Chapitre 1

Vers une Programmation Orientée Objet des Optimiseurs Exemples avec Scilab

1.1. Introduction

Les algorithmes d'optimisation sont généralement implémentés selon un *paradigme fonctionnel* qui place la méthode d'optimisation au sommet de la hiérarchie. Pour prendre l'exemple du logiciel Scilab [CAM 06], on optimise un fonction `f` avec une instruction qui se charge de tout le processus d'optimisation, sans contrôle possible de l'utilisateur :

```
--> [yopt, xopt] = optim(f, x0)
```

Cette structure est très contraignante si l'on souhaite mettre en œuvre des stratégies d'optimisation plus souples, par exemple, pouvoir changer d'optimiseur en cours d'optimisation, estimer un ou plusieurs méta-modèles, etc.

Dans ce chapitre, nous proposons une organisation logicielle selon un *paradigme objet* où les optimiseurs et les modèles à optimiser (simulateurs, méta-modèles, etc.) sont des objets de même niveau hiérarchique. Avec ces objets, l'utilisateur pourra "jongler" pour se construire des stratégie d'optimisation personnalisées. Un tutoriel présentera les détails de l'implémentation d'un optimiseur à travers trois exemples.

Chapitre rédigé par Yann COLLETTE et Nikolaus HANSEN et Gilles PUJOL.

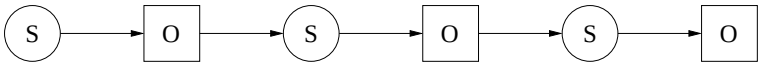


Figure 1.1. *Déroulement d'une procédure d'optimisation.*

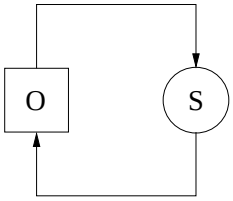


Figure 1.2. *Couplage d'un optimiseur et d'un simulateur.*

Enfin, des réalisations complètes seront présentées : la méthode du simplex, un algorithme génétique, et la stratégie évolutionnaire CMA-ES.

1.2. Découplage optimiseur – simulateur

Optimiser, c'est se déplacer dans un domaine : on explore itérativement de nouveaux points, en tâchant de localiser des points de bonne performance. Un algorithme d'optimisation décrit ce cheminement du début (point initial) à la fin (condition d'arrêt). Assez naturellement, on l'implémente avec une boucle au milieu de laquelle figure le ou les appel(s) à la fonction à optimiser. Cette boucle est encapsulée dans une fonction, dont la fonction à optimiser est un argument. Comme on vient de le voir avec l'exemple de la fonction `optim`, Scilab suit cette logique. Dans un contexte industriel, cette structure trouve rapidement ses limites. Comme chaque appel au simulateur coûte cher, il paraît déraisonnable de confier aveuglément cette ressource à une fonction d'optimisation. Que fait-on si le simulateur "plante", si les ressources en calcul distribué sont allouées à un calcul plus prioritaire, ou si l'administrateur système annonce une panne imminante ? Peu d'environnements de calcul sont aptes à gérer de telles situations.

Pour réconcilier Scilab avec la simulation industrielle, il est nécessaire de découpler d'une manière absolue le code qui réalise l'optimisation (que nous allons appeler "optimiseur") du code qui réalise la simulation (qui se trouve être le simulateur tel qu'il se présente naturellement). Pour y voir clair, observons le déroulement d'une procédure d'optimisation comme représentée sur le diagramme 1.1 : c'est une séquence d'opérations, avec régulièrement des appels au simulateur. Ainsi l'optimiseur se trouve être le morceau de programme entre deux séries de simulations. En dessinant le diagramme 1.1 sous la forme 1.2, les *objets* "optimiseur" et "simulateur" apparaissent clairement dans leur couplage.

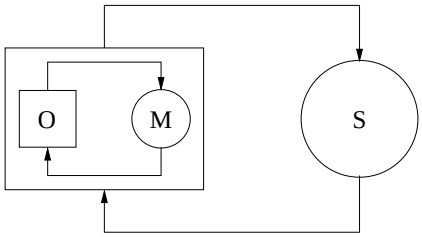


Figure 1.3. *Le couplage d'un optimiseur et d'un méta-modèle constitue un nouvel optimiseur, qui peut être couplé à un simulateur coûteux. Après chaque nouvelle simulation, le méta-modèle est ré-estimé. L'optimum trouvé sur ce méta-modèle est ensuite proposé au simulateur.*

Un intérêt de cette approche objet est de permettre d'envisager, de manière lisible, d'autres scénarios. Par exemple, le diagramme 1.3 représente une stratégie d'optimisation sur un méta-modèle, ce méta-modèle étant ré-estimé en cours d'optimisation grâce à de nouveaux appels au simulateur. On voit ici le caractère modulaire : le couplage d'un optimiseur et d'un méta-modèle forme un nouvel optimiseur, cet optimiseur étant couplé au simulateur. Un autre avantage est de stocker dans une structure bien identifiée l'état de l'optimisation à un instant donné. Ainsi, l'optimiseur peut être sauvegardé dans un fichier et rechargé un autre jour ou sur une autre machine. Cette capacité est structurante pour les stratégies de calcul distribué.

1.3. Le schéma "ask & tell"

Comme nous venons de le voir, l'optimiseur est le morceau de programme qui se trouve entre deux séries de simulations. On doit donc pouvoir communiquer avec un optimiseur en amont (*"quelles sont les simulations dont tu as besoin ?"*) et en aval (*"voici les simulations"*), ce que nous définissons respectivement par les méthodes `ask` et `tell`. Ainsi, un schéma d'optimisation doit pouvoir s'écrire en Scilab de la manière suivante :

```
while ~ opt.stop
  x = ask(opt)
  y = f(x)
  opt = tell(opt, x, y)
end
```

où `opt` est l'*objet* optimiseur. Une fois cette boucle finie, il faut pouvoir lire la valeur de l'optimum, ce qui est fait par la méthode `best` :

```
[yopt, xopt] = best(opt)
```

REMARQUE.– Il est important de noter la syntaxe particulière de la méthode `tell`, avec l'optimiseur présent aussi bien en entrée qu'en sortie. Cette syntaxe permet de mettre à jour l'état de l'optimiseur, car le passage des arguments en Scilab se fait uniquement par valeur.

La programmation objet en Scilab est possible grâce aux listes typées et à la surcharge des opérateurs, voir l'aide de Scilab aux rubriques `mlist` et `overloading`. Un optimiseur sera un objet (liste de type `mlist`), dont les méthodes associées, `ask`, `tell` et `best` devront être définies selon la syntaxe :

```
function x = %<optimizer_type>_ask(this)
function this = %<optimizer_type>_tell(this, x, y)
function [yopt, xopt] = %<optimizer_type>_best(this)
```

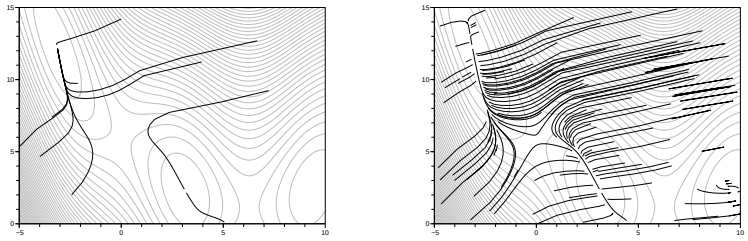
Ceci est possible grâce au mécanisme de *surcharge* des fonctions `ask`, `tell` et `best`. Le langage Scilab permet de surcharger la majorité des opérateurs (`+`, `-`, `()`, ...), ainsi que certaines fonctions internes (`disp`, `plot2d`, ...), mais pour pouvoir surcharger les méthodes `ask`, `tell` et `best`, il faut ajouter le code suivant qui permet de rediriger l'exécution vers le code spécifique de l'optimiseur :

```
function x = ask(this)
  execstr('x = %' + typeof(this) + '_ask(this)')
endfunction

function this = tell(this, x, y)
  execstr('this = %' + typeof(this) + '_tell(this, x, y)')
endfunction

function [yopt, xopt] = best(this)
  execstr('[yopt, xopt] = %' + typeof(this) + '_best(this)')
endfunction
```

En plus de ces trois fonctions, il faudra implémenter un *constructeur*, c'est-à-dire une fonction qui se chargera d'initialiser l'objet optimiseur. Tout ceci est illustré en détail dans la suite du chapitre.

1.4. Exemple : une stratégie "multistart"

Cette logique de programmation objet a un grand avantage : elle permet d'envisager très simplement des scénarios d'optimisation assez évolués. Nous illustrons ceci



(a) (b)

Figure 1.4. *Stratégies d'optimisation multistart. En (a), les itérations sont lancées indépendamment les unes des autres. En (b), la trajectoire d'un optimiseur est interrompue lorsqu'elle recoupe celle d'un précédent ; et un nouvel optimiseur part d'une autre région du domaine.*

sur une stratégie "multistart" qui serait assez difficile à implémenter avec la logique de programmation fonctionnelle traditionnelle.

La stratégie multistart la plus naïve consiste à lancer successivement N procédures d'optimisation (de type descente) en des points initiaux différents. Le but est de trouver un optimum global, comme optimum des optima locaux. En supposant que chaque procédure d'optimisation locale se déroule en n pas, la stratégie multistart coûte au total de $N \times n$ évaluations de la fonction coût. Cette stratégie appliquée sur la fonction de Branin-Hoo (voir page 8) est illustrée sur la figure 1.4(a). On constate que :

1) si un optimiseur met moins de n simulations pour converger, on n'utilise pas les simulations restantes ; on n'exploite donc pas au maximum le budget de simulations ;

2) les trajectoires des différents optimiseurs se regroupent, et on explore des trajectoires déjà explorées auparavant ; il y a donc gaspillage du budget de simulations.

On va donc implémenter la stratégie alternative suivante : on stoppe un optimiseur s'il arrive trop près d'une trajectoire d'un autre optimiseur, et, avec les simulations restantes, on relance un autre optimiseur d'une région non explorée. En utilisant l'optimiseur "ask & tell" `steepdesc` (méthode de la plus grande pente) que nous programmerons au paragraphe 1.5.3, la stratégie multistart s'implémente en quelques lignes :

```
neval = 1000  // budget de simulations
Arch = []     // archive des points déjà visités
i = 0         // numero de l'optimiseur courant
// TANT QUE LE BUDGET DE SIMULATIONS N'EST PAS ÉPUISÉ,
while neval > 0
```

```
  // INITIALISER UN NOUVEL OPTIMISEUR;
  i = i + 1
  opt = steepdesc()
  opt.x0 = grand(1, 2, 'def') .* (xmax - xmin) + xmin
  opt.step = 5E-2
  opt.eps = 1E-2
  // ALLOUER UNE PARTIE DU BUDGET À L'OPTIMISEUR;
  opt.eval = min(100, neval)
  neval = neval - opt.eval
  // TANT QUE L'OPTIMISEUR N'EST PAS ARRÊTÉ,
  while ~ opt.stop
    // DEMANDER UN POINT À L'OPTIMISEUR;
    x = ask(opt)
    // CALCULER LA DISTANCE DE CE POINT AUX POINTS DE L'ARCHIVE;
    Xj = Arch(Arch(:,3) ~= i, 1:2)
    d = min(sqrt(sum( (Xj - x(ones(1,size(Xj,1)),:)).^2 , 'c')))
    // SI CETTE DISTANCE EST TROP PETITE,
    if d < 0.1 then
      // STOPPER L'OPTIMISEUR;
      opt.stop = %t
    else
      // SINON, LANCER LA SIMULATION, ...
      [y, dy] = branin(x)
      // ..., METTRE À JOUR L'OPTIMISEUR, ...
      opt = tell(opt, x, list(y, dy))
      // ET AJOUTER LE POINT À L'ARCHIVE;
      Arch($+1,1:3) = [x, i]
    end
  end
  // RESTITUER LE BUDGET NON UTILISÉ PAR L'OPTIMISEUR;
  neval = neval + opt.eval
end
```

Le résultat de l'exécution est représenté sur la figure 1.4(b). On voit que l'on exploite mieux le budget de simulations, le domaine est beaucoup mieux couvert qu'en 1.4(a).

