Chapter 14

On Object-Oriented Programming of Optimizers – Examples in Scilab

14.1. Introduction

Optimization algorithms are generally implemented following a functional paradigm that places the optimization method at the top of the hierarchy. Consider for instance the package Scilab [CAM 06], where an objective function `f` is optimized by a single call to a procedure that performs the whole optimization process, without any interference of the user:

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

Remark – Objective functions and constraints are calculated from the outputs of a (programming) function or an external software, both of which we call simulators. To keep the text readable, the words "objective function" and "simulator" are used here as a generalization of objective functions and constraints. The ideas presented also apply to constraints and multiple objectives.

Such a structure turns out to be very restrictive to implement more versatile strategies, such as changing the optimizer or `f` during the optimization, which may be needed, e.g., to couple local and global optimizers (e.g., [HAR 05]) or to replace high fidelity simulations by metamodels (see Chapter 5).

Chapter written by Yann Collette, Nikolaus Hansen, Gilles Pujol, Daniel Salazar Aponte and Rodolphe Le Riche.

In this chapter, we consider the object-oriented paradigm where both the optimizers and the models to be optimized (simulators, metamodels, etc.) are conceived as objects of the same hierarchical level. Such objects are the building blocks assembled by the user to create its own optimization strategies. Some tutorial notes on implementation details will be presented along the chapter. In particular we shall develop two examples of optimizers for deterministic functions, namely the simplex method and the Covariance Matrix Adaptation Evolution Strategy (CMA-ES). The examples are given in the Scilab language, which is a simple, opensource language for scientific calculations [INR 09]. Although Scilab is not, strictly speaking, an object-oriented programming (OOP) language, we will apply Scilab programming patterns which emulate OOP. The last part of the chapter extends the Ask & Tell programming pattern to optimizers for noisy functions.

14.2. Decoupling the simulator from the optimizer

Optimization can be seen as an iterative exploration of a search domain where a performance is assigned to each trial point, the goal being to find a point with the best performance possible. An optimization algorithm controls these moves through the search space from the beginning (initial point(s)) to the end (stopping condition). It is quite natural to implement such an algorithm by encapsulating the optimization loop within which the simulator is called inside a function, such that the simulator is an argument (`f`) of the latter. Scilab conforms to this logic, as was just seen with the `optim` function.

In practice however, such a structure quickly meets its limits. Optimizers are not meant (hence not coded) to handle numerically expensive simulators. What if the simulator crashes ? What if one wishes to take advantage of distributed computing resources shared with other users ? Advanced optimization users also often wish to mix various optimizers (like local and global optimizers, [HAR 05]) and simulators of various fidelities (see Chapter 5).

A programming pattern that decouples the calls to the optimizer(s) from the calls to the simulator(s) is necessary to make optimization programs more compatible with engineering facts. To clarify this statement, let us look at the typical calls sequence made during an optimization procedure which is shown in Figure 14.1: it consists in a series of operations with periodical calls to the simulator. The optimizer is made of the piece of code between two simulations. By casting chart 14.1 in the form shown in Figure 14.2, the "optimizer" and "simulator" *objects* are decoupled and become more clearly visible.

One advantage of this object approach is that it makes possible to describe, in a visible manner, new scenarios. For instance, the flowchart in Figure 14.3 represents an optimization strategy based on a metamodel, which is updated during the optimization

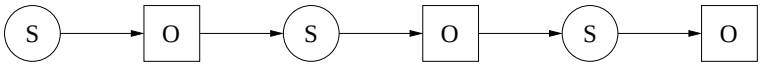


Figure 14.1. *Calls sequence during an optimization procedure.*

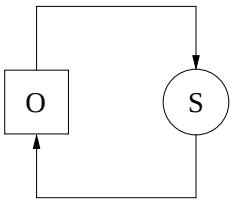


Figure 14.2. *Basic coupling of an optimizer and a simulator.*

after each new simulator call. The underlying modular structure of the overall procedure is clear: the group made by an optimizer and a metamodel yields a new optimizer which is then coupled to a simulator.

A second advantage of the decoupled objects is that the current state of the optimization procedure is stored in a well-defined programming structure, the optimizer object. This makes it possible to save the optimizer in memory and resume the optimization procedure later. The optimization process can also be transferred to another machine. Decoupling simulator and optimizer objects enables distributed optimization procedures.

