# Description de l'architecture Scilab pour le projet RNTL/OMD*

Gilles Pujol† Rodolphe Le Riche‡

17 mars 2008

Lot 8 Intégration logicielle
Lot 8.1 Spécifications pour les développements logiciels

**Résumé**

Les algorithmes d'optimisation classiques sont implémentés selon un *paradigme fonctionnel* qui place la méthode d'optimisation au sommet de la hiérarchie. Par exemple, avec Scilab on écrira la commande `[f,xopt]=optim(costf,x0)`. Cette instruction se chargera de tout le processus d'optimisation, sans contrôle possible de l'utilisateur. Ce type d'implémentation est très contraignant 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.. . .

Nous proposons ici une organisation logicielle selon un *paradigme objet* où les optimiseurs, les simulateurs et les méta-modèles sont des objets de même niveau hiérarchique. L'utilisateur peut alors "jongler" avec ces différents objets pour se contruire des stratégie d'optimisation personnalisées.

*http://omd.lri.fr

†ENSM-SE (pujol@emse.fr)
‡CNRS/ENSM-SE (leriche@emse.fr)

*Scilab–The fastest thing from France since Django Reinhart*
*Christoph L. Spiel*

# Table des matières

## Remarques sur ce document

**Toolbox OMD** L'architecture Scilab décrite dans ce document est implémentée dans la toolbox Scilab "`OMD-toolbox`". Cependant, ce document n'est pas la documentation de cette toolbox.

**Exemples** La plupart des exemples de ce document sont fournis comme exemples dans la toolbox OMD (dossier `OMD-toolbox/examples/spec`).

**Notations** Dans ce document, on note $d$ la dimension de l'espace, et $n$ la taille d'un ensemble de points de cet espace. Par exemple, un échantillon aléatoire (ou *plan d'expériences*) est représenté par une matrice $n \times d$.

## Première partie

# Spécifications de l'architecture logicielle

## 1 Découplage optimiseur – simulateur

Optimiser, c'est se déplacer dans un domaine. On part d'un ou plusieurs point(s), puis on explore itérativement des nouveaux points, en tâchant de localiser des points de bonne performance. Traditionnellement, un algorithme d'optimisation décrit ce cheminement dans son ensemble : on l'implémente avec une boucle au milieu de laquelle figure l'appel au simulateur. Cette boucle est encapsulée dans une fonction, dont la fonction à optimiser en est un argument. Scilab ne déroge pas à cette règle ; par exemple, la fonction `optim` s'invoque de la manière suivante :

```
[f, xopt] = optim(costf, x0)
```

où `costf` est la fonction à optimiser.

Dans le contexte du projet RNTL/OMD, cette structure nous semble trop rigide. On souhaite en effet construire des scénarios d'optimisation plus évolués : par exemple, deux pas d'algorithme génétique, construction d'un méta-modèle, puis trois pas de gradient... La fonction à optimiser est elle aussi susceptible de varier en cours d'optimisation ; par exemple si l'on utilise un méta-modèle on veut pouvoir le ré-estimer, en changer, en estimer plusieurs simultanément... On voit naturellement se dessiner ici une gestion *objet* de l'algorithmique d'optimisation : une collection d'objets (optimiseurs, simulateurs, méta-modèles) à assembler en fonction des besoins.

Dans cet objectif (qui est un objectif de découplage), le plus délicat est de séparer la partie *optimisation* de la partie *simulation*. Pour y voir clair, observons le déroulement d'une procédure d'optimisation : c'est une séquence d'opérations, avec régulièrement des appels au simulateur, ce qui peut se représenter comme sur la figure 1. Ceci nous permet de donner une définition de l'optimiseur : c'est le morceau de programme qui se trouve entre deux séries de simulations. Une version équivalente du diagramme précédent, mais qui illustre mieux le couplage optimiseur – simulateur est représenté sur la figure 2a.

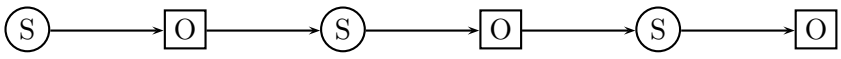



FIG. 1: Déroulement d'une procédure d'optimisation.

Un intérêt de cette approche est de permettre d'envisager, de manière lisible, d'autres scénarios. Par exemple, la figure 2b représente une stratégie d'optimisation dans laquelle on a simplement remplacé le simulateur par un méta-modèle. Le caractère modulaire est encore plus visible avec l'exemple de la figure 3 : le couplage d'un optimiseur avec un méta-modèle permet de construire un nouvel optimiseur, qui peut être ainsi couplé à un simulateur ; dans la stratégie qui est ainsi définie, faisant intervenir un optimiseur, un simulateur et un méta-modèle, l'optimisation est réalisée sur le méta-modèle, ce


(a) optimiseur – simulateur (b) optimiseur – méta-modèle

FIG. 2: Stratégies d'optimisation simples.

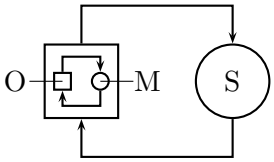



FIG. 3: Ré-estimation du méta-modèle en cours d'optimisation.

méta-modèle étant régulièrement ré-estimé par de nouvelles évaluations du simulateur. (L'implémentation de cette stratégie est présentée à la section 6.3.)

Un autre avantage de la programmation "objet" que nous proposons ici 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 sera importante si nous envisageons un jour des architectures de calcul distribuées.

**Remarque** Les fonctions d'optimisation existantes de Scilab (`optim`, `fsearch`, ...) ne rentrent pas dans le cadre qui vient d'être évoqué. Pour les utiliser, une petite adaptation sera nécessaire (voir section 9).

## 2 Scilab, les listes et la programmation objet

Avant de présenter les éléments de l'architecture, il est nécessaire de dévoiler les astuces de programmation Scilab qui permettent de disposer d'un ersatz de programmation objet[1].

**Types de listes** Nous allons utiliser les listes de manière systématique pour structurer les données et modulariser les programmes. Scilab définit quatre types de listes : `list`, `tlist`, `mlist` et `struct`.

Le type `list` est le type de liste le plus simple. Il contient des éléments de type arbitraire. Ces élements sont identifiés par un indice selon la syntaxe `l(i)`.

[1]Pour le lecteur qui trouverait cette section trop difficile en première lecture, les rudiments de programmation objet en Scilab sont présentés en section 7.

Les types `tlist` et `mlist` implémentent les listes typées qui permettent de définir des nouveaux types de données. Il s'agit déjà de programmation objet. Pour chaque nouveau type, on peut définir des actions associées par l'intermédiaire de la surcharge des opérateurs[2]. Les types `tlist` et `mlist` permettent de nommer leurs éléments, et ainsi fournir une syntaxe du type `objet.champ`, comme dans la majorité des langages orientés objets. Il n'y a pas de différence fondamentale entre ces deux types mais on préfèrera en général le type `mlist`[3].

Le type `struct` implémente les structures selon la syntaxe Matlab. Ce type est très pratique pour agréger différents objets dans une seule variable. Par exemple, les instructions suivantes créent la variable `v` contenant deux champs `x` et `y` :

```
v = struct()
v.x = ...
v.y = ...
```

La variable `v` n'aura pas de type particulier (autre que `st`). Elle pourra être passée comme argument d'une fonction, et c'est un moyen très pratique et lisible de passer un très grand nombre de paramètres (les éléments de la `struct`) à une fonction[4].