Il est important de noter que cette stratégie aurait été très difficile (mais possible) à implémenter en utilisant la fonction `optim` de Scilab, la difficulté étant d'arrêter un optimiseur en cours d'optimisation. Ici, nous l'avons fait tout simplement avec la commande `opt.stop = %t`.

1.5. Programmer un optimiseur “ask & tell” : tutoriel

Maintenant que l’on a vu à quoi ressemble un optimiseur “ask & tell” et comment l’utiliser pour construire des stratégies d’optimisation personnalisées, nous présentons l’implémentation de trois optimiseurs “ask & tell”. Le but n’est pas de présenter des méthodes d’optimisation efficaces (voir pour cela les paragraphes suivant), mais de proposer un tutoriel pour s’entraîner. Ainsi, les méthodes exposées illustrent les points suivants :

1) recherche aléatoire : objets optimiseurs et méthodes associées ;

2) $(\mu/\mu, \lambda)$-ES : condition initiale, optimiseur à densité de points ;

3) méthode de la plus grand pente : réponses multiples du modèle, optimiseurs ayant différents états, condition d’arrêt composite.

1.5.1. *Exemple 1 : recherche aléatoire*

La recherche aléatoire est certainement l’algorithme d’optimisation le plus simple : à chaque itération on tire un point aléatoirement dans le domaine, et on le conserve s’il est meilleur que le meilleur point exploré auparavant. Cette approche peut sembler naïve, mais c’est la base de tous les algorithmes de recherche stochastique, dont le recuit simulé, les algorithmes évolutionnaires, etc. C’est l’algorithme de comparaison par excellence car il n’a pas de paramètre, il est applicable quelque soit la dimension, et converge vers l’optimum dans des conditions très générales (voir par ex. Spall [SPA 03]).

Implémentation

On définit d’abord un constructeur, c’est à dire une fonction qui se chargera d’initialiser l’objet optimiseur :

```
function this = rsearch()
  this = mlist(['rsearch', 'd', 'xmin', 'xmax', 'iter', 'stop',
                '_x', '_y'])
  this.stop = %f
  this._x = []
  this._y = %inf
endfunction
```

L’optimiseur retourné par cette fonction est une liste (`mlist`) de type “`rsearch`” contenant les champs `d`, la dimension de l’espace, `xmin` et `xmax`, les bornes inférieures et supérieures du domaine, `iter`, le nombre d’itérations restantes, `stop`, un booléen indiquant l’arrêt de l’optimiseur, `_x` et `_y`, le point courant (meilleur point connu) et la valeur de la fonction coût en ce point.

La méthode `ask` permet de demander un nouveau point à l'optimiseur. Ici il s'agit simplement de tirer aléatoirement un point dans le domaine :

```
function x = %rsearch_ask(this)
  x = (this.xmax - this.xmin) .* grand(1, this.d, 'def') + ...
      this.xmin
endfunction
```

La méthode `tell` permet de mettre à jour l'optimiseur avec la valeur de la fonction coût au point évalué. On conserve le point s'il a une meilleure performance que le meilleur point connu :

```
function this = %rsearch_tell(this, x, y)
  if y < this._y then
    this._x = x
    this._y = y
  end
  this.iter = this.iter - 1
  this.stop = this.stop | this.iter <= 0
endfunction
```

Noter qu'on en profite au passage pour mettre à jour les variables `iter` et `stop`.

Enfin, la méthode `best` se contente de renvoyer la valeur du meilleur point connu :

```
function [yopt, xopt] = %rsearch_best(this)
  yopt = this._y
  xopt = this._x
endfunction
```

Test

L'optimiseur ainsi programmé est testé sur la fonction de Branin-Hoo [JON 98] :

$$f(x_1, x_2) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10 \qquad (1.1)$$

Cette fonction a la particularité d'avoir trois minima locaux sur $[-5, 10] \times [0, 15]$. L'implémentation Scilab de cette fonction renvoie la valeur de la fonction ainsi que le gradient :

```
function [y, dy] = branin(x1, x2)
  a = -5.1 / (4 * %pi^2) * x1
  b = x2 + (a + 5 / %pi) * x1 - 6
  c = 10 * (1 - 1 / (8 * %pi))
  y = b^2 + c * cos(x1) + 10
  dy = [2 * b * (-2 * a + 5 / %pi) - c * sin(x1), 2 * b]
endfunction
```

Pour utiliser l'optimiseur, il faut d'abord initialiser un objet de type "`rsearch`" avec les paramètres choisis :

```
opt = rsearch()
opt.d = 2
opt.xmin = [-5, 10]
opt.xmax = [0, 15]
opt.iter = 100
```

Ensuite, la boucle d'optimisation "ask & tell" est écrite à la main :

```
while ~opt.stop
  x = ask(opt)
  y = branin(x)
  opt = tell(opt, x, y)
end
```

ce qui donne le résultat représenté sur la figure 1.5. Comme attendu, les points se répartissent uniformément sur le domaine.

Améliorations possibles

Spall [SPA 03] présente deux améliorations de la recherche aléatoire. La première amélioration ("*localized random search*") consiste à tirer les points dans un voisinage du meilleur point connu. On récupère ainsi la notion de trajectoire d'optimisation. La seconde amélioration ("*enhanced random search*") ajoute une dynamique dans la trajectoire de recherche. Ces deux améliorations sont implémentées selon la logique "ask & tell" dans la toolbox `OMD_toolbox`.

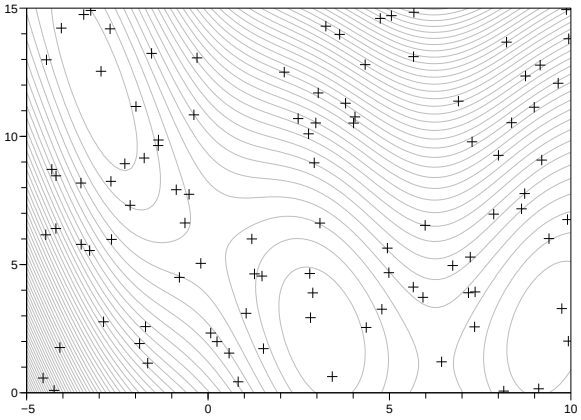


Figure 1.5. *Test de l'optimiseur* `rsearch` *(Recherche aléatoire), avec 100 itérations sur la fonction de Branin-Hoo.*

1.5.2. *Exemple 2 : stratégie évolutionnaire* $(\mu/\mu, \lambda)$

La stratégie évolutionnaire $(\mu/\mu, \lambda)$ (voir par ex. [BEY 01]) consiste à faire évoluer une densité de points d'une itération à l'autre selon le principe suivant : pour $i = 1, \ldots, n$,

1) tirer aléatoirement λ points suivant la densité de recherche p^i de loi $\mathcal{N}(\bar{x}^i, \sigma^2 \mathrm{I})$:

$$x_1^i, \ldots, x_\lambda^i \sim p^i \tag{1.2}$$

2) sélectionner les μ points ($\mu < \lambda$) donnant les meilleurs performances :

$$x_{1:\lambda}^i, \ldots, x_{\mu:\lambda}^i \tag{1.3}$$

(où la notation $x_{j:\lambda}^i$ signifie "le points ayant la j^{e} meilleure performance").

3) estimer à partir de ces points les paramètres de la densité de recherche pour l'itération suivante p^{i+1} (ici seulement la moyenne) :

$$\bar{x}^{i+1} = \frac{1}{\mu} \sum_{j=1}^{\mu} x_{j:\lambda}^i \tag{1.4}$$

Dans le jargon des stratégies évolutionnaires, les μ points sont appelés *parents*, et les λ points de la génération suivante, les *enfants*.

Implémentation

Les paramètres de l'optimiseur sont : les entiers `mu` et `lambda`, le vecteur des écart-types `sigma`, le point initial `x0`, auxquels se rajoutent les variables internes de l'optimiseur, `iter`, le nombre d'itérations restantes, `stop`, la condition d'arrêt, `_X`, la population des μ parents et `_y`, les performances correspondantes. Ces paramètres identifiés, l'implémentation du constructeur du type "`mulambda`" ne pose aucune difficulté :

```
function this = mulambda()
  this = mlist(['mulambda', 'mu', 'lambda',  'sigma', 'x0', 'iter', ...
                'stop', '_X', '_y'])
  this.stop = %f
  this._X = []
  this._y = %inf
endfunction
```

La méthode `ask` génère une population dont le centre est le point initial lors de la première itération, et le barycentre des μ parents les itérations suivantes :

```
function X = %mulambda_ask(this)
  if this._X == [] then
    xbar = this.x0
  else
    xbar = mean(this._X, 'r')
  end
  X = grand(this.lambda, 'mn', xbar', diag(this.sigma.^2))'
endfunction
```

La méthode `tell` sélectionne les μ points ayant les meilleurs performances, et met à jour les variables internes de l'optimiseur :

```
function this = %mulambda_tell(this, X, y)
  [s, k] = gsort(y, 'c', 'i')
  i = k(1 : this.mu)
  this._X = X(i, :)
  this._y = y(i)
  this.iter = this.iter - 1
  thi.stop = this.stop | this.iter <= 0
endfunction
```

La méthode `best` renvoie la valeur du meilleur point de la population courante (qui se trouve être le meilleur point des μ meilleurs...) :

```
function [yopt, xopt] = %mulambda_best(this)
  [yopt, iopt] = min(this._y)
  xopt = this._X(iopt,:)
endfunction
```

Test

L'optimiseur est testé sur la même fonction que précédemment. On initialise d'abord un nouvel objet de type "`mulambda`" avec les paramètres suivants :

```
opt = mulambda()
opt.x0 = [6, 12]
opt.mu = 2
opt.lambda = 10
opt.sigma = [1, 1]
opt.iter = 10
```

La boucle d'optimisation "ask & tell" diffère légèrement du cas précédent car il faut ici gérer une population de points à chaque itération, d'où la boucle `for` :

```
while ~opt.stop
  X = ask(opt)
  n = size(X, 1)
  y = zeros(1, n)
  for i = 1:n
    y(i) = branin(X(i, :))
  end
  opt = tell(opt, X, y)
end
```

Le déroulement de l'optimisation est représenté sur la figure 1.6. La méthode semble bien converger vers un optimum local.