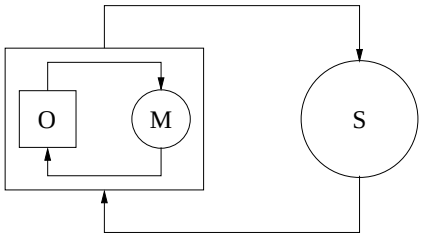


Figure 14.3. *Example of an optimization procedure based on a metamodel. The group made of an optimizer and a metamodel yields a new optimizer which is coupled to a computationally expensive simulator. After each new simulation, the metamodel is updated. The solution point found by optimizing the metamodel is presented to the simulator at each outerloop iteration.*

14.3. The "ask & tell" pattern

As we have seen, the optimizer is the piece of program between two simulation series. We therefore need to interact with the optimizer upstream (*"which simulations are needed?"*) and downstream (*"here are the simulations"*). This is what is defined with the `ask` and `tell` methods. Hence, an optimization scheme should be written:

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

where `opt` and `f` are the optimizer and simulator *objects*. Once the loop is done, the optimum is retrieved by the `best` method:

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

Remark. It is worth noticing the particular syntax of the `tell` method, where the optimizer is present as argument as well as result. It enables optimizer updating, given that Scilab only allows argument passing by value.

Pseudo[1] object-oriented programming is possible in Scilab thanks to typed lists and operator overloading (see the Scilab help entries `mlist` and `overloading` for further details, [INR 09]). An optimizer is therefore an object (an `mlist` list of the specific <`optimizer_type`>) whose associated methods `ask`, `tell` and `best` should be defined according to the following syntax:

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

1. The programming pattern used provides neither strict data encapsulation (inner data can always be changed) nor inheritance. This is the reason why we qualify this programming scheme of "pseudo" object-oriented.

This step *overloads* the functions `ask`, `tell` and `best`. In Scilab, it is possible to overload most of the operators (+, -, `()`, ...) as well as some internal functions (`disp`, `plot2d`, ...). However, in order to overload the `ask`, `tell` and `best` methods, it is necessary to redirect the execution to the methods of the specific optimizer, as is done by the following code:

```
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
```

In addition, it is necessary to implement a *constructor*, i.e. a function in charge of initializing the optimizer. All these points will be illustrated in details in the rest of this chapter.

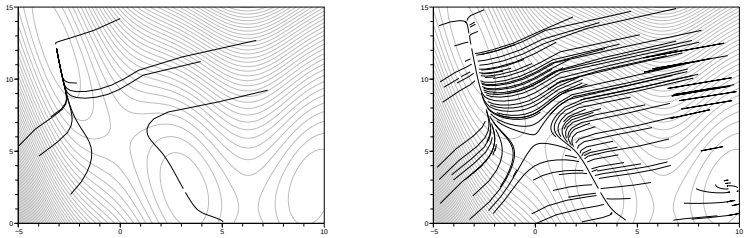
14.4. Example: a "multi-start" strategy

The (pseudo) object-oriented programming pattern provides a first advantage: it allows to imagine complex optimization scenarii. We illustrate this point through a "multi-start" strategy which turns out to be quite complicated to program following the traditional functional paradigm.

A naive multi-start approach consists in performing N successive runs of a gradient-based optimizer from different initial points. The goal is to locate the global optimum, defined as the best of the local optima. Assuming that each run performs n iterations, the multi-start strategy requires $N \times n$ evaluations of the objective function. Figure 14.4(a) illustrates such a strategy applied to the Branin-Hoo function. The following can be noticed :

1) If the optimizer takes less than n simulations to converge, the remaining simulation budget is not used.

2) The trajectories of different optimization runs tend to follow the same paths, i.e., some old trajectories are revisited. This is a waste of simulation budget.



(a) (b)

Figure 14.4. *Multistart optimization strategies. In (a), different runs are performed independently of each other. In (b), the optimization trajectory is interrupted when it meets that of a precedent run; a new run begins in a different region of the domain.*

