8  Conv=zeros(T,1); sd=0; neval0=0;
9  x=OperateurGenerationAleatoire(S,N,D);
10 f=feval(FonctionObjectif,x,N,D); neval0=neval0+N;
11 [x,f,mx,mf]=OperateurSelection(x,f,N, ...
12                  zeros(1,D),Inf,[]);
13 memfval=f;
14 t=1;
15 while t<=T && mf>=VA
16  TestAlgorithme;
17  [xc1,xc2]=OperateurCombGen(x,x,0*x,N,D,[],[], ...
18                             1,'ea1',[],[],[],[]);
19  xm1=OperateurMutGen(xc1,N,D,0,1, ...
20                          'U','s',t,T);
21  xm2=OperateurMutGen(xc2,N,D,0,1, ...
22                          'N','t',t,T);
23  [xm1,sd]=GestionSortieDomaine(endogenes,xm1, ...
24                                N,D,sd);
25  [xm2,sd]=GestionSortieDomaine(endogenes,xm2, ...
26                                N,D,sd);
27  fm1=feval(FonctionObjectif,xm1,N,D);
28  fm2=feval(FonctionObjectif,xm2,N,D);
29  neval0=neval0+2*N;
30  [x,f,mx,mf]=OperateurSelection([xm1;xm2], ...
31                    [fm1;fm2],N,mx,mf,'t');
32  memfval=[memfval f];
33  [mx,mf,neval0]= ...
34    OperateurRechercheLocale(endogenes,mx,mf, ...
35            1,D,neval0);
36  [x,f,neval0,memfval]= ...
37   OperateurAntiStagnation(endogenes,x,f, ...
38                    memfval,N,D,t,T,neval0);
39   Conv(t)=mf;
40   t=t+1;
41 end
42 end
```

En dimension $D = 2$, les résultats sont présentés dans les tables 6.1 et 6.2. Tous les taux de succès sont de 100% sauf pour F_0, F_8, F_{11} et F_{13}.

F	Succès (%)	Moyenne sorties	Moyenne iter.	Moyenne temps (s)	Moyenne éval.	Écart-type éval.
F_0	99(2)	4	7	1.8e-02	411	517
F_1	100(0)	0	43	6.8e-02	2562	1478
F_2	100(0)	5	50	8.0e-02	2943	1808
F_3	100(0)	0	4	8.4e-03	195	188
F_4	100(0)	4	3	6.8e-03	134	85
F_5	100(0)	0	12	2.1e-02	678	228
F_6	100(0)	0	1	4.1e-03	30	0
F_7	100(0)	0	19	3.3e-02	1140	367
F_8	78(8)	1	24	1.0e-01	1411	646
F_9	100(0)	0	1	3.9e-03	30	0
F_{10}	100(0)	2	166	3.6e-01	9905	1538
F_{11}	10(6)	0	43	3.0e-01	2550	2094
F_{12}	100(0)	1	22	4.2e-02	1297	570
F_{13}	0(0)	4688	–	3.2e-01	–	–
Mémoire utilisée 272477 octets.						

TABLE 6.1 – Caractérisation de GA1, D = 2, N = 30, T = 200.

F	Succès (%)	Moyenne sorties	Moyenne iter.	Moyenne temps (s)	Moyenne éval.	Écart-type éval.
F_0	38(10)	0	3	3.8e-01	145	113
F_1	97(3)	0	51	1.4e-01	3313	1455
F_2	100(0)	0	10	2.6e-02	596	176
F_3	100(0)	0	2	5.8e-03	77	27
F_4	100(0)	0	2	6.1e-03	82	24
F_5	100(0)	0	2	7.7e-03	112	27
F_6	100(0)	0	1	4.0e-03	30	0
F_7	100(0)	0	5	1.4e-02	286	139
F_8	45(10)	0	17	3.7e-01	1171	831
F_9	100(0)	0	1	4.0e-03	30	0
F_{10}	100(0)	0	23	8.6e-02	1435	515
F_{11}	100(0)	0	4	1.1e-02	203	82
F_{12}	100(0)	0	5	1.6e-02	276	153
F_{13}	100(0)	0	10	2.5e-02	582	345

F	Succès (%)	Moyenne sorties	Moyenne iter.	Moyenne temps (s)	Moyenne éval.	Écart-type éval.
Mémoire utilisée 272476 octets.						