**Taxinomie** De manière plus abstraite, définissons quatre catégories de listes :

1. Une $\underline{\mathcal{L}\text{-liste}}$ ($\mathcal{L}$ pour liste) est une liste dont les éléments sont accessibles par un indice, selon la syntaxe `l(i)`. En Scilab, cela correspond au type `list` ( ainsi qu'aux types `tlist` et `mlist` avec une surcharge adéquate de l'opérateur `()`).
2. Une $\underline{\mathcal{S}\text{-liste}}$ ($\mathcal{S}$ pour structure) est une liste dont les éléments sont nommés et accessibles selon la syntaxe `l.nom`. En Scilab, cela correspond aux types `tlist`, `mlist` et `struct`.
3. Une $\underline{\mathcal{O}\text{-liste}}$ ($\mathcal{O}$ pour objet) est une $\mathcal{S}$-liste, typée[5], et pour laquelle on peut surcharger des opérateurs. Cela correspond en Scilab aux types `tlist` et `mlist`.
4. Une $\underline{\mathcal{F}\text{-liste}}$ ($\mathcal{F}$ pour fonction) est une $\mathcal{O}$-liste pour laquelle on a surchargé l'opérateur d'appel de fonction `()`. Dans la littérature informatique (par exemple dans la librairie standard du C++), on appelle ceci un *objet fonction* ou *foncteur*. Les $\mathcal{F}$-listes correspondent en Scilab aux types `tlist` et `mlist` pour lesquels on surcharge l'opérateur d'appel de fonction `()`.

Dans la suite de ce document, l'architecture est spécifiée en termes de $\{\mathcal{L}, \mathcal{S}, \mathcal{O}, \mathcal{F}\}$-listes. Sauf mention explicite du contraire, le programmeur est libre d'utiliser le type de

[2] La surcharge d'une fonction (ou d'un opérateur) consiste à définir une fonction qui existe déjà mais pour d'autres types de données. Pour plus de détails, se reporter à la documentation de Scilab à la rubrique `overloading`.

[3] Voir la documentation de Scilab à la rubrique `mlist` pour la différence avec le type `tlist`.

[4] Le type `struct`, est une surcharge du type `mlist` et a été ajouté à Scilab pour émuler le type de données équivalent de Matlab. Les opérations sur le type `struct` dans Scilab ne sont donc pas très performantes.

[5] Voir la documentation de Scilab à la rubrique `typeof`.

liste de son choix (`list`, `tlist`, `mlist`, `struct`) correspondant à la catégorie idoine[6].

**Construction d'un objet** Nous allons présenter le principe de la construction d'un objet sur un exemple simple, celui d'un méta-modèle de type régression linéaire :

$$y_i = \beta_0 + \sum_{j=1}^{d} \beta_j x_{ij} + \varepsilon_i \quad (i = 1 \dots n)$$

où les $\varepsilon_i$ sont $n$ réalisations indépendantes d'une loi $\mathcal{N}(0, \sigma^2)$. Sous forme matricielle, on écrit $y = X\beta + \varepsilon$, avec

$$y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} 1 & x_{11} & \dots & x_{1d} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \dots & x_{nd} \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_d \end{pmatrix}, \quad \varepsilon = \begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

Pour estimer les paramètres $\beta$ et $\sigma^2$, on choisit la méthode des moindres carrés pondérés qui permet d'affecter un poids $w_i$ à chaque observation $y_i$[7]. Avec cette méthode d'estimation, on cherche à minimiser l'erreur quadratique pondérée entre les observations $y_i$ et les prédictions $\tilde{y}_i = \beta_0 + \sum_{j=1}^{d} \beta_j x_{ij}$ :

$$\min_{\beta} \sum_{i=1}^{n} w_i (y_i - \tilde{y}_i)^2$$

L'estimation du vecteur $\beta$ est donnée par la formule de Gauss-Markov :

$$\hat{\beta} = (X'WX)^{-1} X'Wy \tag{1}$$

où $W = w\ \mathrm{I}_n$, $w = (w_1, \dots, w_n)'$. L'estimation de la variance du résidu est donnée par :

$$\hat{\sigma^2} = \frac{\|y - X\hat{\beta}\|^2}{n - d - 1} \tag{2}$$

**Implémentation** Les paramètres estimés $\hat{\beta}$ et $\hat{\sigma^2}$, ainsi toutes les autres données relatives à la régression vont être stockés dans un objet. Cet objet, de type `linreg`[8] sera un objet fonction (type abstrait $\mathcal{F}$-liste, implémenté par le type `mlist`) contenant les champs suivants :

[6] Le programmeur pourra regretter de ne pas disposer d'un seul type de liste s'adaptant à tous les cas (comme le propose le langage R)...

[7] Les moindres carrés pondérés sont typiquement utilisés dans un cas d'hétéroscédasticité, c'est-à-dire quand le résidu dépend du point $x$ (par exemple, l'erreur de mesure dépend du point où est faite la mesure).

[8] Scilab ne reconnaît pas les noms de types dépassant 8 caractères. Cette contrainte devrait disparaître dans une prochaine version.

– `weights` : vecteur des poids (vecteur de taille $n$).
– `X` : plan d'expériences (matrice $n \times d$). C'est la matrice notée précédemment $X$ sans la première colonne de 1.
– `y` : observations associées aux points du plan d'expériences (vecteur de taille $n$).
– `beta` : estimation des coefficients $\beta$, noté précédemment $\hat{\beta}$ (vecteur de taille $d+1$).
– `sigma2` : estimation de la variance du résidu, noté précédemment $\hat{\sigma^2}$ (scalaire).

L'objet, appelé `this`[9], est déclaré comme une `mlist` contenant les champs désirés :

```
this = mlist(['linreg', 'weights', 'X', 'y', 'beta', 'sigma2'])
```

L'utilisateur fournit la valeur des champs `weights`, `X` et `y`. On suppose pour l'instant qu'ils sont correctement initialisés. Il reste donc à initialiser les champs `beta` et `sigma2`, qui sont calculés selon les formules (1) et (2) :

```
[n, d] = size(this.X)
X = [ones(n, 1), this.X]
y = matrix(this.y, -1)  // (on s'assure que y soit un vecteur colonne)
W = diag(this.weights)
this.beta = (X' * W * X) \ (X' * W * y)
this.sigma2 = sum((y - this.beta).^2) / (n - d - 1)
```

Toutes ces instructions, qui constituent l'initialisation de l'objet "régression linéaire" (ou `linreg`) peuvent être regroupées dans une fonction que l'on appelle *constructeur* :

```
1   function this = linreg(X, y, param)
2     this = mlist(['linreg', 'weights', 'X', 'y', 'beta', 'sigma2']
3     this.weights = param.weights
4     this.X = X
5     this.y = y
6     [n, d] = size(this.X)
7     X = [ones(n, 1), this.X]
8     y = matrix(this.y, -1)
9     W = diag(this.weights)
10    this.beta = (X' * W * X) \ (X' * W * y)
11    this.sigma2 = sum((y - X * this.beta).^2) / (n - d - 1)
12  endfunction
```

Notons que le paramètre `weights` est passé au constructeur via la structure `param`. En effet, les paramètres ne peuvent pas être passés directement au constructeur, mais doivent être auparavant encapsulés dans une structure ($\mathcal{S}$-liste). C'est une contrainte de l'architecture pour permettre d'avoir une syntaxe similaire entre tous les méta-modèles.