Améliorations possibles

L'écart-type σ peut être adapté pour améliorer la convergence ; la règle d'usage est la règle dite du "1/5^{e}" qui vise à augmenter ou diminuer l'écart-type σ pour maintenir une probabilité d'amélioration de 20%. Dans la variante $(\mu/\mu + \lambda)$-ES, les μ parents sont conservés à la génération suivante, qui compte ainsi $\mu + \lambda$ points ; ainsi, on ne risque jamais de perdre le meilleur point. Ces deux améliorations sont implémentées selon la logique "ask & tell" dans la toolbox `OMD_toolbox`.

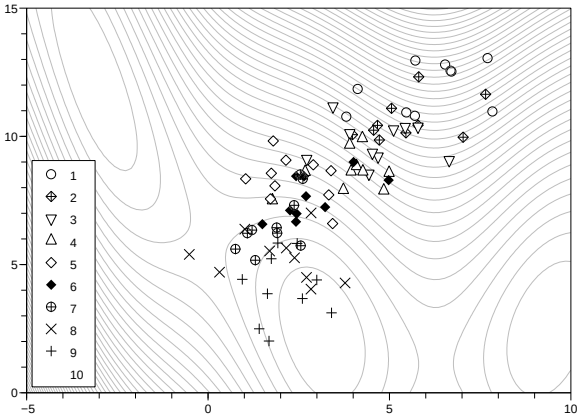


Figure 1.6. *Test de l'optimiseur* `mulambda` *(stratégie évolutionnaire $(\mu/\mu, \lambda)$), sur la fonction de Branin-Hoo, avec, $\mu = 2$ "parents" et $\lambda = 10$ "enfants" et 10 itérations (soit 100 évaluations au total).*

1.5.3. *Exemple 3 : méthode de la plus grande pente*

La méthode de la plus grande pente consiste simplement à descendre dans le sens opposé au gradient :

$$x_0 \text{ donné} \tag{1.5}$$

$$x_i = x_{i-1} - \alpha_i \nabla f(x_{i-1}) \,, \quad i \geq 1 \tag{1.6}$$

où α_i est le pas qui minimise la fonction dans cette direction :

$$\alpha_i = \underset{\alpha}{\text{argmin}}\, f(x_{i-1} - \alpha \nabla f(x_{i-1})) \tag{1.7}$$

Cette étape, appelée "*line search*", est souvent remplacée par des heuristiques plus simples, par exemple un α_i constant. La variante que nous allons implémenter consiste simplement à réduire le α_i de manière à être sûr de toujours descendre :

– on initialise α_i à la valeur initiale α_0 (donnée par l'utilisateur) ;

– on divise α_i par deux ($\alpha_i \leftarrow \alpha_i/2$) tant que

$$f(x_{i-1} - \alpha_i \nabla f(x_{i-1})) \geq f(x_{i-1})$$

La condition d'arrêt sera de type composite :

$\mathrm{dist}(x_i, x_{i-1}) \leq \varepsilon$
ou nombre maximum d'itérations atteint
ou nombre maximum d'évaluations atteint

Ce type de condition est nécessaire car, avec l'étape d'adaptation du α_i (ou même un "*line search*"), on ne connaît pas à l'avance le nombre d'évaluations de la fonction à chaque itération.

Implémentation

Le constructeur initialise un optimiseur de type "`steepdesc`" (pour *steepest descent*) :

```
function this = steepdesc()
  this = mlist(['steepdesc', 'x0', 'step', 'eps', 'iter', ...
                'eval', 'stop', '_x', '_y', '_dy', '_step'])
  this.iter = %inf
  this.eval = %inf
  this.stop = %f
  this._x = []
  this._y = %inf
endfunction
```

Les paramètres sont `x0`, le point initial, `step`, la valeur initiale du paramètre α_i au début de chaque itération (la valeur courante étant `_step`), `eps`, la tolérance sur les x_i (notée précédemment ε), ainsi que les variables internes habituelles. Noter que l'on stocke aussi le gradient (`_dy`).

La méthode `ask` renvoie le point initial `x0` lors de la première itération, et le point donné par (1.6) lors des itérations suivantes :

```
function x = %steepdesc_ask(this)
  if this._x == [] then
    x = this.x0
  else
    x = this._x - this._step * this._dy
  end
endfunction
```

La méthode `tell` doit gérer deux états de l'optimiseur, selon que le point retourné correspond à une nouvelle itération, ou à une étape de l'adaptation du α_i :

```
function this = %steepdesc_tell(this, x, y)
  if y(1) > this._y then
    this._step = this._step / 2
  else
    this._x = x
    this._y = y(1)
    this._dy = y(2)
    this._step = this.step
    this.iter = this.iter - 1
  end
  this.eval = this.eval - 1
  this.stop = this.stop | this.iter <= 0 | this.eval <= 0 | ...
              this._step * sqrt(sum(this._dy.^2)) <= this.eps
endfunction
```

Dans cette fonction, `y` est supposé être une liste dont le premier élément est la valeur de la fonction coût, et le second sont gradient. Noter aussi que le compteur d'itérations restantes `iter` n'est pas toujours décrémenté (au contraire du compteur d'évaluations restantes `eval`), car les étapes de recherche du α_i ne comptent pas dans les itérations.

La méthode `best` renvoie simplement le point courant :

```
function [yopt, xopt] = %steepdesc_best(this)
  yopt = this._y
  xopt = this._x
endfunction
```

Test

L'optimiseur ainsi programmé est testé sur le même exemple que précédemment. On initialise un optimiseur avec les paramètres suivants :

```
opt = steepdesc()
opt.x0 = [6, 10]
opt.iter = 100
opt.step = 5E-2
opt.eps = 1E-2
```

Noter que, comme on ne fixe pas de nombre maximum d'évaluations de la fonction coût (paramètre `eval`), la valeur prise par défaut est $+\infty$[1].

1. revoir l'implémentation du constructeur

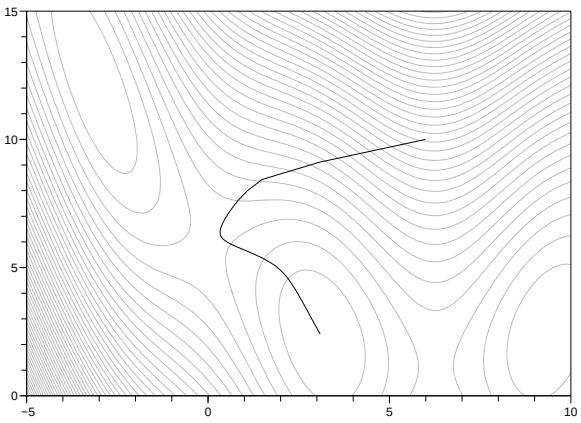


Figure 1.7. *Test de l'optimiseur* `steepdesc` *(méthode de la plus grande pente), avec 100 itérations sur la fonction de Branin-Hoo.*

La boucle d'optimisation "ask & tell" ne présente aucune surprise, si ce n'est la gestion du couple $(f(x_i), \nabla f(x_i))$ dans une liste :

```
while ~opt.stop
  x = ask(opt)
  [y, dy] = branin(x)
  opt = tell(opt, x, list(y, dy))
end
```

Le déroulement de cet algorithme est représenté sur la figure 1.7. L'optimiseur converge bien vers un minimum local.

Améliorations possibles

Pour améliorer cet optimiseur, on peut implémenter le *line search*, avec une méthode de recherche de zéro de type dichotomie par exemple. Si en plus du gradient, on connaît le Hessien de la fonction coût, la méthode de Newton-Raphson, qui consiste à utiliser $\alpha_i = \nabla^2 f(x_i)^{-1}$. On peut aussi approximer ce Hessien pour obtenir une méthode dite de quasi-Newton (par ex. la méthode BFGS)...

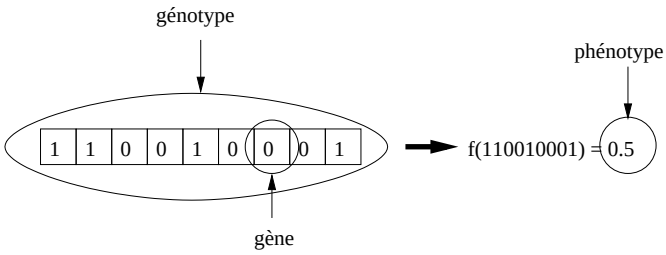


Figure 1.8. *Le vocabulaire des algorithmes génétiques.*

1.6. Un algorithme génétique

1.6.1. *Principe*

Les algorithmes génétiques sont inspirés de la génétique classique (voir [GOL 94]) : on considère une “population” de points répartis dans l’espace.

Avant d’expliquer en détail le fonctionnement d’un algorithme génétique, nous allons présenter quelques mots de vocabulaire relatifs à la génétique. Ces mots sont souvent utilisés pour décrire un algorithme génétique.

Génotype ou chromosome : c’est une autre façon de dire “individu”.

Individu : correspond au codage sous forme de gènes d’une solution potentielle à un problème d’optimisation.

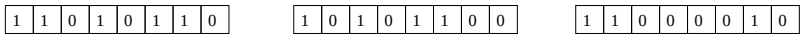
Gène : un chromosome est composé de gènes. Dans le codage binaire, un gène vaut soit 0 soit 1.

Phénotype : chaque génotype représente une solution potentielle à un problème d’optimisation. La valeur de cette solution potentielle est appelée le phénotype.

Tout ce vocabulaire est illustré à la figure 1.8.

– Chaque individu va être “codé” à la manière d’un gène. Par exemple, le plus souvent, on fait correspondre une chaîne binaire à l’individu (cette chaîne binaire est une image de la position de l’individu dans l’espace de recherche). Par exemple, trois individus (de 8 bits chacun) sont représentés à la figure 1.9.

– A chaque individu on associe une efficacité (on appelle aussi cette valeur l’ “adaptation”). Cette efficacité va correspondre à la performance d’un individu dans la résolution d’un problème donné. Par exemple, si l’on considère un problème de maximisation d’une fonction, l’efficacité de l’individu croîtra avec sa capacité à maximiser cette fonction.



(a) Individu (b) Individu (c) Individu

Figure 1.9. *Trois individus.*

$f(x)$	10	20	5
Efficacité	2	4	1
Individu	1	2	3

Tableau 1.1. *Un exemple de valeurs d'efficacité.*

Si l'on prend l'exemple de la maximisation d'une fonction $f(x)$, on a le tableau 1.1.

Comme on peut le voir sur cet exemple, l'individu 1 est moins efficace que l'individu 2. En effet, il maximise moins $f(x)$ que l'individu 2.

– Après avoir déterminé l'efficacité des individus, on opère une reproduction. On copie les individus proportionnellement à leur efficacité. Par exemple, si l'individu 1 est reproduit deux fois, l'individu 2 sera reproduit quatre fois et l'individu 3 est reproduit une fois.

Ce schéma de fonctionnement est à l'image de la vie réelle. Plus un individu est adapté à son milieu (il se défend mieux contre ses prédateurs, il mange mieux) plus sa capacité à se reproduire sera grande.

– Une fois que l'on a déterminé la fonction d'efficacité et opéré la reproduction, on effectue des "croisements" entre individus (définis plus loin). Plus un individu sera efficace dans la résolution du problème, plus celui-ci se croisera avec un grand nombre d'autres individus. Pour cela, on utilise une roue sur laquelle un individu occupe une place qui est proportionnelle à son efficacité (roulette wheel selection en anglais). Cette roue, dans le cas de l'exemple que l'on a traité ci-dessus, est représentée à la figure 1.10.