TABLE 6.2 – Caractérisation de GA3, D = 2, N = 30, T = 200.

Les figures 6.1- 6.2 permettent de visualiser la convergence moyenne des méthodes GA1 et GA3 pour chaque fonction de test.

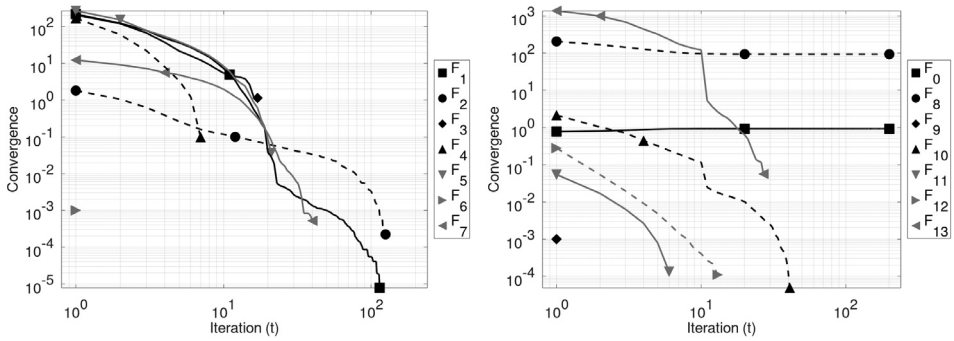

FIGURE 6.1 – Valeurs moyennes de la convergence de la méthodes GA1 pour $D = 2$ sur les 168 réalisations, pour chaque fonction de test.

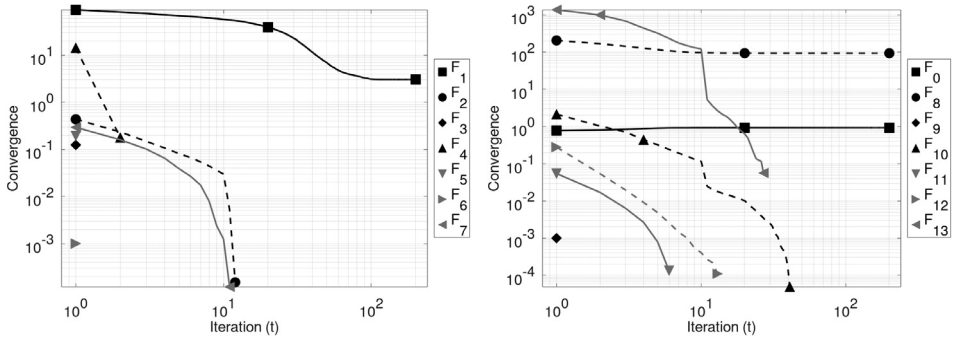

FIGURE 6.2 – Valeurs moyennes de la convergence de la méthode GA3 pour $D = 2$ sur les 168 réalisations, pour chaque fonction de test.

La méthode GA3 permet une convergence plus rapide. Elle nécessite en général moins de cinq itérations en dimension $D = 2$. Le paramétrage des opérateurs de combinaison et de mutation conditionne le taux de succès pour les différentes fonctions de test. L'ordre des opérateurs influe également.

L'opérateur de recherche locale peut être appliqué à tous les vecteurs plutôt qu'au seul meilleur. De plus, l'analyse des résultats montre que l'association judicieuse d'opérateurs anti-stagnation et recherche locale ne permet pas de résoudre tous les problèmes. Le « no free lunch theorem » est vérifié (page 50). Il est possible d'anticiper les résultats en utilisant les résultats de caractérisation des opérateurs (Sec. 5.2, page 88), en activant $test2 = 1$.

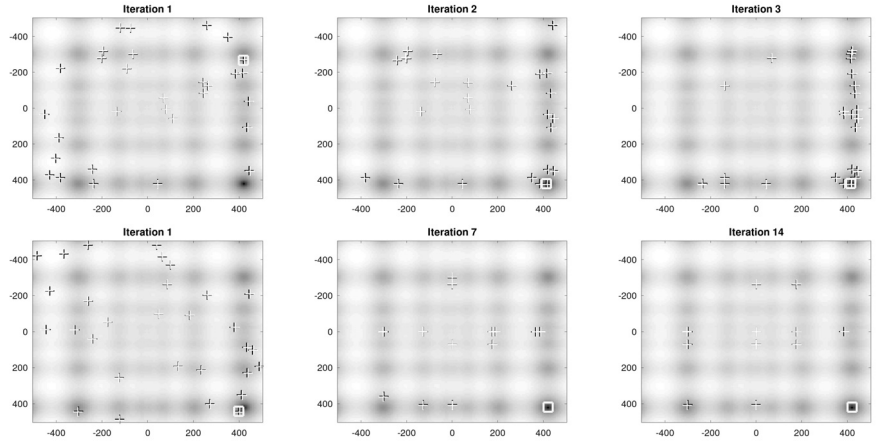

FIGURE 6.3 – Positions successives des variables de décision des méthodes GA1 et GA3 pour $D = 2$ et la fonction de test F_8.