Donnons un exemple d'utilisation de ce contructeur par l'utilisateur final. Supposons que les variables `X` et `y` contiennent respectivement le plan d'expériences et les observations correspondantes. Alors on pourra construire un objet régression très simplement :

[9] Référence non dissimulée au C++.

```
param = struct();
param.weights = ones(1, length(y)); // on met le même poids à
                                    // toutes les observations
ma_regression = linreg(X, y, param);
```

**Écriture de méthodes pour les objets** Toujours sur l'exemple de la régression linéaire, nous souhaitons utiliser notre objet pour effectuer des prédictions. Il faut donc implémenter la fonction qui effectue ces prédictions. Comme l'objet de type `linreg` est une `mlist`, on va surcharger[2] l'opérateur d'appel de fonction `()` de manière à ce que les prédictions se fassent selon la syntaxe des fonctions : `y = ma_regression(x)`. Cette surcharge est implémentée par la fonction suivante :

```
13  function [y, v] = %linreg_e(X, this)  // "_e" pour extraction...
14    n = size(X, 1)
15    y = [ones(n, 1), X] * this.beta
16    v = this.sigma2 * ones(n, 1)
17  endfunction
```

Cette fonction renvoie la (ou les) valeur(s) prédite(s), ainsi que l'estimation de la variance au(x) point(s) prédit(s), ici constante.

**Extension du mécanisme de surcharge** Le langage Scilab permet de surcharger la majorité des opérateurs (`+`, `()`, ...), ainsi que certaines fonctions internes (`disp`, `plot2d`, ...). Ce mécanisme permet une certaine généricité, mais nous souhaitons définir de nouvelles fonctions génériques, par exemple la fonction `ask` qui se rapporte à un optimiseur :

```
x = ask(my_optimizer)
```

Lors de l'appel à la fonction `ask`, l'optimiseur `my_optimizer` propose, en fonction de son type et de son état interne, un jeu de variables `x` à simuler. La fonction `ask` n'est pas une fonction interne de Scilab, donc normalement, elle ne peut pas être surchargée. Néanmoins, pour les besoins de l'architecture, nous avons développé un mécanisme de substitution qui autorise cette surcharge. Ainsi, le développeur d'un optimiseur pourra écrire la fonction suivante :

```
function x = %<type_of_optimizer>_ask(optimizer)
```

L'appel à la fonction *générique* `ask` sera redirigé[10] vers la fonction *spécifique* `%<typeof_optimizer>_ask`. Ce mécanisme permet d'intégrer sans programmation supplémentaire le code des développeurs dans la plate-forme. Mais ce mécanisme permet surtout de construire des stratégies d'optimisation indépendamment des objets manipulés (voir section 6).

[10] La fonction `evstr` qui permet d'évaluer une instruction donnée comme une chaîne de caractères, ce qui permet de créer de nouvelles instructions de manière dynamique. Ainsi, le corps de la fonction `x = ask(this)` est le suivant : `x = evstr('%' + typeof(this) + '_ask(this)')`.

# 3 Simulateur

Un simulateur est un objet qui va réaliser l'évaluation des fonctions d'un problème d'optimisation de la forme suivante :

$$\begin{cases} \min\limits_{\vec{x}} f_i(\vec{x}) & i = 1 \dots p \\ g_j(\vec{x}) \leq 0 & j = 1 \dots q \\ h_k(\vec{x}) = 0 & k = 1 \dots r \end{cases}$$

Les $f_i$ sont les fonctions objectives, les $g_j$ les contraintes inégalités et les $h_k$ les contraintes égalités. Un simulateur renvoie donc pour une valeur du vecteur $\vec{x}$ une valeur pour chaque fonction du problème d'optimisation. Le simulateur peut également renvoyer les gradients de ces fonctions s'ils sont calculés, ainsi que toutes les données provenant du code de calcul jugées utiles, typiquement des données sur le déroulement de la simulation (critères de convergence...).

Sur le plan de l'implémentation en Scilab, un simulateur doit pouvoir s'invoquer comme une fonction :

```
y = nom_du_simulateur(x)
```

avec

- `x` : un vecteur ligne de dimension $d$.
- `y` : les différents éléments de la réponse, objectifs, contraintes, gradients..., sous la forme d'un objet de type "criteria" (voir paragraphe suivant).

Sauf cas très simple, un simulateur n'est jamais une fonction mais un objet fonction ($\mathcal{F}$-liste).

**Objet de type "criteria"** La variable `y` est un objet ($\mathcal{O}$-liste) de type `criteria` contenant les champs suivants :

- `objective` : les valeurs des fonctions objectifs (scalaire ou vecteur),
- `list_grad_objective` : les numéros des fonctions objectifs dont les gradients sont calculés (scalaire ou vecteur),
- `grad_objective` : les valeurs des gradients des fonctions objectif (vecteur ou matrice),
- `constraint`, `list_grad_constraint`, `grad_constraint` : mêmes définitions pour les contraintes d'inégalités,
- `econstraint`, `list_grad_econstraint`, `grad_econstraint` : mêmes définitions pour les contraintes d'égalités,

ainsi que tous les champs nécessaires pour les sorties additionnelles. Tous ces champs sont optionnels.

Rajoutons trois contraintes d'intégrité :

1. Les tailles d'un champ principal (`objective`, `constraint`, `econstraint`) et du champ des gradients associé (`grad_`) doivent être compatibles :
   - si le champ principal est un scalaire, alors le champ des gradients est un vecteur de taille $d$.

– si le champ principal est un vecteur, alors le champ des gradients est une matrice de $d$ colonnes. Les gradients sont stockés en lignes : le nombre de lignes correspond au nombre de gradients calculés (donnés par le champ `list_grad_`).

2. Si tous les gradients sont calculés, le champ `list_grad_` est optionnel.
3. Dans le cas où des dérivées ne sont pas calculées par rapport à toutes les variables, on retournera `%nan` pour les valeurs non calculées. Par exemple en dimension 3 avec une seule fonction objective $f$, si le code calcule uniquement les dérivées de $f$ par rapport à $x_1$ et $x_3$ , alors le champ `grad_objective` contiendra le vecteur `[` $\partial f/\partial x_1$ `, %nan,` $\partial f/\partial x_3$ `]`.

**Exemple** Nous allons implémenter un simulateur ayant pour simple objectif la fonction $f(x_1, x_2) = x_1 + \alpha x_2$, sans contrainte. Dans un premier temps, le paramètre $\alpha$ sera considéré comme un paramètre interne du simulateur, donc invisible pour l'utilisateur. Ensuite, le paramètre $\alpha$ pourra être fixé par l'utilisateur.

**Version non paramétrée** La première implémentation se fait par une simple fonction :

```
function y = f(x)
  alpha = 2
  y = mlist(['criteria', 'objective', 'grad_objective'])
  y.objective = x(1) + alpha * x(2)
  y.grad_objective = [1, alpha]
endfunction
```

La valeur du paramètre $\alpha$ est fixée à l'intérieur de cette fonction. L'utilisateur ne peut donc pas le modifier. Bien remarquer la composition de la variable de sortie `y` : la valeur de la fonction objectif et la valeur du gradient de cette fonction (on aurait pu ne pas spécifier ce gradient).