Pour notre exemple, on aura le tableau 1.2.

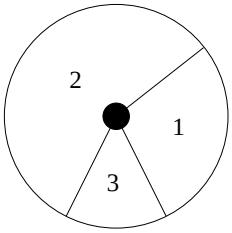


Figure 1.10. *Roue servant à sélectionner un individu pour le croisement.*

Sélection pour le croisement	
“Mâle”	“Femelle”
Individu sélectionné suivant l’efficacité	Individu sélectionné aléatoirement
1	2
1	3
2	2
2	2
2	3
2	1
3	2

Tableau 1.2. *Le croisement, première étape.*

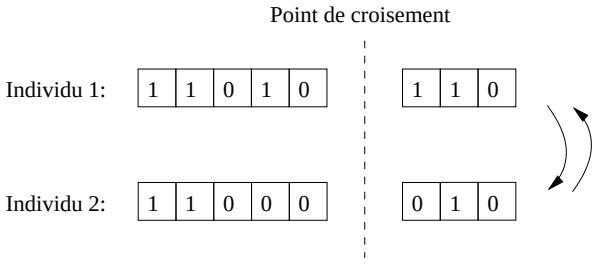


Figure 1.11. *Le croisement, deuxième étape.*

– On va maintenant fusionner l’individu de départ avec l’individu associé. Cette opération s’appelle le croisement. Pour cela, on sélectionne aléatoirement une position dans le codage de l’individu. A cette position, on sépare le codage puis on échange les morceaux de droite entre les deux individus. Un exemple de croisement est représenté à la figure 1.11.

On peut répéter cette opération pour tous les individus du tableau précédent. On obtient alors le tableau 1.3.”

Ici, on constate que l’on a obtenu deux nouveaux éléments dans la colonne résultat. Notre population s’est diversifiée. A ce moment, soit on conserve toute la population (mais on supprime les doublons), soit on supprime les éléments les moins performants, de manière à conserver le même nombre d’individus dans la population. D’autres méthodes de réduction de population peuvent être mises en œuvre ici (par exemple, on sélectionne au hasard les individus qui vont être supprimés).

– Pour finir, on peut procéder à une mutation au niveau des gènes d’un individu.

Pour cela, on commence par choisir au hasard un certain nombre d’individus (en général, la probabilité de sélection pour la mutation est faible).

“Mâle”	“Femelle”	Position de croisement	Résultat
1 1 0 1 0 1 1 0 1	1 0 1 0 1 1 0 0 2	1	1 0 1 0 1 1 0 0 1 1 0 1 0 1 1 0
1 1 0 1 0 1 1 0 1	1 1 0 0 0 0 1 0 3	3	1 1 0 0 0 0 1 0 1 1 0 1 0 1 1 0
1 0 1 0 1 1 0 0 2	1 0 1 0 1 1 0 0 2	2	1 0 1 0 1 1 0 0 1 0 1 0 1 1 0 0
1 0 1 0 1 1 0 0 2	1 0 1 0 1 1 0 0 2	5	1 0 1 0 1 1 0 0 1 0 1 0 1 1 0 0
1 0 1 0 1 1 0 0 2	1 1 0 0 0 0 1 0 3	7	1 0 1 0 1 1 0 0 1 1 0 0 0 0 1 0
1 0 1 0 1 1 0 0 3	1 1 0 1 0 1 1 0 1	4	1 1 0 1 1 1 0 0 1 0 1 0 0 1 1 0
1 1 0 0 0 0 1 0 3	1 0 1 0 1 1 0 0 2	4	1 1 0 1 1 1 0 0 1 0 1 0 0 1 1 0

Tableau 1.3. *Le croisement, résultat final.*

Ensuite, pour chaque individu sélectionné, on choisit au hasard le gène où aura lieu la mutation.

Finalement, à cette position, on change le 0 en 1 et réciproquement.

– On recommence ce procédé jusqu’à ce que l’on atteigne un critère d’arrêt (par exemple, un nombre maximal d’itérations).

Le pseudo-code d’un algorithme génétique est le suivant :

```
Initialisation de la population
Évaluation des fonctions objectif
Calcul de l'efficacité
POUR i = 1, ..., MaxIter
  Sélection aléatoire
  Sélection proportionnelle à l'efficacité
  Croisement
  Mutation
  Évaluation des fonctions objectif
  Calcul de l'efficacité
FIN POUR
```

L’avantage de cette méthode est que, dans certains cas, elle peut procurer un jeu de solutions optimales.

1.6.2. *Implémentation*

Cette version de l'algorithme génétique utilise la bibliothèque parameters qui permet de gérer une liste de paramètres à la manière des méthodes optimset et optimget de Matlab. Les fonctions que nous utilisons sont les suivantes :

– `param = init_param` : cette fonction permet d'initilialiser une structure qui va contenir la liste des paramètres ;

– `param = add_param(param,'nom_param',value)` : cette fonction ajoute le paramètre nom_param à la liste de paramètres param et lui affecte la valeur value ;

– `value = get_param(param,'nom_param',default_value)` : cette fonction permet de récupérer la valeur du paramètre nom_param de la liste param. Si ce paramètre n'existe pas, alors on affecte la valeur default_value à value.

Nous allons aussi utiliser trois fonctions spécifiques à notre algorithme génétique :

– `crossover_func_default` : cette fonction permet de croiser deux vecteurs de paramètres continus. Cette fonction est définie comme suit :

```
function [Crossed_Indiv1, Crossed_Indiv2] = ...
          crossover_func_default(Indiv1,Indiv2,param)
  if ~ isdef('param','local') then
    param = [];
  end
  Beta      = get_param(param,'beta',0);
  MinBounds = get_param(param,'minbound', ...
                        -2 * ones(size(Indiv1,1),size(Indiv1,2)));
  MaxBounds = get_param(param,'maxbound', ...
                         2 * ones(size(Indiv1,1),size(Indiv1,2)));
  mix = (1 + 2 * Beta) * rand(1,1) - Beta;
  Crossed_Indiv1 =     mix * Indiv1 + (1-mix) * Indiv2;
  Crossed_Indiv2 = (1-mix) * Indiv1 +     mix * Indiv2;
  Crossed_Indiv1 = max(min(Crossed_Indiv1, MaxBounds),MinBounds);
  Crossed_Indiv2 = max(min(Crossed_Indiv2, MaxBounds),MinBounds);
endfunction
```

– `mutation_func_default` : cette fonction permet de muter un vecteur de paramètres continus. Cette fonction est définie comme suit :

```
function Mut_Indiv = mutation_func_default(Indiv,param)
  if ~ isdef('param','local') then
    param = [];
  end
```

```
  Delta     = get_param(param,'delta',0.1);
  MinBounds = get_param(param,'minbound', ...
                        -2 * ones(size(Indiv,1),size(Indiv,2)));
  MaxBounds = get_param(param,'maxbound', ...
                         2 * ones(size(Indiv,1),size(Indiv,2)));
  Mut_Indiv = Indiv + ...
              2 * Delta * rand(size(Indiv,1),size(Indiv,2)) - ...
              Delta * ones(size(Indiv,1),size(Indiv,2));
  Mut_Indiv = max(min(Mut_Indiv, MaxBounds),MinBounds);
endfunction
```

– `selection_func_default` : Cette fonction permet de recombiner plusieurs populations d'individus (les parents et les enfants) en une population de taille pop_size. On ne retient ici que les individus les plus efficaces pour résoudre le problème. Cette fonction permet aussi de générer des fonctions objectifs vectorielles et est ainsi adaptée à l'optimisation multiobjectif au format Ask & Tell. Cette fonction est définie comme suit :

```
function [Pop_out,FObj_Pop_out,Efficiency,MO_Total_FObj_out] = ...
          selection_func_elitist(Pop_in,Indiv1,Indiv2, ...
               FObj_Pop_in,FObj_Indiv1,FObj_Indiv2, ...
               MO_Total_FObj_in,MO_FObj_Indiv1,MO_FObj_Indiv2,param)

[nargout,nargin] = argn();

mo_is_defined = ~ isempty('MO_Total_FObj_in');
if ~ isdef('param','local') then
  param = [];
end

pressure = get_param(param,'pressure',0.05);
pop_size = length(Pop_in);
Total_Pop  = lstcat(Pop_in, Indiv1, Indiv2);
Total_FObj = [FObj_Pop_in' FObj_Indiv1' FObj_Indiv2']';

// Normalisation de l'efficacité des individus
FObj_Pop_Max = max(Total_FObj);
FObj_Pop_Min = min(Total_FObj);
Efficiency = (1 - pressure) * (FObj_Pop_Max - Total_FObj) / ...
             max([FObj_Pop_Max - FObj_Pop_Min, %eps]) + pressure;
[Efficiency, Index_sort] = sort(Efficiency);
Efficiency = Efficiency(1:pop_size);
```

```
// Extraction et sélection du phénotype
Total_FObj   = Total_FObj(Index_sort);
FObj_Pop_out = Total_FObj(1:pop_size);

// Extraction et sélection du génotype
Total_Pop = list(Total_Pop(Index_sort));
Pop_out   = list(Total_Pop(1:pop_size));

// Extraction des valeurs de fonctions multiobjectifs si elles
// sont définies
if mo_is_defined then
  MO_Total_FObj_out = [MO_Total_FObj_in' MO_FObj_Indiv1' ...
                       MO_FObj_Indiv2']';
  MO_Total_FObj_out = MO_Total_FObj_out(Index_sort,:);
  MO_Total_FObj_out = MO_Total_FObj_out(1:pop_size,:);
end
Total_Pop = list(); // Réinitialisation de Total_Pop
endfunction
```

Nous allons maintenant utiliser ces fonctions pour définir un algorithme génétique pour des problèmes d'optimisation à variables continues et respectant le formalisme Ask & Tell.

Tout d'abord, le constructeur de l'algorithme génétique. Celui-ci permet de définir les valeurs par défaut des différents paramètres de l'algorithme.

Les paramètres de l'algorithme sont les suivants :

– `nb_generation` : ce paramètre correspond au nombre d'itérations ;

– `pop_size` : c'est la taille de la population de l'algorithme génétique ;

– `nb_couples` : c'est le nombre de couples que l'on forme avant chaque étape de croisement / mutation. La plupart du temps, on a pop_size = nb_couples ;

– `p_mut`, `p_cross` : ce sont les probabilités de croisement et de mutation ;

– `pressure` : c'est la pression de sélection. On donne une efficacité minimale à l'individu le moins adapté (ici 0.01) pour qu'il ait toujours la possibilité d'être sélectionné dans un couple. Cela permet de conserver une certaine diversité d'individus dans la population génération après génération.

```
function this = gaomd()
  this = mlist(['gaomd', 'nb_generation', 'pop_size', ...
                'nb_couples', 'p_mut', 'p_cross', ...
                'param', 'pressure', 'stop', ...
```

```
                'pop','fobj_pop']);
  this.nb_generation = 100;
  this.pop_size      = 100;
  this.nb_couples    = 100;
  this.p_mut         = 0.01;
  this.p_cross       = 0.7;
  this.pressure      = 0.01;
  this.stop          = %F;

  this.param = init_param();
  // Parameters to adapt to the shape
  // of the optimization problem
  this.param = add_param(this.param,'minbound',[]);
  this.param = add_param(this.param,'maxbound',[]);
  this.param = add_param(this.param,'dimension',2);
  this.param = add_param(this.param,'beta',0);
  this.param = add_param(this.param,'delta',0.1);
  this.param = add_param(this.param,'crossover_func', ...
                         crossover_func_default);
  this.param = add_param(this.param,'mutation_func', ...
                         mutation_func_default);
  this.param = add_param(this.param,'selection_func', ...
                         selection_func_elitist);
  this.param = add_param(this.param,'nb_couples', ...
                         this.nb_couples);
  this.param = add_param(this.param,'pressure', ...
                         this.pressure);
  this.pop       = list();
  this.fobj_pop  = [];
endfunction
```

Définissons maintenant les fonctions ask et tell par surcharge d'opérateur.