La comparaison des méthodes GA1 et GA3 en dimension $D = 30$ (tables 9.4-9.8, pages 203 et suivantes) montre qu'une hybridation utilisant les opérateurs de recherche locale et anti-stagnation améliore les taux de succès pour certaines fonctions mais le dégrade pour d'autres. La comparaison avec les résultats en dimension $D = 2$ et $D = 30$ montrent que lorsque le taux de succès n'est pas 100% en $D = 2$, il peut être dégradé à 0% pour $D = 30$. Trouver l'optimum global est encore plus difficile lorsque la dimension du problème augmente. Cette difficulté est également observable lorsque le nombre d'itérations et d'évaluations augmente. L'activation de la représentation graphique en 2D (grâce à *TestAlgorithme.m* page 72) permet d'illustrer la discussion précédente pour la fonction de test F_8 par exemple. La dernière itération représentée est celle qui a vu la méthode converger ($t = 3$ pour GA1 et $t = 14$ pour GA3) .

Exercice II.3. L'opérateur mutation (*OperateurMutGen.m*, page 84) permet 24 variantes, sans introduire de probabilité de mutation *pm* comprise entre 0 et 1 . L'opérateur de combinaison (*OperateurCombGen.m*, page 86) permet également 120 variantes, sans introduire de probabilité de mutation *pc* comprise entre 0 et 1. Rappelons que *GestionSortieDomaine.m* (page 33) tire

au sort parmi trois méthodes pour remplacer des vecteurs qui sortiraient du domaine après combinaison ou mutation. En fonction des résultats de caractérisation de ces opérateurs (Secs. 5.2.3 et 5.2.4), déterminer les critères de performance de plusieurs associations d'opérateurs (précision, efficacité et robustesse définies pages I.18, I.17 et I.16).

6.2 Évolution Différentielle (DE)

L'évolution différentielle (en anglais Differential Evolution, DE) a été introduite par Storn et Price en 1997 [39]. Elle est inspirée par les algorithmes génétiques et utilise un opérateur de combinaison particulier. La combinaison en évolution différentielle exploite les différences géométriques entre des vecteurs solutions tirés au hasard (d'où l'origine de son nom). Dans la méthode initiale, il y a une mise à jour des solutions progressive à chaque itération.

6.2.1 Algorithme de l'évolution différentielle

Comme pour les autres métaheuristiques à population, on part d'une population générée aléatoirement de N parents notés x_1,...,x_N. Les qualités f_i des solutions sont évaluées par le biais de la fonction objectif. Ensuite, la boucle d'optimisation commence en répétant les étapes suivantes.

1. *Croisement ou combinaison* : pour i de 1 à N on génère un nouveau vecteur *trial* comme suit :
   (a) Pour chaque composante j, on génère une variable aléatoire uniforme *rand* entre 0 et 1
   (b) Si $rand < CR$ (CR étant le seuil de combinaison) alors

$$trial_j = x_{c,j} + FF \times (x_{a,j} - x_{b,j}), \tag{6.1}$$

   avec la constante $F \in [0, 2]$ qui contrôle l'amplification de la variation différentielle, et a, b et c sont 3 entiers aléatoires entre 1 et N deux à deux différents et différents de i.
   (c) Sinon

$$trial_j = x_{i,j}. \tag{6.2}$$

2. *Sélection* : la qualité du nouveau vecteur *trial* est évaluée et elle remplace x_i si elle présente une meilleure qualité et ceci pour chaque i au fur et à mesure.

L'opérateur d'évolution différentielle correspond à un paramétrage particulier de l'opérateur de combinaison présenté pages 86. La sélection est également