We now consider the following alternative strategy: the optimizer is stopped if it is close to an already visited trajectory. Optimization runs are sequentially restarted from non-explored regions in the limit of the remaining simulation budget. Using the "ask & tell" optimizer `steepdesc`, that shall be described in paragraph 14.5.3, the multi-start strategy is implemented in a few lines:

```
neval = 1000  // simulation budget
Arch = []     // archive of visited points
i = 0         // optimizer index
// WHILE THE OPTIMIZATION BUDGET IS NOT EXHAUSTED:
while neval > 0
  // INITIALIZE A NEW OPTIMIZER;
  i = i + 1
  opt = steepdesc()
  opt.x0 = grand(1, 2, 'def') .* (xmax - xmin) + xmin
  opt.step = 5E-2
  opt.eps = 1E-2
  // ALLOCATE A FRACTION OF THE BUDGET TO THE OPTIMIZER;
  opt.eval = min(100, neval)
  neval = neval - opt.eval
  // WHILE THE OPTIMIZER DOES NOT STOP;
  while ~ opt.stop
    // ASK THE OPTIMIZER FOR A POINT,
    x = ask(opt)
    // CALCULATE THE DISTANCE OF THE CURRENT
```

```
    // POINT TO THOSE OF THE ARCHIVE,
    Xj = Arch(Arch(:,3) ~= i, 1:2)
    d = min(sqrt(sum((Xj-x(ones(1,size(Xj,1)),:)).^2,'c')))
    // IF THE DISTANCE IS TOO SMALL,
    if d < 0.1 then
      opt.stop = %t // STOP THE OPTIMIZER,
    else
      // RUN THE SIMULATION OTHERWISE,
      [y, dy] = branin(x)
      // UPDATE THE OPTIMIZER,
      opt = tell(opt, x, list(y, dy))
      // AND ADD THE POINT TO THE ARCHIVE;
      Arch($+1,1:3) = [x, i]
    end
  end
  // RESTORE THE BUDGET NOT USED BY THE OPTIMIZER;
  neval = neval + opt.eval
end
```

An example of execution of the above code is presented in Figure 14.4(b). We see how the optimizer better exploits the simulations budget as the domain is much more covered than in fig. 14.4(a).

It is important to note that this strategy would be quite complicated to implement (although it is possible) using the function `optim` of Scilab, because it is not easy to stop the optimizer during the execution of the multi-start scheme. By contrast, we simply did it here with the instruction `opt.stop = %t`.

14.5. Programming an ask & tell optimizer: a tutorial

We have just introduced the ask & tell optimizer pattern and shown how it can be used to build customized optimization strategies. We now bring further details by giving the implementation of three ask & tell optimizers. This section is a programming tutorial. The optimizer examples have not been chosen for their efficiency but for their pedagogical interest. The three examples illustrate the following features:

1) random search: optimizer objects and associated methods;

2) $(\mu/\mu, \lambda)$-ES: starting parameters, optimizer based on a density of points;

3) steepest descent: multiple model responses, optimizers with different states, composite stopping criterion.

14.5.1. *Example 1: Random Search*

The random search is undoubtedly the simplest optimization algorithm: at each iteration a point is randomly generated within the domain, and it is kept if it performs better than any other previously found point. Such an approach is indeed naive, but it illustrates the basic ingredients of stochastic search algorithms such as simulated annealing, evolutionary algorithms, etc. It is also a benchmark algorithm because it has no parameter and, no matter the search space dimension, it converges (although slowly) to the optimum (see e.g. Spall [SPA 03]).

Implementation

We begin by defining a constructor, i.e. a function in charge of initializing the optimizer object:

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

The optimizer returned by this function is a list (`mlist`) of type "`rsearch`", containing several fields, namely `d`, the search space dimension, `xmin` and `xmax`, the lower and upper bounds of the domain respectively, `iter`, the number of remaining iterations, `stop`, a boolean that indicates if the stopping condition was met, `_x` and `_y`, the best point found so far and its objective function value.

The `ask` method fetches a new point from the optimizer. For the random search, it simply consists in generating a random point inside the domain:

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

The `tell` method updates the optimizer with an evaluated search point. We keep this point if it is better than any other already known point:

```
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
```

Note that the variables `iter` and `stop` are simultaneously updated.

Finally, the `best` method reports the best point found:

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

Test

The random search just programmed is tested on the Branin-Hoo function [JON 98] :

$$f(x_1, x_2) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10 \qquad (14.1)$$