La fonction ask va générer une liste de couple et la retourner en paramètres afin que la valeur de fonction objectif de chaqu'un des individus soit calculée.

```
function x = %gaomd_ask(this)
  crossover_func = get_param(this.param,'crossover_func',[]);
  mutation_func  = get_param(this.param,'mutation_func',[]);
  pop_indiv1 = list();
  pop_indiv2 = list();
  fobj_pop_indiv1 = [];
```

```
fobj_pop_indiv2 = [];

// Normalisation de l'efficacité (entre pressure et 1)
FObj_Pop_Max = max(this.fobj_pop);
FObj_Pop_Min = min(this.fobj_pop);
Efficiency = (1 - this.pressure) * ...
             (FObj_Pop_Max - this.fobj_pop) / ...
              max([FObj_Pop_Max - FObj_Pop_Min, %eps]) + ...
              this.pressure;
//
// Sélection des individus
//
Wheel = cumsum(Efficiency);
for j=1:this.nb_couples
  // Sélection du premier individu du couple
  Shoot = rand(1,1)*Wheel($);
  Index = 1;
  while((Wheel(Index)<Shoot)&(Index<length(Wheel)))
    Index = Index + 1;
  end
  pop_indiv1(j)      = this.pop(Index);
  fobj_pop_indiv1(j) = this.fobj_pop(Index);
  // Sélection du second individu du couple
  Shoot = rand(1,1)*Wheel($);
  Index = 1;
  while((Wheel(Index)<Shoot)&(Index<length(Wheel)))
    Index = Index + 1;
  end
  pop_indiv2(j)      = this.pop(Index);
  fobj_pop_indiv2(j) = this.fobj_pop(Index);
end
//
// Croisement
//
for j=1:this.nb_couples
  if (this.p_cross>rand(1,1)) then
    [x1, x2] = crossover_func(pop_indiv1(j), ...
                              pop_indiv2(j), this.param);
    pop_indiv1(j) = x1;
    pop_indiv2(j) = x2;
  end
end
//
```

```
  // Mutation
  //
  for j=1:this.nb_couples
    if (this.p_mut>rand(1,1)) then
      x1 = mutation_func(pop_indiv1(j),this.param);
      pop_indiv1(j) = x1;
    end
    if (this.p_mut>rand(1,1)) then
      x2 = mutation_func(pop_indiv2(j),this.param);
      pop_indiv2(j) = x2;
    end
  end

  x = lstcat(pop_indiv1,pop_indiv2);
endfunction
```

Maintenant la fonction tell. Cette fonction va effectuer l'opération de sélection. Elle va fusionner les parents et les enfants et ne retenir que les meilleurs individus. Ensuite, on met à jour la structure interne de l'objet gaomd.

```
function this = %gaomd_tell(this, x, y)
  //
  // Recombinaison
  //
  selection_func = get_param(this.param,'selection_func',[]);
  pop_indiv1      = list();
  pop_indiv2      = list();
  fobj_pop_indiv1 = [];
  fobj_pop_indiv2 = [];

  for i=1:this.nb_couples
    fobj_pop_indiv1(i) = y(i);
    pop_indiv1(i)      = x(i);
  end
  for i=1:this.nb_couples
    fobj_pop_indiv2(i) = y(i+this.nb_couples);
    pop_indiv2(i)      = x(i+this.nb_couples);
  end

  [this.pop, this.fobj_pop] = selection_func(...
                      this.pop,pop_indiv1,pop_indiv2, ...
                      this.fobj_pop,fobj_pop_indiv1,fobj_pop_indiv2, ...
```

```
                      [],[],[],this.param);
  this.nb_generation = this.nb_generation - 1;
  this.stop = this.stop | this.nb_generation <= 0;
endfunction
```

Pour finir, définission une méthode qui permet de récupérer dans l'objet gaomd la population finale ainsi que les valeurs de fonction objectif.

```
function [yopt, xopt] = %gaomd_best(this)
  yopt = this.fobj_pop;
  xopt = this.pop;
endfunction
```

1.6.3. *Exemple*

Nous allons appliquer l'algorithme génétique au formalisme Ask & Tell à l'optimisation de la fonction Branin-Hoo (voir paragraphe 1.5.1).

Nous définissons les paramètres par défaut de l'algorithme génétique :

```
nb_generation = 10;
pop_size      = 100;
p_mut         = 0.1;
p_cross       = 0.7;

gaopt               = gaomd();
gaopt.nb_generation = nb_generation;
gaopt.pop_size      = pop_size;
gaopt.nb_couples    = pop_size;
gaopt.p_mut         = p_mut;
gaopt.p_cross       = p_cross;

// On règle le domaine de validité du problème
Min = [-5, 0]';
Max = [15, 10]';
gaopt.param = add_param(gaopt.param,'minbound',Min);
gaopt.param = add_param(gaopt.param,'maxbound',Max);
```

On initialise aléatoirement la population de l'algorithme génétique :

```
for i=1:gaopt.pop_size
  gaopt.pop(i) = (Max - Min) .* ...
            rand(size(Max,1),size(Max,2)) + Min;
  gaopt.fobj_pop(i) = branin(gaopt.pop(i));
end
pop_init      = gaopt.pop;
fobj_pop_init = gaopt.fobj_pop;
y_min = [];
```

On lance l'algorithme génétique :

```
while ~ gaopt.stop
  printf('gaopt running: iteration %d / %d - ', ...
          nb_generation - gaopt.nb_generation + 1, ...
          nb_generation);
  x = ask(gaopt);
  y = [];
  for i=1:length(x)
    y(i) = branin(x(i));
  end
  y_min($+1) = min(y);
  printf(' fmin = %f\n',y_min($));

  gaopt = tell(gaopt, x, y);
end
```

On récupère la population finale et les valeurs de fonction objectif :

```
[fobj_pop, pop] = best(gaopt);
```

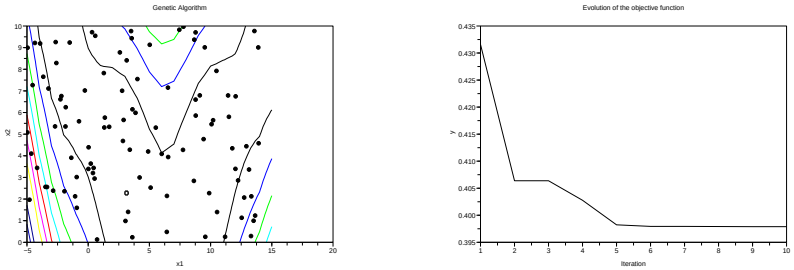
1.7. La méthode du simplex

1.7.1. *Principe*

Cette méthode n'a rien à voir avec la méthode du simplexe utilisée en programmation linéaire. Il s'agit là d'une méthode de recherche séquentielle [NEL 65] de l'optimum d'un problème d'optimisation. Le logiciel Multisimplex réalise cette optimisation pour un problème d'optimisation multiobjectif de manière interactive.

Cette méthode utilise $k+1$ essais (où k représente la dimension de la variable de décision $\overrightarrow{x}$) pour définir une direction d'amélioration des fonctions objectif.

L'amélioration des fonctions objectif est obtenue en utilisant une méthode d'agrégation floue.



(a) La population initiale (points noirs) et la population finale (points blancs).

(b) L'évolution de la meilleure valeur de fonction objectif en fonction des itérations.

Figure 1.12. *Les résultats obtenus par l'algorithme génétique implémenté sous le formalisme Ask & Tell.*

1.7.2. *Présentation de la méthode*

On commence par choisir au hasard $k + 1$ valeurs pour la variable de décision $\overrightarrow{x}$. L'algorithme évalue alors les $k + 1$ points et supprime le point le moins "efficace". Il crée alors un nouveau point à partir du point supprimé (voir figure 1.13) et recommence l'évaluation.

L'algorithme comporte quelques règles, qui permettent de choisir des points qui évitent de tourner autour d'une mauvaise solution. Les deux principales règles sont les suivantes :

Règle 1 : rejeter les pires solutions.

Une nouvelle position de la variable de décision $\overrightarrow{x}$ est calculée par réflexion de la position rejetée (voir la figure 1.13). Après cette transformation, on recherche le nouveau pire point. La méthode est alors répétée en éliminant ce point, etc. A chaque étape, on se rapproche de la zone où se trouve l'optimum recherché.

Règle 2 : ne jamais revenir sur un point qui vient juste d'être rejeté.

Sans cette règle, l'algorithme pourrait osciller entre deux "mauvais" points.

Sur la figure 1.14, on peut voir le cheminement de la méthode dans l'espace des variables de décision, dans le cas d'un exemple à deux variables de décision. Sur cette figure sont représentées des courbes de niveau représentant la "direction" d'amélioration de la fonction objectif à optimiser. Dans cet exemple, l'optimisation se déroule de manière à maximiser le pourcentage correspondant à la courbe de niveau.

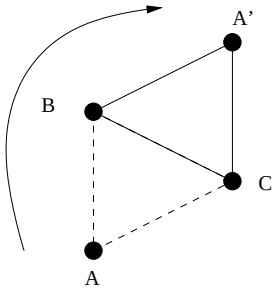


Figure 1.13. *Choix d'un nouveau point par symétrie.*

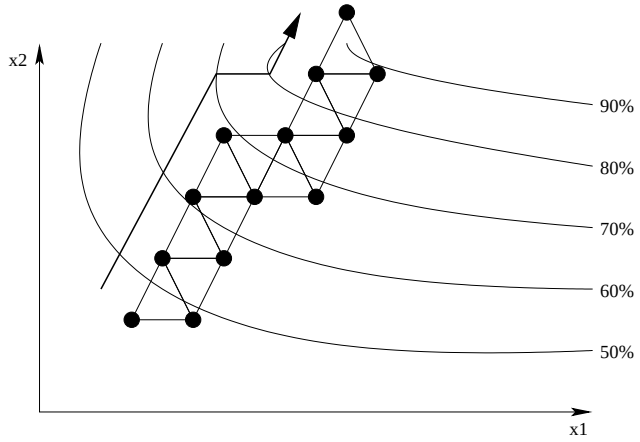


Figure 1.14. *Cheminement de la méthode du simplex.*

Présentons maintenant l'algorithme complet de la méthode du simplex. Nous utiliserons les notations suivantes :

– W : point le moins favorable ou point venant d'être rejeté.