**Paramétrage par variable globale** Pour la seconde implémentation, l'utilisateur doit pouvoir fixer la valeur du paramètre $\alpha$. La première idée est de passer ce paramètre à la fonction définie précédemment :

```
function y = f(x, alpha)                                    // incorrect
  y = mlist(['criteria', 'objective', 'grad_objective'])    // incorrect
  y.objective = x(1) + alpha * x(2)                         // incorrect
  y.grad_objective = [1, alpha]                             // incorrect
endfunction                                                 // incorrect
```

Cette implémentation est syntaxiquement correcte, mais n'est pas compatible avec l'architecture. En effet, dans ce cas, le simulateur s'invoquerait sous la forme `y = f(x, alpha)`, alors qu'il doit s'invoquer sous la forme `y = f(x)`. Une solution simple pour contourner ce problème consiste à définir la variable `alpha` comme une variable globale :

| Simulateur non paramétré | Simulateur paramétré | |
|---|---|---|
| | "variable globale" | "objet fonction" |
| `y = f(x)` | `global alpha`<br>`alpha = 2`<br>`y = f(x)` | `param = struct()`<br>`param.alpha = 2`<br>`f = monsimu(param)`<br>`y = f(x)` |

TAB. 1: Comparaison des différentes implémentations possibles d'un simulateur du point de vue de l'utilisateur.

```
function y = f(x)
  global alpha
  y = mlist(['criteria', 'objective', 'grad_objective'])
  y.objective = x(1) + alpha * x(2)
  y.grad_objective = [1, alpha]
endfunction
```

L'utilisateur peut fixer la valeur du paramètre $\alpha$ de l'extérieur de la fonction.

**Objet fonction** Une autre méthode consiste à définir le simulateur paramétré comme un objet fonction ($\mathcal{F}$-liste), avec un constructeur qui fixe la valeur du paramètre $\alpha$, et la surcharge de l'opérateur de fonction `()` pour l'appel au simulateur :

1. constructeur du type `monsimu` :

   ```
   function this = monsimu(param)
     this = mlist(['monsimu', 'alpha'], param.alpha)
   endfunction
   ```

2. surcharge de l'opérateur `()` pour le type `monsimu` :

   ```
   function y = %monsimu_e(x, this)
     y = mlist(['criteria', 'objective', 'grad_objective'])
     y.objective = x(1) + this.alpha * x(2)
     y.grad_objective = [1, this.alpha]
   endfunction
   ```

Cette implémentation est un peu plus longue qu'avec les variables globales, mais certainement plus propre au niveau de la gestion de la mémoire.

Le tableau 1 compare ces trois implémentations dans la manière de les utiliser. Comme désiré, l'appel `y = f(x)` est le même dans tous les cas. L'utilisation d'un objet fonction ($\mathcal{F}$-liste) est légèrement plus lourde, mais le gain en modularité est réel.

**Remarque** Le principal travail pour la programmation d'un simulateur réside dans l'interfaçage avec le code de calcul. Pour les cas où le code de calcul a une communication par fichiers, des routines facilitant ce travail ont été réalisées : ce sont les fonctions `template_replace` et `parse`.

**Méthodes associées au type "criteria"** Deux méthodes ont été définies pour le type `criteria` :

1. affichage : `disp(y)`
2. extraction : `y('objective', 1)`...

Se rapporter à l'aide en ligne de la toolbox "architecture" pour plus de précisions.

# 4 Optimiseur

Rappelons la définition d'un optimiseur (voir section 1) : un optimiseur est le morceau de programme qui se trouve entre deux séries de simulations. Le simulateur 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`. Ajoutons à cela la condition d'arrêt (*"as-tu fini ?"*, méthode `stop`) et la lecture de l'optimum (*"quelle est la meilleure valeur connue ?"*, méthode `best`). La communication avec un optimiseur repose donc sur quatre fonctions : `ask`, `tell`, `stop` et `best` qui devront être implémentées selon les modèles suivants :

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

où `<optimizer_type>` est à remplacer par le type (`typeof`) que l'on souhaite donner à l'optimiseur. Ajoutons une cinquième fonction, le constructeur, pour initialiser l'optimiseur :

```
function this = <optimizer_type>(param)
```

On peut aussi surcharger la fonction `disp` de Scilab pour avoir un affichage personnalisé :

```
function %<optimizer_type>_p(this)
```

Ce qui fait au final six fonctions à programmer pour implémenter un optimiseur. Les quatre premières s'utilisent dans ce que nous appelons des schémas de couplage, présentés en section 6 ; l'étude de ces schémas est le meilleur moyen de comprendre le rôle de chaque fonction.

Détaillons les arguments de ces fonctions :

- `this` : objet optimiseur ($\mathcal{F}$-liste)
- `param` : paramètres pour initialiser l'optimiseur, regroupés dans une structure ($\mathcal{S}$-liste)
- `x` : point(s) demandé(s) par l'optimiseur (vecteur ligne ou matrice, selon que l'optimiseur demande un point ou plusieurs points)
- `y` : réponses attendues par l'optimiseur ; `y` n'est pas un objet de type `criteria`, mais plus simplement un scalaire, un vecteur, ... (voir exemples plus loin)
- `out` : condition d'arrêt (booléen)
- `xopt`, `yopt` : meilleur point connu (vecteur ligne), et valeur(s) associée(s)

**Remarque** Il est important de noter la syntaxe particulière de la méthode `tell`, avec l'optimiseur présent aussi bien en entrée qu'en sortie :

```
my_optimizer = tell(my_optimizer, x, y)
```

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 méthode `tell` est la seule pouvant mettre à jour l'état de l'optimiseur ; c'est pour cette raison que la variable `x` (générée préalablement par la méthode `ask`) est rappelée en argument : la méthode `tell` se chargera de la sauvegarde des points `x` dans l'optimiseur.

**Exemple** Nous allons programmer un optimiseur de type "recherche aléatoire" : cet optimiseur renverra l'optimum connu sur une population de points tirée au hasard. Cet optimiseur sera mono-objectif et ne tiendra pas compte de contraintes.

Les paramètres sont les suivants :

- `x_min` : bornes inférieures de l'espace (vecteur de taille `d`)
- `x_max` : bornes supérieures de l'espace (vecteur de taille `d`)
- `nb_max_iter` : nombre maximum d'itérations (entier)

et les variables internes sont les suivantes :

- `iter` : itération courante (entier)
- `xopt` : meilleur point connu (vecteur)
- `yopt` : réponse associée à `xopt` (scalaire)

Le constructeur initialise l'objet avec les paramètres fournis par l'utilisateur :

```
1  function this = rsearch(param)
2    this = mlist(['rsearch', 'd', 'x_min', 'x_max', ...
3                  'nb_max_iter', 'iter', 'xopt', 'yopt'])
4    this.d = param.d
5    this.x_min = getfield_deft('x_min', param, zeros(1, param.d))
6    this.x_max = getfield_deft('x_max', param, ones(1, param.d))
7    this.nb_max_iter = param.nb_max_iter
8    this.iter = 0
9    this.xopt = []
10   this.yopt = %inf
11 endfunction
```

Noter l'emploi de la fonction `getfield_deft`, implémentée dans la toolbox "architecture", et qui permet de gérer très simplement les valeurs par défaut ; dans cet exemple, si l'utilisateur ne spécifie pas les valeurs de `x_min` et `x_max` dans la structure `param`, alors le constructeur affectera à ces bornes $0_d$ et $1_d$ respectivement.