– B : point le plus favorable.

– N : second point le plus favorable.

L'algorithme du simplex est présenté à la figure 1.15.

Il existe aussi une version modifiée de la méthode du simplex. Cette méthode, dite du simplex modifié, comporte deux règles supplémentaires :

Règle 3 : dilater le déplacement dans le sens de l'amélioration des fonctions objectif.

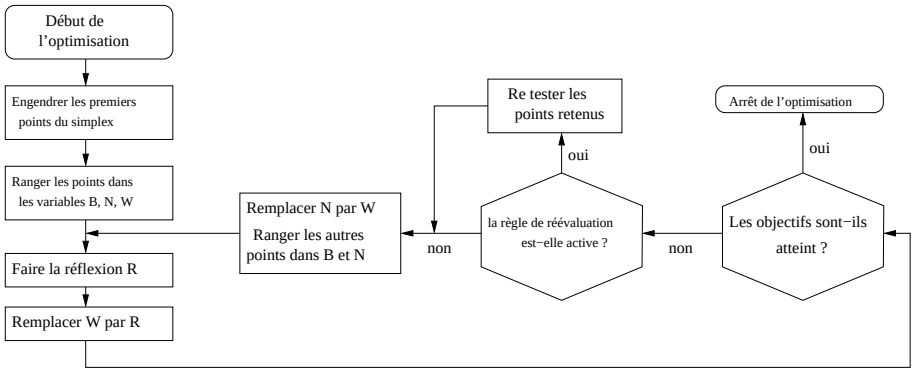


Figure 1.15. *L'algorithme de la méthode du simplex.*

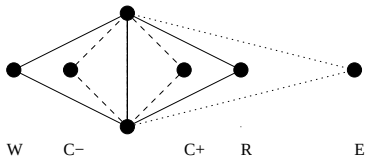


Figure 1.16. *Le processus de dilatation et de contraction de la méthode du simplex modifié.*

Règle 4 : contracter le déplacement, si celui-ci a été fait dans une direction qui n'est pas favorable à l'amélioration des fonctions objectif.

Sur un problème à deux variables de décision, on obtient le schéma situé à la figure 1.16.

En reprenant les mêmes notations que pour l'algorithme précédent, nous obtenons la représentation de la figure 1.17.

Pour les différentes transformations, on a les expressions suivantes :

– Réflexion : $R = \bar{C} + \alpha \cdot \left(\bar{C} - W\right)$

– Dilatation : $E = \bar{C} + \gamma \cdot \left(\bar{C} - W\right)$

– Contraction positive : $C_+ = \bar{C} + \beta^+ \cdot \left(\bar{C} - W\right)$

– Contraction négative : $C_- = \bar{C} - \beta^- \cdot \left(\bar{C} - W\right)$

où

– W est le point rejeté,

– $\bar{C}$ est le barycentre des points restants,

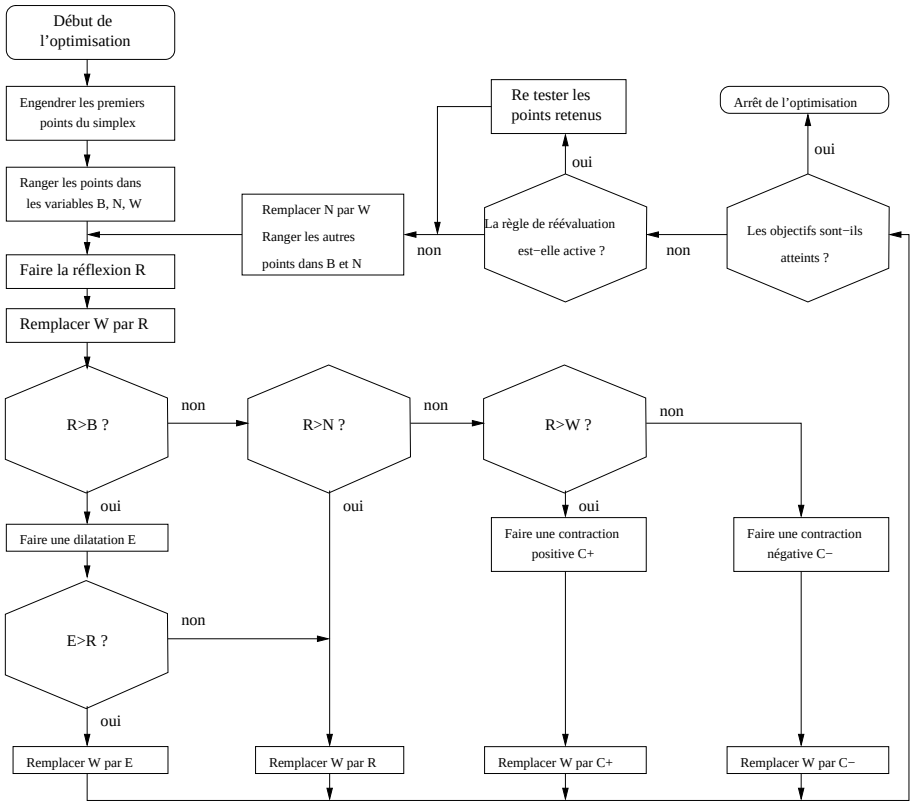


Figure 1.17. *L'algorithme de la méthode du simplex modifié.*

– α est le coefficient de réflexion ($\alpha = 1$ par défaut),
– γ est le coefficient de dilatation ($\gamma = 2$ par défaut),
– β^+ est le coefficient de contraction positive ($\beta^+ = 0.5$ par défaut),
– β^- est le coefficient de contraction négative ($\beta^- = 0.5$ par défaut).

1.7.3. *Implémentation*

Notre implémentation Ask & Tell de l'algorithme de Nelder et Mead utilise une version "step by step" de l'algorithme du simplex. Cette méthode scilab a le prototype suivant :

```
function [x_next, data_next, eval_Func, f_hist, x_hist] = ...
            step_nelder_mead(f_current, x_current, data_current, ...
```

```
nm_mode, Log, kelley_restart, kelley_alpha)
```

Les différents paramètres de cette méthode d'optimisation on la signification suivante :

– `f_current` : la valeur de fonction objectif correspondant à x_current. Si nm_mode='init', alors f_current doit être un vecteur de $n+1$ valeurs correspondant à chacune des coordonnées du simplex ;

– `x_current` : le simplex initial ($n+1$ vecteurs colonne) ou la solution à calculer (1 vecteur colonne) - dépend de la valeur de nm_mode ;

– `data_current` : l'état de la méthode d'optimisation. Ce paramètre doit être vide lors de la première itération de la méthode d'optimisation (nm_mode='init') ;

– `nm_mode` : le mode opératoire de la méthode de Nelder & Mead "step by step". Les valeurs admissibles sont les suivantes :
  - `'init'` : pour l'itération initiale ;
  - `'run'` : pour toutes les autres itérations ;
  - `'exit'` : pour récupérer la meilleure solution.

– `Log` : un booléen. S'il vaut %T alors des informations sont affichées durant le déroulement de l'optimisation (c'est un paramètre optionel qui est à %F par défaut) ;

– `kelley_restart` : un booléen. S'il vaut %T alors on autorise le simplex à être recalculé quand un seuil relatif à la forme du simplex courant a été atteint (c'est un paramètre optionel qui est à %F par défaut) ;

– `kelley_alpha` : un seuil relatif à la forme du simplex courant. Ce seuil permet de limiter la dégénérescence du simplex au cours de l'optimisation (c'est un paramètre optionel qui vaut 1e-4 par défaut) ;

– `x_next` : un vecteur de paramètres pour lequel on doit calculer la valeur de fonction objectif ou alors la meilleure solution trouvé par la méthode de Nelder & Mead (si nm_mode vaut 'init' x_next correspond à $n+1$ vecteurs colonne) ;

– `data_next` : structure contenant l'état de la méthode de Nelder & Mead. Cette structure doit être transmise à la fonction itération après itération ;

– `eval_Func` : le nombre d'évaluation de la fonction objectif (ce paramètre est optionel) ;

– `f_hist` : la meilleure valeur de fonction objectif pour chaque itération (ce paramètre est optionel) ;

– `x_hist` : le vecteur de paramètre correspondant au simplex obtenu à chaque itération ($n+1$ vecteur colonne) (ce paramètre est optionel).

Nous allons maintenant utiliser cette fonction pour définir un algorithme de Nelder & Mead respectant le formalisme Ask & Tell.

Les paramètres importants de notre algorithme de Nelder & Mead sont les suivant :

– `ItMX` : le nombre maximum d'itérations ;

– `x0` : le point initial de départ de notre méthode d'optimisation (un simplex initial sera construit autour de ce point) ;

– `upper` et `lower` : les bornes minimales et maximales de notre domaine d'optimisation ;

– `simplex_relsize` : la taille relative du simplex qui sera construit autour du point x0 ;

Le constructeur de notre méthode d'optimisation est le suivant :

```
function this = nmomd()
  this = mlist(['nmomd', 'ItMX', 'x0', 'x_init', ...
                'f_init', 'upper','lower', ...
                'kelley_restart', 'kelley_alpha', ...
                'simplex_relsize', 'log', 'stop', ...
                'data_next', 'init']);

  this.ItMX = 100;
  this.x0   = [];
  this.kelley_restart  = %F;
  this.kelley_alpha    = 1e-4;
  this.simplex_relsize = 0.1;
  this.log             = %F;
  this.stop            = %F;
  this.upper           = 1e6*ones(size(x0,1),size(x0,2));
  this.lower           = - 1e6*ones(size(x0,1),size(x0,2));
  this.data_next       = [];
  this.init            = %T;
  this.x_init          = [];
  this.f_init          = [];
endfunction
```

Définissons maintenant les fonctions ask et tell par surcharge d'opérateur.