La méthode `stop` se contente de tester le nombre d'itérations effectuées :

```
12 function out = %rsearch_stop(this)
13   out = this.iter >= this.nb_max_iter
14 endfunction
```

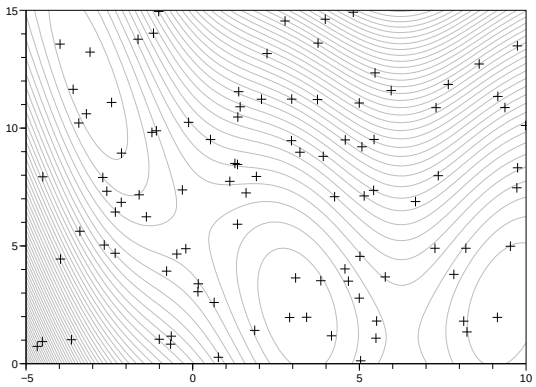



FIG. 4: Test de l'optimiseur `rsearch` sur la fonction de Branin. Comme attendu, les points se répartissent aléatoirement sur tout le domaine.

La méthode `ask` génére un vecteur aléatoire dans le domaine délimité par les bornes `x_min` et `x_max` :

```
15  function x = %rsearch_ask(this)
16    x = (this.x_max - this.x_min) .* grand(1, this.d, 'def') + this.x_min
17  endfunction
```

La méthode `tell` se contente d'incrémenter le compteur du nombre d'itérations et de sauvegarder `x` et `y` si ce point est meilleur que celui déjà connu :

```
18  function this = %rsearch_tell(this, x, y)
19    this.iter = this.iter + 1
20    if y < this.yopt then
21      this.xopt = x
22      this.yopt = y
23    end
24  endfunction
```

Enfin, la méthode `best` renvoie simplement la valeur de `xopt` et `yopt` mémorisée dans l'objet :

```
25  function [yopt, xopt] = %rsearch_best(this)
26    yopt = this.yopt
27    xopt = this.xopt
28  endfunction
```

Un exemple d'utilisation de l'optimiseur que nous venons de programmer est donné par la figure 4.

# 5 Méta-modèle

Un méta-modèle est une approximation d'un élément de la réponse d'un simulateur (par exemple, une fonction objectif, une contrainte...). Cette approximation est construite sur la base d'un ensemble de points $(\vec{x}_i, \vec{y}_i)_{i=1...n}$, appelée dans la suite *base de construction*.

Les paramètres d'un méta-modèle sont :

1. les paramètres propres du méta-modèle (par exemple, le degré de la régression polynomiale). Ils sont donnés par une structure ($\mathcal{S}$-liste).
2. les points de la base de construction. Ils sont donnés par une matrice $n \times d$.
3. les réponses correspondantes. Elles sont données par un vecteur de taille $n$ pour une réponse scalaire, et par une matrice de $n$ lignes pour une réponse vectorielle.

Ces trois informations permettent de construire (ou *estimer*) le méta-modèle :

```
g = metamodel(X, y, param)
```

où `metamodel` est le nom du constructeur ; par convention, ce sera le même nom que le type du méta-modèle. On voit déjà qu'un méta-modèle est un objet ($\mathcal{O}$-liste). C'est en fait un objet fonction ($\mathcal{F}$-liste) : l'opérateur d'appel de fonction `()` servira à invoquer les prédictions :

```
[y, v] = g(x)
```

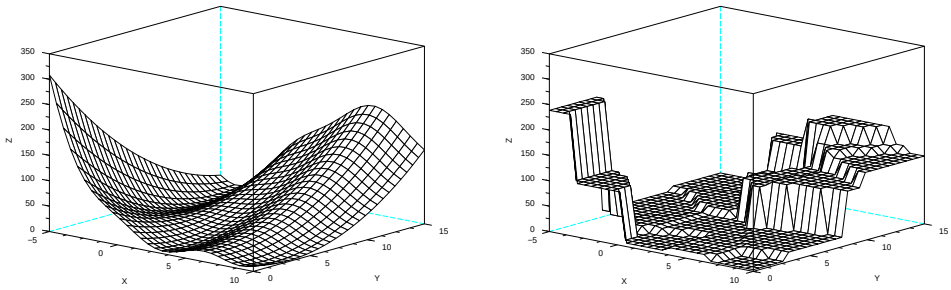
- contrairement au cas d'un simulateur, `y` n'est plus un objet de type `criteria`, mais un simple vecteur (ou un scalaire). Le fait qu'on ne retrouve pas pour les méta-modèles les mêmes sorties que pour les simulateurs est nécessaire. En effet, un même méta-modèle doit pouvoir modéliser aussi bien des fonctions objectifs, des contraintes, des gradients... Au besoin, plusieurs méta-modèles pourront être agrégés pour représenter l'ensemble de la réponse d'un simulateur.
- `v` est un indicateur la dispersion au point prédit ; par exemple, dans le cas classique de la régression, la variance du résidu (le $\varepsilon$), et dans le cas du krigeage, la variance de krigeage...

**Remarque** Si l'on ne s'intéresse pas à la dispersion au point prédit, Scilab permet d'invoquer un méta-modèle sous la forme simplifiée :

```
y = g(x)
```

**Exemple** Nous allons programmer un méta-modèle très simple : pour un point de l'espace, la prédiction sera la valeur du point le plus proche connu. Ce méta-modèle correspond en une dimension à une fonction constante par morceaux ; en plusieurs dimensions, les morceaux sont les cellules de Voronoï centrées sur les points de la base de construction. On nomme `nearest` ce méta-modèle.

Le constructeur initialise le métamodèle :




(a) fonction de Branin (b) méta-modèle `nearest`

FIG. 5: Test du méta-modèle `nearest` sur la fonction de Branin. Les plateaux se découpent selon les cellules de Voronoï.

```
1   function this = nearest(X, y, param)
2     rhs = argn(2)
3     if rhs == 2 then param = struct(), end
4     this = mlist(['nearest', 'X', 'y'], X, y)
5   endfunction
```

Le méta-modèle n'ayant pas de paramètre, on permet à l'utilisateur de l'initialiser sans avoir à passer de structure vide pour `param`, ce qui est implémenté par les deux premières lignes.

L'opérateur de fonction effectue les prédictions, en renvoyant la valeur du point connu le plus proche :

```
6   function [y, v] = %nearest_e(x, this)
7     // d est le vecteur des distances (au carré) de x
8     // à tous les points du plan X
9     d = []
10    for i = 1:size(this.X, 1)
11      d = [d, (sum((this.X(i,:) - x).^2))]
12    end
13    // on retourne la valeur du point dont la distance est minimale
14    [val, i] = min(d)
15    y = this.y(i)
16    v = %nan // pas d'estimation de l'erreur
17  endfunction
```

Un exemple d'utilisation du méta-modèle que nous venons de programmer est donné par la figure 5.

# 6 Couplage : schémas “ask & tell”

Cette section présente des algorithmes simples de couplage entre optimiseurs, simulateurs et méta-modèles.

## 6.1 Optimiseur “1-point”

Un optimiseur “1-point” est un optimiseur qui demande l'évaluation d'un seul point à chaque itération.

**Avec une fonction** C'est le schéma de base :

```
while ~stop(optimizer)
  x = ask(optimizer);
  y = f(x);
  optimizer = tell(optimizer, x, y);
end
```

**Avec un simulateur** Comme un simulateur renvoie un objet de type `criteria`, il faut en extraire la ou les valeur(s) d'intérêt, ici la première fonction objectif :

```
while ~stop(optimizer)
  x = ask(optimizer);
  y = simulator(x);
  obj1 = y('objective', 1);
  optimizer = tell(optimizer, x, obj1);
end
```

## 6.2 Optimiseur “n-points”

Un optimiseur “n-point” est un optimiseur qui demande l'évaluation de plusieurs point à chaque itération (typiquement un algorithme évolutionnaire).

**Avec une fonction** Comme l'optimiseur demande plusieurs points, il faut faire une boucle pour les évaluer :

```
while ~stop(optimizer)
  X = ask(optimizer)
  n = size(X, 1);
  y = zeros(1, n);
  for i = 1 : n
    y(i) = f(X(i,:));
  end
  optimizer = tell(optimizer, X, y);
end
```

**Avec un simulateur**

```
while ~stop(optimizer)
  X = ask(optimizer);
  n = size(X, 1);
  obj1 = zeros(1, n);
  for i = 1 : n
    y = simulator(X(i,:));
    obj1(i) = y('objective', 1);
  end
  optimizer = tell(optimizer, X, obj1);
end
```

## 6.3 Ré-estimation d'un méta-modèle en cours d'optimisation

On présente maintenant l'implémentation de la stratégie d'optimisation suivante : on dispose d'un budget de $N$ simulations ; un plan d'expériences de $n < N$ points est généré, les simulations correspondantes sont réalisées et un méta-modèle est estimé ; ensuite de manière itérative ($N - n$ itérations), l'optimisation est réalisée sur le méta-modèle, l'optimum trouvé est évalué par le simulateur, cette valeur servant à ré-estimer le méta-modèle pour l'itération suivante. Cette stratégie a été illustrée au début de ce document par la figure 3 (page 6).

```
// Paramètres
N = 20; // nb total de simulations
n = 10; // 1er batch pour estimer le méta-modèle
        // (il en reste donc N-n pour l'optim)

// Plan d'expériences initial de taille n
// (on se sert de l'optimiseur 'rsearch' comme d'un générateur aléatoire)
opt_param = struct();
opt_param.d = ...;     // dimension de l'espace
opt_param.x_min = ...;  // bornes min
opt_param.x_max = ...;  // bornes max
opt_param.nb_max_iter = n;
optimizer = rsearch(opt_param);
while ~stop(optimizer)
  x = ask(optimizer);
  y = f(x);
  optimizer = tell(optimizer, x, y);
end

// Les données sont stockées dans une structure nommée 'archive'
archive = struct();
```

```
archive.X = optimizer.X;
archive.y = optimizer.y;

// Les N-n simulations qui restent...
for i = n+1 : N

  // initialisation de l'optimiseur
  // (on garde rsearch, mais on peut en changer...)
  opt_param.nb_max_iter = 100;
  optimizer = rsearch(opt_param);

  // (ré)-estimation du méta-modèle
  metamodel = nearest(archive.X, archive.y);

  // optimisation sur le méta-modèle
  while ~stop(optimizer)
    x = ask(optimizer);
    y = metamodel(x);
    optimizer = tell(optimizer, x, y);
  end

  // l'optimum trouvé sur le méta-modèle est évalué par le simulateur...
  [yopt, xopt] = best(optimizer);
  yopt = f(xopt);

  // ... et ajouté à l'archive de points
  archive.X = [archive.X; xopt];
  archive.y($+1) = yopt;
end

[yopt, xopt] = best(optimizer);
```

## Deuxième partie

# Tutoriels

## 7 La programmation objet en Scilab “pour les nuls”

Dans cette section, nous présentons les rudiments de programmation objet avec Scilab.

On définit un vecteur en dimension 2 comme un objet grâce à une `mlist` :

```
-->v = mlist(['vect2d', 'x', 'y'], 0., 0.);
```

La liste `v` a le type `vect2d` et contient deux champs, nommés `x` et `y`, valant tous les deux 0.

Le type de l'objet est renvoyé par la fonction `typeof` :

```
-->typeof(v)

 vect2d
```

Les champs se manipulent en lecture / écriture selon la syntaxe bien connue :

```
-->v.x = 1;

-->v.x
 ans  =

    1.
```

Affichons le contenu de l'objet :

```
-->v

       v(1)

!vect2d  x  y  !

       v(2)

    0.

       v(3)

    0.
```

On voit ici comment sont stockés les éléments dans la liste : le premier élément mémorise le type de l'objet et les noms des champs, les éléments suivants mémorisent les valeurs des champs.

Cet affichage n'est pas très joli, mais il peut être personnalisé en *surchargeant*[11] la fonction `disp`[12] pour le type `vect2d` :

```
-->function %vect2d_p(v)
-->printf("x: %f\n", v.x)
-->printf("y: %f\n", v.y)
-->endfunction
```

Ainsi, l'affichage est plus agréable :

```
-->v

x: 0.000000
y: 0.000000
```

Noter que l'on peut ainsi cacher des champs d'un objet ; dans notre exemple, on aurait pu ne pas afficher `y`.

Pour connaître le nom de tous les champs, y compris les champs "cachés", il faut passer par la fonction `getfield` pour lire la valeur du premier élément de la liste :

```
-->getfield(1, v)

!vect2d  x  y  !
```

Pour manipuler plusieurs vecteurs avec une syntaxe simple, on définit un *constructeur* qui se charge d'initialiser l'objet :

```
-->function v = vect2d(x, y)
-->v = mlist(['vect2d', 'x', 'y'], x, y)
-->endfunction
```

On peut donc ainsi facilement créer plusieurs objets différents

```
-->v1 = vect2d(0, 1);

-->v2 = vect2d(1, 0);
```

On surcharge l'opérateur `+` pour définir la somme de deux vecteurs :

```
-->function v3 = %vect2d_a_vect2d(v1, v2)
-->v3 = vect2d(v1.x + v2.x, v1.y + v2.y)
-->endfunction
```

[11] La syntaxe de la surcharge est expliquée dans l'aide de Scilab à la rubrique `overloading`.

[12] Pour mémoire, la fonction `disp` est appelée implicitement lorsque l'on évalue la variable sans mettre de point virgule à la fin de l'instruction.

Ainsi, on peut écrire :

```
-->v3 = v1 + v2

x: 1.000000
y: 1.000000
```

On peut ainsi redéfinir la plupart des opérateurs de Scilab.

Voilà pour les rudiments. Pour approfondir, le point d'entrée est l'aide de la fonction `mlist`, qui renvoie ensuite vers l'aide de toute les fonctions utiles.

# 8 Programmer un optimiseur "ask & tell"

Programmer un optimiseur en "ask & tell" n'est pas très compliqué une fois que l'on a compris la logique, qui va un peu à l'encontre de ce qui se fait traditionnellement. Pour s'y faire, le plus efficace est de programmer un optimiseur selon la logique traditionnelle, et de le faire évoluer peu à peu vers la logique découplée. C'est ce que nous allons illustrer dans cette section.

L'optimiseur que nous avons choisi de programmer est un des plus simples, il s'agit d'une méthode de descente en 1D. La suite de points $(x_k)_{k=0...n}$ est définie par :

$$\begin{aligned} x_0 & \quad \text{donné} \\ x_{k+1} & = x_k - \alpha \nabla_k f, \quad k \geq 0 \end{aligned}$$

où $\nabla_k f$ est l'estimation du gradient par différences finies :

$$\begin{aligned} \nabla_0 f & \quad \text{donné} \\ \nabla_k f & = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}, \quad k \geq 1 \end{aligned}$$

Le paramètre $\alpha$ est constant et fixé par l'utilisateur (il s'agit donc de l'algorithme du gradient à pas constant). L'algorithme s'arrête lorsque le gradient est suffisamment petit :

$$||\nabla_n f|| < \varepsilon$$

où $\varepsilon$ est aussi fixé par l'utilisateur.