La fonction ask va générer un simplex initial si on est à l'itération initiale. Sinon, la fonction ask retourne la valeur x_init qui correspond au vecteur de paramètres pour lequel on souhaite calculer la valeur de la fonction objectif.

```
function x = %nmomd_ask(this)
  if this.init then
```

```
    // We set the initial simplex
    for i=1:length(this.x0)+1
      this.x_init(:,i) = this.x0 + ...
             this.simplex_relsize*0.5* ...
             ((this.upper - this.lower) .* ...
               rand(size(this.x0,1),size(this.x0,2)) ...
               + this.lower);
    end
  end
  x = this.x_init;
endfunction
```

Maintenant la fonction tell. Cette fonction va mettre à jour le vecteur de paramètres x_init en fonction de la valeur de la fonction objectif qui a été calculée. Ensuite, on met à jour la structure interne de l'objet nmomd (on décrémente le nombre d'itérations et on met à jour le booléen indiquant l'arrêt de la méthode (stop).

```
function this = %nmomd_tell(this, x, y)
  if this.init then
    [this.x_init, this.data_next] = ...
                 step_nelder_mead(y, x, [], 'init', ...
                         this.log, this.kelley_restart, ...
                         this.kelley_alpha);
    this.init = %F;
  else
    [this.x_init, this.data_next] = ...
                 step_nelder_mead(y, x, this.data_next, 'run', ...
                         this.log, this.kelley_restart, ...
                         this.kelley_alpha);
  end
  this.ItMX = this.ItMX - 1;
  this.stop = this.stop | (this.ItMX <= 0);
endfunction
```

Pour finir, définission une méthode qui permet de récupérer dans l'objet nmomd le meilleur vecteur de paramètres obtenu ainsi que la valeure de fonction objectif correspondante.

```
function [yopt, xopt] = %nmomd_best(this)
  [xopt, yopt] = step_nelder_mead(this.f_init, this.x_init, ...
```

```
                                    this.data_next, 'exit', ...
                                    this.log, this.kelley_restart, ...
                                    this.kelley_alpha);
endfunction
```

1.7.4. *Exemple*

Nous allons appliquer la méthode de Nelder & Mead au formalisme Ask & Tell à l'optimisation de la fonction Branin-Hoo (voir paragraphe 1.5.1). On commence par initialiser les paramètres de la méthode de Nelder & Mead :

```
ItMX = 100;
nmopt      = nmomd();
nmopt.ItMX = ItMX;
nmopt.kelley_restart  = %F;
nmopt.kelley_alpha    = 1e-4;
nmopt.simplex_relsize = 0.1;
nmopt.log             = %F;
```

On définit le domaine de validité de notre problème d'optimisation :

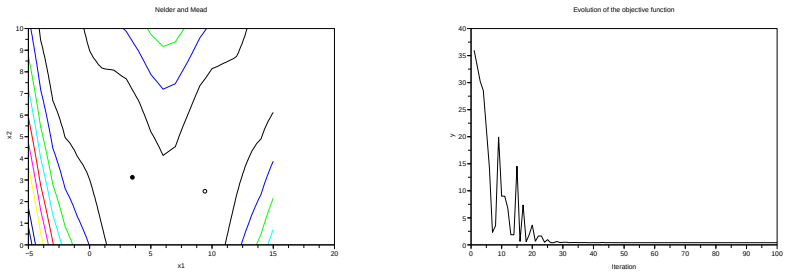
```
nmopt.lower = [-5, 0]';
nmopt.upper = [15, 10]';
```

On définit on point initial aléatoirement dans notre domaine d'optimisation :

```
nmopt.x0   = (Max - Min).*rand(size(Max,1),size(Max,2)) + Min;
```

On lance la méthode d'optimisation :

```
y_min = [];
while ~ nmopt.stop
  printf('nmopt running: iteration %d / %d - ', ...
         ItMX - nmopt.ItMX + 1, ItMX);
  x = ask(nmopt);
  y = [];
  for i=1:size(x,2)
```



(a) Le point initial (point noir) et le point final (point blanc).

(b) L'évolution de la meilleure valeur de fonction objectif en fonction des itérations.

Figure 1.18. *Les résultats obtenus par l'algorithme de Nelder & Mead implémenté sous le formalisme Ask & Tell.*

```
    y(i) = branin(x(:,i));
  end
  y_min($+1) = min(y);
  printf(' fmin = %f\n', y_min($));

  nmopt = tell(nmopt, x, y);
end
```

On récupère le meilleur vecteur de paramètres ainsi que la valeure de la fonction objectif correspondante :

```
[f_opt, x_opt] = best(nmopt);
```

1.8. Stratégie évolutionnaire CMA-ES

La stratégie évolutionnaire CMA-ES (*Covariance Matrix Adaptation Evolution Strategy*) [HAN 01, HAN 03, HAN 04, HAN 06] est une méthode stochastique d'optimisation reprenant les bases de la stratégie $(\mu/\mu, \lambda)$ présentée au paragraphe 1.5.2.

1.8.1. *Principe*

À chaque itération, les points candidats sont tirés au hasard suivant une *loi gaussienne multivariée* : autour d'une valeur moyenne $\vec{m}$, les points candidats se répartissent selon une distribution gaussienne, centrée, de matrice de covariance $\underline{\underline{C}}$ qui est

en outre réduite par le facteur σ, la longueur de pas,

$$\vec{x}_i = \vec{m} + \sigma \times \mathcal{N}_i\left(\vec{0}, \underline{\underline{C}}\right) \qquad \text{for } i = 1, \ldots, \lambda \ . \tag{1.8}$$

Les λ nouveaux points définis par (1.8) sont des réalisations d'un vecteur gaussien de moyenne $\vec{m}$ et de matrice de covariance $\sigma^2\underline{\underline{C}}$; ils sont distribués selon la loi $\mathcal{N}\left(\vec{m}, \sigma^2\underline{\underline{C}}\right)$. Une population de taille $\lambda = 10$ est généralement suffisante, bien que *parfois*, une valeur plus grande ($\lambda \gg 100$) soit préférable. Outre le point initial et le critère d'arrêt de l'algorithme, la taille de la population λ (> 3) est le principal paramètre qui peut être (librement) fixé par l'utilisateur.

Dans la méthode $(\mu/\mu, \lambda)$-CMA-ES, les μ meilleurs points ($\mu < \lambda$) de l'itération courante sont sélectionnés. Ils permettent de calculer le point moyen $\vec{m}$ et mettre à jour les autres paramètres de la distribution, σ et $\underline{\underline{C}}$, pour l'itération suivante. Deux grands principes sont mis en œuvre pour effectuer ces mises à jour :

– L'évolution temporelle du point moyen $\vec{m}$ dans l'espace de recherche est "analysée". Le pas σ est adapté selon la distance parcourue. La trajectoire modèle la matrice de covariance. La vitesse de changement de $\vec{m}$ est décrite par une équation d'état appelée "chemin d'évolution".

– La forme de la distribution, c'est-à-dire la matrice de covariance $\underline{\underline{C}}$, est mise à jour dans le but d'augmenter la vraissemblance d'avoir une progression de la distribution efficace. Toutes les estimations sont sans biais, compte tenu de la sélection aléatoire.

L'algorithme $(\mu/\mu, \lambda)$-CMA-ES se révèle être une méthode d'optimisation stochastique robuste et efficace.

– Une population de $\lambda \gg 1$ points permet d'une part, une sélection $(\mu/\mu, \lambda)$ robuste au bruit, et d'autre part, évite de se faire piéger dans un minimum local. Augmenter la taille de la population λ peut permettre d'améliorer encore les performances dans la recherche de l'optimum global [HAN 04].

– L'adaptation de la matrice de covariance est particulièrement utile pour des problèmes mal conditionnés, les performances sur les problèmes bien conditionnés n'étant pas affectées. Comme inconvénient, sa complexité algorithmique est en $O(n^2)$ (celle de l'échantillonnage dans une loi gaussienne multivariée), où n est la dimension de l'espace de recherche.

– Le contrôle du pas σ favorise les échantillons avec une grande variance, ce qui est essentiel pour éviter une convergence prématurée. Selon le cas, le contrôle du pas permet une *augmentation* rapide de la variance de la distribution de recherche, ainsi qu'une diminution rapide pour la convergence vers un optimum.

1.8.2. *Implémentation et interfaces*

L'implémentation suit de très près les descriptions de [HAN 04, HAN 06, HAN 09]. Outre les fonctions `%cma_ask`, `%cma_tell` et `%cma_best` bien connues (elles ont toutes une homologue sans le % de tête), nous présentons rapidement les fonctionnalités les plus importantes.

```
param = cma_new([])
[es, param] = cma_new(param)
```

Dans le premier cas, `cma_new` retourne simplement ses paramètres. Dans le second, `cma_new` retourne un nouvel optimiseur, nommé `es`. L'utilisateur doit définir au moins deux paramètres obligatoires dans la structure `param` (voir plus bas). Une structure complète avec tous les paramètres utilisés est retournée.

```
cma_plot(fignb, name_prefix, name_extension,
         object_variables_name, plotrange)
```

Cette commande trace des graphiques permettant de suivre le déroulement de l'optimisation. La visualisation des résultats peut s'avérer très instructive et, dans certains cas, aider à la formulation du problème (paramétrisation, ...).

```
[xopt, f, out, param] = cma_optim(costf, x0, sigma0, param)
[xopt, f, out, param] = cma_optim_restarted(costf, x0, sigma0,
                                            restarts, param)
```

Ces fonctions fournissent une interface similaire à la fonction `optim` de Scilab. Avec `cma_optim_restarted`, la méthode est relancée plusieurs fois, λ étant multiplié par deux à chaque relance [AUG 05].

1.8.3. *Exemples*

Un exemple complet d'utilisation est fourni avec le script `demos/runcma.sce`. Ici, nous présentons quelques exemples en partant de zéro. Un objet de type `cma` est créé avec la fonction `cma_new(param)`. La structure `param` a deux champs obligatoires, `x0` et `sigma0`. Tout d'abord, nous récupérons la structure contenant les valeurs par défaut :

```
-->p = cma_new();
cma_new has two mandatory fields in its input parameter struct:
   x0 (or typical_x) and sigma0.
A complete parameter struct has been returned.

-->disp(p)

   x0: [0x0 constant]
   typical_x: [0x0 constant]
   sigma0: [0x0 constant]
   opt: [1x1 struct]
   stop: [1x1 struct]
   verb: [1x1 struct]
   readme: [1x1 struct]
```

La structure `p` des paramètres contient aussi quelques commentaires dans le champ `readme`, où l'on peut voir la signification des paramètres `x0` et `sigma0`.

```
-->disp(p.readme)

   x0: "initial solution, either x0 or typical_x MUST be provided"
   typical_x: "typical solution, the genotypic zero for more conveni...
   sigma0: "initial coordinate-wise standard deviation MUST be provided"
   stop: "termination options, see .stop.readme"
   opt: "more optional parameters, see .opt.readme"
   verb: "verbosity parameters, see .verb.readme"
```

Les paramètres `x0` `sigma0` n'ont pas de valeur par défaut. Tous les autres paramètres sont optionnels : des valeurs par défaut sont affectées à ceux qui ne sont pas précisés dans la structure `p` en entrée de `cma_new`.

Nous continuons avec un exemple bien connu, la minimisation de la fonction de Rosenbrock (dimension $n = 8$). Nous initialisons d'abord l'optimiseur :

```
clear param
param.x0 = zeros(8,1);
param.sigma0 = 0.001;  // far too small for testing puspose
es = cma_new(param);
```

Le paramètre `sigma0` a délibérément été choisi trop petit. Il devrait être choisi de manière à ce que l'optimum soit dans la région `x0` $\pm$ `3*sigma0`, dans notre cas `sigma0 = 0.5` serait approprié. Dans le cas où des valeurs différentes sont appropriées dans

différentes directions, on peut utiliser le paramètre optionnel `opt.scaling_of_variables`.