**But à atteindre** L'optimiseur dans sa version finale doit pouvoir s'utiliser dans une boucle de la forme :

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

Dans cette boucle :

1. l'état de l'optimiseur (i.e. ses variables internes) n'est mis à jour que lors de l'appel de la méthode `tell`, ce qui veut dire que les variables internes de l'optimiseur ne pourront pas être modifiées lors de l'appel à la méthode `ask`,
2. il n'y a pas d'appel à la fonction $f$ en dehors de la boucle ; au départ de la boucle, l'optimiseur vient juste d'être initialisé, et aucune simulation n'a encore été faite ; comme il faut initialement deux point dans notre exemple pour estimer un gradient, la première itération consistera simplement à effectuer la simulation correspondant au point initial $x_0$.

Then, let's start the coding !

**Première itération** Pour commencer, programmons l'algorithme selon la logique "traditionnelle" :

```
1   // function to optimize
2   function y = f(x)
3     y = x^2
4   endfunction
5
6   // setting parameters
7   x0 = 5.;
8   epsilon = 1.E-2;
9   alpha = 1.E-1;
10  grad = 1;
11
12  // initializations
13  xcur = x0;
14  ycur = f(xcur);
15
16  // optimization loop
17  while ~(abs(grad) < epsilon)
18    // i++
19    xlast = xcur;
20    ylast = ycur;
21
22    // point to evaluate
23    xcur = xcur - alpha * grad;
24    ycur = f(xcur);
25
26    // updating the gradient
27    grad = (ycur - ylast) / (xcur - xlast);
28
29    // output
```

```
30    printf('x = %f   y = %f\n', xcur, ycur);
31  end
```

Ce script nous permet d'identifier les variables de l'optimisation :

| $\varepsilon$ | $\alpha$ | $\nabla_k f$ | $x_0$ | $x_k$ | $f(x_k)$ | $x_{k-1}$ | $f(x_{k-1})$ |
|---|---|---|---|---|---|---|---|
| `epsilon` | `alpha` | `grad` | `x0` | `xcur` | `ycur` | `xlast` | `ylast` |

Ces variables seront les variables internes de l'optimiseur.

**Deuxième itération** Maintenant que les variables internes de l'optimiseur sont identifiées, nous allons modifier le script pour éviter que ces variables ne soient modifiées avant l'appel au simulateur (premier point noté au paragraphe "But à atteindre"). Ce sont ici les variables `xcur`, `ycur`, `xlast` et `ylast` qui sont concernées. Pour cela, nous introduisons des variables temporaires, `x` et `y`, et rejetons la mise à jour des variables internes de l'optimiseur en fin de boucle. Ce qui donne la version suivante :

```
1   // function to optimize
2   function y = f(x)
3     y = x^2
4   endfunction
5
6   // setting parameters
7   x0 = 5.;
8   epsilon = 1.E-2;
9   alpha = 1.E-1;
10  grad = 1;
11
12  // initializations
13  xcur = x0;
14  ycur = f(xcur);
15  xlast = xcur;
16  ylast = ycur;
17
18  // optimization loop
19  while ~(abs(grad) < epsilon)
20    // point to evaluate
21    x = xcur - alpha * grad;
22    y = f(x);
23
24    // saving values
25    xcur = x;
26    ycur = y;
27
28    // updating the gradient
```

```
29    grad = (ycur - ylast) / (xcur - xlast);
30
31    // i++
32    xlast = xcur;
33    ylast = ycur;
34
35    // output
36    printf('x = %f   y = %f\n', xcur, ycur);
37  end
```

Noter que les lignes 17 et 18 ont été ajoutées pour initialiser les variables `xlast` et `ylast` (leur affectation étant passée en fin de boucle).

**Troisième itération** Nous allons maintenant éliminer tous les appels au simulateur en dehors de la boucle (deuxième point noté au paragraphe "But à atteindre"). Il n'y a ici qu'un appel, ligne 16. Pour pouvoir intégrer cet appel dans la boucle, nous introduisons un booléen, `firstiter`, qui permettra de changer le comportement de la boucle pour la première itération :

```
1   // function to optimize
2   function y = f(x)
3     y = x^2
4   endfunction
5
6   // setting parameters
7   x0 = 5.;
8   epsilon = 1.E-2;
9   alpha = 1.E-1;
10  grad = 1;
11
12  // initializations
13  xcur = x0;
14  xlast = xcur;
15  firstiter = %t;
16
17  // optimization loop
18  while ~(abs(grad) < epsilon)
19    // point to evaluate
20    if firstiter
21        x = xcur;
22    else
23        x = xcur - alpha * grad;
24    end
25    y = f(x);
```

```
26
27      // saving values
28      xcur = x;
29      ycur = y;
30
31      // updating variables
32      if firstiter
33          firstiter = %f;
34      else
35          grad = (ycur - ylast) / (xcur - xlast);
36      end
37
38      // i++
39      xlast = xcur;
40      ylast = ycur;
41
42      // output
43      printf('x = %f   y = %f\n', xcur, ycur);
44    end
```

Bien noter que, lors de la première itération, le gradient n'est pas calculé. N'oublions pas d'ajouter la variable `firstiter` à la liste des variables internes de l'optimiseur.

**Quatrième itération** Maintenant, l'algorithme est prêt pour être porté en "ask & tell" :

- le code de la boucle qui est *avant* l'appel au simulateur (lignes 21 – 26) constitue le corps de la méthode `ask`
- le code de la boucle qui est *après* l'appel au simulateur (lignes 29 – 42) constitue le corps la méthode `tell`

Il s'agit donc simplement de couper – coller !

L'optimiseur s'appellera `desc1d` :

```
1   // constructor
2   function this = desc1d(param)
3     this = mlist(['desc1d', 'x0', 'epsilon', 'alpha', 'grad', ...
4                   'xcur', 'ycur', 'xlast', 'ylast', 'firstiter'])
5     for i = this.param, this(i) = param(i), end
6     this.xcur = this.x0
7     this.xlast = this.xcur
8     this.firstiter = %t
9   endfunction
10
11  // method 'stop'
12  function out = %desc1d_stop(this)
```

```
13      out = abs(this.grad) < this.epsilon
14  endfunction
15
16  // method 'ask'
17  function x = %desc1d_ask(this)
18    if this.firstiter
19        x = this.xcur
20    else
21        x = this.xcur - this.alpha * this.grad
22    end
23  endfunction
24
25  //  method 'tell'
26  function this = %desc1d_tell(this, x, y)
27    // saving values
28    this.xcur = x
29    this.ycur = y
30    // updating variables
31    if this.firstiter
32        this.firstiter = %f
33    else
34        this.grad = (this.ycur - this.ylast) / (this.xcur - this.xlast)
35    end
36    // i++
37    this.xlast = this.xcur
38    this.ylast = this.ycur
39  endfunction
40
41  //  method 'best'
42  function [yopt, xopt] = %desc1d_best(this)
43    yopt = this.ycur
44    xopt = this.xcur
45  endfunction
```

Notre programme principal devient alors :

```
46  // function to optimize
47  function y = f(x)
48    y = x^2
49  endfunction
50
51  // initializing the optimizer
52  param = struct();
53  param.x0 = 5.;
```

```
54   param.epsilon = 1.E-2;
55   param.alpha = 1.E-1;
56   param.grad = 1;
57   optimizer = desc1d(param);
58
59   // optimization loop
60   while ~stop(optimizer)
61     x = ask(optimizer);
62     y = f(x);
63     optimizer = tell(optimizer, x, y);
64
65     // output
66     [yopt, xopt] = best(optimizer);
67     printf('x = %f   y = %f\n', xopt, yopt);
68   end
```

Et voilà !