Nous définissons la fonction à minimiser, la fonction de Rosenbrock[2] :

```
function f = frosen(x)
  f = -1e-5 + 1e2*sum((x(1:$-1).^2 - x(2:$)).^2) ...
      + sum((x(1:$-1)-1).^2);
endfunction
```

La boucle d'optimisation suivante est similaire aux exemples rencontrés tout au long de ce chapitre.

```
while ~es.stop
  X = ask(es);
  y = [];
  for i = 1:length(X)
    y(i) = frosen(X(i));
  end
  es = tell(es, X, y);
end
[yopt, xopt] = best(es);
```

Ici, la fonction `ask` retourne une *liste* de points solution (depuis la version 0.99), ces points étant des *vecteurs colonnes*. Cet exemple produit les sorties suivantes :

```
(5/5_W,10)-CMA-ES (W=[46,27,16,...]%, mueff=3.2) in 8-D
Iter, Evals: Function Value   (worst) |Axis Ratio |idx:Min SD, idx:Max SD
   1,    10: +6.9926606e+000 +(2e-002)| 1.05e+000 |  2:8.80e-004, 8:8.96e-004
   2,    20: +6.9908033e+000 +(1e-002)| 1.15e+000 |  2:8.93e-004, 8:9.62e-004
   3,    30: +6.9840590e+000 +(1e-002)| 1.28e+000 |  2:1.01e-003, 8:1.11e-003
 101,  1010: +4.5686563e+000 +(6e-001)| 6.63e+000 |  8:1.15e-002, 2:3.62e-002
 201,  2010: +1.4533785e+000 +(8e-001)| 1.09e+001 |  8:1.23e-002, 5:3.71e-002
 301,  3010: +1.4976704e-001 +(3e-002)| 1.63e+001 |  1:3.11e-003, 8:1.69e-002
 401,  4010: +1.0262306e-005 +(5e-005)| 4.81e+001 |  2:8.23e-005, 8:1.50e-003
 501,  5010: -9.9999987e-006 +(4e-012)| 5.95e+001 |  1:2.74e-008, 8:6.83e-007
 537,  5370: -1.0000000e-005 +(1e-014)| 6.22e+001 |  1:1.24e-009, 8:3.18e-008
```

2. Voir aussi `demos/fitfuns.sci`, faire `getf('path_to_toolbox/demos/fitfuns.sci')`. Pour l'exemple, nous avons soustrait 10^{-5} à la fonction originale pour que les valeurs proches de l'optimum soient négatives. La méthode CMA-ES est invariante à l'ajout d'une constante à la valeur de la fonction, et se révèle avoir un grand nombre de propriétés d'invariance plus importantes, voir [HAN 01, HAN 06].

La colonne “Axis Ratio” (5e colonne) donne la racine carrée du conditionnement de la matrice de covariance $\underline{\underline{C}}$. Des grandes valeurs indiquent un problème mal conditionné. Avec un conditionnement final de $60^2 \approx 4 \times 10^3$, la fonction de Rosenbrock se révèle être un problème modérément mal conditionné. Des conditionnements allant jusqu'à 10^7 ne sont pas rares en pratique. La précision numérique utilisée permet de manipuler des conditionnements jusqu'à 10^{14}.

D'autres sorties utiles sont fournis par la structure `es.out`.

```
-->disp(es.out)

   seed: 1.151D+09
   version: 0.99
   genopheno: [3 genopheno]
   dimension: 8
   stopflags: [2 list]
   solutions: [1x1 struct]
   evals: 5370
   iterations: 537
```

Le champ `es.out.solutions.mean.x` donne le point moyen $\vec{m}$ final. On peut voir que l'algorithme s'est arrêté à cause de faibles variations de la fonction :

```
-->for s = es.out.stopflags, printf(s + '\n'); end
tolfun
tolfunhist
```

La figure 1.19 montre la sortie graphique qui a été générée lors de l'exécution de l'exemple. Le tracé peut aussi être généré en appelant

```
-->cma_plot
```

La fonction `cma_plot` lit les données dans les fichiers générés par `tell()`. Dans le quadrant inférieur gauche de la figure 1.19, nous pouvons voir que le mauvais conditionnement du problème se traduit par un axe principal beaucoup plus important que les autres (i.e. une valeur propre clairement plus grande). Cela signifie qu'il n'y a qu'une direction dans l'espace de recherche qui permet d'améliorer notablement la valeur de la fonction. Dans le quadrant supérieur droit, nous pouvons observer l'évolution des coordonnées qui révèle une structure tout à fait particulière de la fonction de Rosenbrock.

Si l'on ne veut pas afficher de graphique ni/ou écrire de fichiers, le champ `verb` de la structure de paramètres `p` permet de régler les options de verbosité, respectivement `plotmodulo` et `logmodulo`.

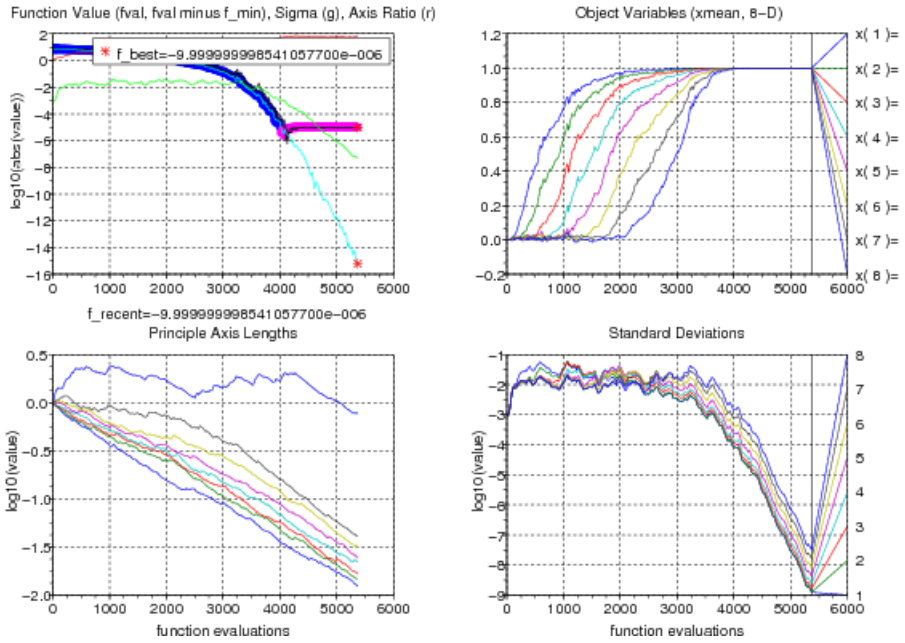


Figure 1.19. *Tracé de* `cma_plot` *après une optimisation de la fonction de Rosenbrock* -10^{-5} *en dimension 8.* ***En haut à gauche*** *: La valeur de la fonction (ligne épaisse bleue et rose) devient négative après environ* 4000 *évaluations de la fonction (*400 *itérations). La ligne bleu ciel (valeur finale* 10^{-15}*) représente l'écart à la meilleure valeur obtenue et montre une amélioration continue, puisque la courbe décroît jusqu'à la fin.* ***En bas à gauche*** *: Les longueurs des axes principaux de la distribution d'échantillonnage (en échelle log) s'adaptent à la fonction à minimiser. Elles révèlent la structure (locale) du problème d'optimisation sous-jacent.* ***En haut à droite*** *: Les coordonnées du point* $\vec{m}$ *se déplacent de zéro à un dans l'ordre des coordonnées. L'optimum global est à* $x_i = 1$ *pour tous les* $i = 1, \ldots, 8$*.* ***En bas à droite*** *: Les écart-types augmentent rapidement dans les premières itérations, d'un ordre de grandeur de* 10^{-3} *à* 10^{-2}*. Ils varient ensuite au cours de l'optimisation par un facteur trois, et décroissent rapidement dans la phase finale de convergence après* 4000 *évaluations de la fonction.*

```
-->p = cma_new([]);

-->disp(p.verb)

   logmodulo: 1
   displaymodulo: 100
   plotmodulo: "max(logmodulo, 500)"
   logfunvals: [0x0 constant]
   readfromfile: "signals.par"
   filenameprefix: "outcmaes"
```

```
   append: 0
   readme: [1x1 struct]

-->p.verb.plotmodulo = 0  // do not plot

-->p.verb.logmodulo = 10  // write only every 10-th iteration
```

Pour avoir une exécution silencieuse, `logmodulo` et `displaymodulo` peuvent être mis à zéro, réduisant les sorties au minimum.

1.9. Conclusion

Nous avons présenté dans ce chapitre une logique de programmation orientée objet des optimiseurs, appelée logique “ask & tell”, d’après les noms des deux fonctions autour desquelles s’articulent les stratégies d’optimisation. Cette logique rend la programmation des optimiseurs plus délicate, comme on a pu le voir à travers les exemples du paragraphe 1.5. Cependant, l’investissement demandé au programmeur se retrouve chez l’utilisateur. En effet, comme illustré avec la stratégie multistart au paragraphe 1.4, ce formalisme permet d’implémenter très simplement des *stratégies d’optimisation* avancées. De telles stratégies sont indispensables dans le contexte de la conception à partir de simulations numériques. Il faut par exemple gérer au mieux le budget de simulations, envisager le fait que des simulations peuvent “planter”, etc. Ceci est parfaitement possible avec l’approche fonctionnelle traditionnelle (i.e. celle de la fonction `optim` de Scilab), mais il faut alors écrire des fonctions coût compliquées et abuser des variables globales. Au final, cela donne un code obscur et difficile à maintenir.

Les réalisations présentées dans ce chapitre (méthode du simplex, algorithme génétique, méthode CMA-ES) fournissent déjà quelques outils selon la logique “ask & tell”. Ces optimiseur sont connus pour être performants, et la plupart sont adaptés aux problèmes en grande dimension.

1.10. Bibliographie

[AUG 05] AUGER A., HANSEN N., « A Restart CMA Evolution Strategy With Increasing Population Size », *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2005)*, IEEE Press, p. 1769-1776, 2005.

[BEY 01] BEYER H.-G., *The Theory of Evolution Strategies*, Springer, 2001.

[CAM 06] CAMPBELL S. L., CHANCELIER J.-P., NIKOUKHAH R., *Modeling and Simulation in Scilab/Scicos*, Springer, 2006.

[GOL 94] GOLDBERG D., *Algorithmes génétiques*, Addison Wesley, 1994.

[HAN 01] HANSEN N., OSTERMEIER A., « Completely derandomized self-adaptation in evolution strategies », *Evolutionary Computation*, vol. 9, n°2, p. 159–195, MIT Press, 2001.

[HAN 03] HANSEN N., MÜLLER S. D., KOUMOUTSAKOS P., « Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES) », *Evolutionary Computation*, vol. 11, n°1, p. 1-18, 2003.

[HAN 04] HANSEN N., KERN S., « Evaluating the CMA evolution strategy on multimodal test functions », *Parallel Problem Solving from Nature-PPSN VIII*, vol. 3242, p. 282–291, Springer, 2004.

[HAN 06] HANSEN N., « The CMA evolution strategy : a comparing review », LOZANO J., LARRAÑAGA P., INZA I., BENGOETXEA E., Eds., *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, p. 75–102, Springer, 2006.

[HAN 09] HANSEN N., NIEDERBERGER S. P. N., GUZZELLA L., KOUMOUTSAKOS P., « A Method for Handling Uncertainty in Evolutionary Optimization with an Application to Feedback Control of Combustion », *IEEE Transactions on Evolutionary Computation*, 2009, to appear.

[JON 98] JONES D. R., SCHONLAU M., WELCH W. J., « Efficient global optimization of expensive black-box functions », *Journal of Global Optimization*, vol. 13, p. 455–492, 1998.

[NEL 65] NELDER J. A., MEAD R., « A Simplex method for function minimization », *Computer Journal*, vol. 7, p. 308-313, 1965.

[SPA 03] SPALL J. C., *Introduction to stochastic search and optimization*, Wiley, 2003.