# 9 Utilisation des optimiseurs existants de Scilab

La standardisation des optimiseurs et des simulateurs proposée dans ce document diffère de l'approche "normale" de Scilab. On montre ici comment :

1. utiliser un simulateur "OMD" avec les fonction `optim` et `fsolve`, et
2. utiliser la fonction `optim` comme un optimiseur "OMD" (c'est-à-dire "ask & tell").

## 9.1 Encapsulation d'un simulateur dans une fonction coût

On souhaite optimiser un simulateur avec avec les fonctions d'optimisation standards de Scilab. Ces fonctions ne prennent pas en entrée un simulateur, mais une fonction ; il va donc falloir encapsuler le simulateur dans une fonction.

Considérons l'exemple donné dans [1]

$$\begin{aligned} \min f(x,y,z) &= (x-z)^2 + 3(x+y+z-1)^2 + (x-z+1)^2 \\ g(x,y,z) &= 1 - x - 3y - z^2 = 0 \end{aligned}$$

Le simulateur implémentant ce problème est le suivant :

```
function y = mysim(x)
  y = mlist(['criteria', 'objective', 'grad_objective', ...
             'econstraint', 'grad_econstraint'])
  xs = x(1)+x(2)+x(3)-1
  y.objective = (x(1)-x(3))^2 + 3*xs^2 + (x(1)-x(3)+1)^2
  y.grad_objective = [2*(x(1)-x(3)) + 6*xs + 2*(x(1)-x(3)+1), ...
                      6*xs, ...
```

```
                    -2*(x(1)-x(3)) + 6*xs - 2*(x(1)-x(3)+1)]
  y.econstraint = 1-x(1)-3*x(2)-x(3)^2
  y.grad_econstraint = [-1, -3, -2*x(3)]
endfunction
```

Dans un premier temps, on va optimiser la fonction objectif sans tenir compte de la contrainte égalité. On va utiliser la fonction Scilab `optim`. Pour cela, on va encapsuler l'appel au simulateur dans une fonction renvoyant les valeurs de l'objectif et de son gradient dans le bon format :

```
function [f, g, ind] = costf1(x, ind)
  y = mysim(x)
  f = y.objective
  g = y.grad_objective
endfunction
```

L'optimisation se fait alors sous la forme traditionnelle :

```
-->x0 = [0, 0, 0];

-->[fopt, xopt] = optim(costf1, x0)

 xopt  =

    0.0833333    0.3333333    0.5833333
 fopt  =

    0.5
```

Pour tenir compte de la contrainte égalité, une méthode est de récrire le problème d'optimisation sous la forme d'un problème de recherche de zéro en introduisant un multiplicateur de Lagrange :

$$\nabla f + \lambda \nabla g = 0 \quad (3)$$

$$g = 0 \quad (4)$$

La fonction coût correspondant à ce problème est

```
function z = costf2(x)
  y = mysim(x(1:3))
  z = [y.grad_objective'+x(4)*y.grad_econstraint'; y.econstraint]
endfunction
```

Et la résolution du problème se fait par la fonction `fsolve` :

```
-->x0 = [0, 0, 0, 0];
```

```
-->[x, v] = fsolve(x0, costf2)

 v  =

   1.0E-18 *

  - 0.0650978  - 0.1952935  - 0.0907756    0.
 x  =

    0.1972244    0.1055513    0.6972244    6.510E-20
```

## 9.2 Utilisation de la fonction `optim` en "ask & tell"

Pour découpler la fonction `optim` de la fonction coût, on utilise deux astuces de la fonction `optim` :

1. la fonction coût `costf` peut arrêter brutalement l'optimisation si elle renvoie 0 dans la variable `ind`, et
2. on utilise la condition d'arrêt `'ar'` qui permet de fixer le nombre d'évaluation de la fonction coût ; dans notre cas, deux évaluations.

Avec ces deux astuces, la fonction coût est la suivante :

```
function [f, g, ind] = _costf(x, ind)
  global _lock
  global _y
  if ~_lock then
    f = _y.objective
    g = _y.grad_objective
    _lock = %T
  else
    f = 0
    g = zeros(x)
    ind = 0
  end
endfunction
```

Initialement, le booléen `lock` est fixé à faux. Ainsi, lors du premier appel, la fonction se contente de renvoyer les valeurs de la fitness et du gradient calculé lors du dernier appel au simulateur (et mémorisés dans la variable globale `_y`). Lors du second appel, la fonction interromp l'optimisation (`ind = 0`).

Avec cette fonction coût, l'écriture du simulateur "ask & tell" ne pose aucune difficulté :

```
function this = optimwrap(param)
  this = mlist(['optimwrap', 'nb_max_iter', 'iter', 'x', ...
```

```
                'xopt', 'fopt', 'gopt'])
  this.nb_max_iter = param.nb_max_iter
  this.iter = 0
  this.x = param.x0
endfunction

function x = %optimwrap_ask(this)
  x = this.x
endfunction

function this = %optimwrap_tell(this, x, y)
  this.iter = this.iter + 1
  this.xopt = x
  this.fopt = y.objective
  this.gopt = y.grad_objective
  global _lock
  global _y
  _lock = %F
  _y = y
  [fopt, xopt] = optim(_costf, x', 'ar', 2)
  this.x = xopt'
endfunction

function out = %optimwrap_stop(this)
  out = this.iter >= this.nb_max_iter
endfunction

function [fopt, xopt] = %optimwrap_best(this)
  fopt = this.fopt
  xopt = this.xopt
endfunction
```

En reprenant l'exemple précédent, l'optimisation se fait avec le programme

```
param = struct();
param.x0 = [0, 0, 0];
param.nb_max_iter = 10;
opt = optimwrap(param);

while ~stop(opt)
  x = ask(opt);
  y = mysim(x);
  opt = tell(opt, x, y);
  [fopt, xopt] = best(opt);
```

```
  printf('%i: fopt = %f, xopt = (%f, %f, %f)\n', opt.iter, fopt, xopt);
end
```

et à l'exécution, on obtient l'affichage

```
1: fopt = 4.000000, xopt = (0.000000, 0.000000, 0.000000)
2: fopt = 2.706875, xopt = (0.050000, 0.075000, 0.100000)
3: fopt = 1.747273, xopt = (0.094186, 0.147093, 0.200000)
4: fopt = 1.095484, xopt = (0.130209, 0.215104, 0.300000)
5: fopt = 0.714911, xopt = (0.153615, 0.276808, 0.400000)
6: fopt = 0.548410, xopt = (0.153763, 0.326882, 0.500000)
7: fopt = 0.502820, xopt = (0.085529, 0.342764, 0.600000)
8: fopt = 0.617218, xopt = (0.036385, 0.268192, 0.500000)
9: fopt = 0.515725, xopt = (0.114301, 0.357151, 0.600000)
10: fopt = 0.612185, xopt = (0.014301, 0.268923, 0.523545)
```

Ainsi, on peut surveiller plus facilement le déroulement de l'optimisation (voire avoir un contrôle plus fin).

## Références


[1] S. L. Campbell, J.-P. Chancelier et R. Nikoukhah : *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.