This function has three local minima in $[-5, 10] \times [0, 15]$. The following implementation of Branin-Hoo returns the function value as well as the 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
```

Use of the optimizer starts with the initialization of an object of type "`rsearch`" with the chosen parameters:

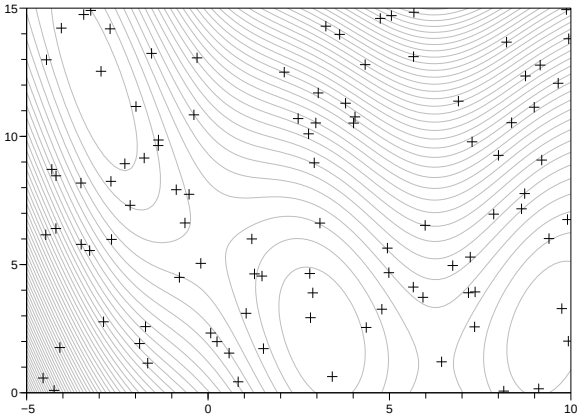


Figure 14.5. *Sample run of the* `rsearch` *optimizer with 100 iterations on the Branin-Hoo function.*

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

Next, the ask & tell optimization loop is straightforward:

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

It yields results such as those shown in Figure 14.5. As expected, the points are uniformly spread in the domain.

Possible improvements

Spall [SPA 03] suggests two improvements for the random search. The first one ("*localized random search*") consists in biasing the point's sampling towards the

neighborhood of the best known point. We recover in this way the notion of optimization trajectory. The second improvement ("*enhanced random search*") adds an average of recent successful steps to the search trajectory. These two improvements are implemented according to the ask & tell logic in the `OMD_toolbox` toolbox [Con 09].

14.5.2. *Example 2:* $(\mu/\mu, \lambda)$ *Evolution Strategy*

The $(\mu/\mu, \lambda)$ evolution strategy (see e.g. [BEY 01]) consists in evolving a density of points from an iteration to the next, according to the following principle: for $i = 1, \ldots, n$,

1) randomly generate λ points of probability density p^i which follows $\mathcal{N}(\bar{x}^i, \sigma^2 \mathrm{I})$:

$$x_1^i, \ldots, x_\lambda^i \sim p^i \tag{14.2}$$

2) select the μ best points ($\mu < \lambda$):

$$x_{1:\lambda}^i, \ldots, x_{\mu:\lambda}^i \tag{14.3}$$

(where $x_{j:\lambda}^i$ denotes the j-th best point)

3) from these points estimate the next iteration search density, p^{i+1} (here we only update the mean):

$$\bar{x}^{i+1} = \frac{1}{\mu} \sum_{j=1}^{\mu} x_{j:\lambda}^i \tag{14.4}$$

Using the evolution strategy jargon, the μ points are called *parents*, whereas the λ points of the next iteration are called *offsprings*.

Implementation

The optimizer parameters are: the `mu` and `lambda` integers, the vector of standard deviations `sigma` and the initial point `x0`. Some optimizer's inner variables join this group, viz. `iter`, the number of remaining iterations, `stop`, the stopping condition, `_X`, the μ parents and finally `_y`, the corresponding performances. Once these parameters are identified, the implementation of the `mulambda` constructor is straightforward:

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

The ask method generates a population centered on the initial point for the first iteration, and on the barycenter of the μ parents for the subsequent iterations:

```
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
```

The tell method selects the μ best points and updates the inner variables of the optimizer:

```
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
```

Finally, the best method retrieves the best point in the parent population (i.e., the best point out of the already selected μ points ...) :

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

Test

The mulambda optimizer is also tested on the Branin-Hoo function. The initialization of the mulambda object is:

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

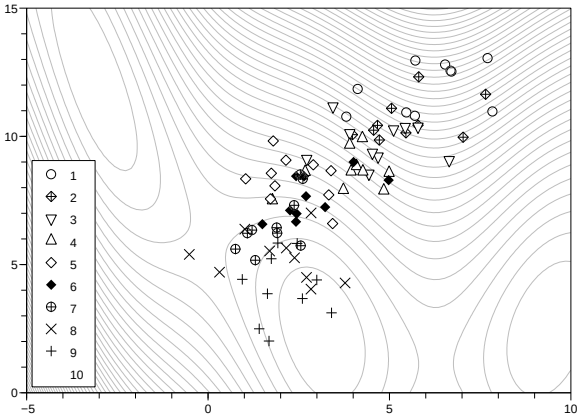


Figure 14.6. *Example of a* `mulambda` *optimizer (evolution strategy* $(\mu/\mu, \lambda)$*) convergence on the Branin-Hoo function with* $\mu = 2$ *parents,* $\lambda = 10$ *offsprings and 10 iterations (i.e., 100 evaluations in total).*

The ask & tell optimization loop slightly differs from the preceeding case as it manages a set of points, which induces a new `for` loop:

```
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
```

A sample `mulambda` run is shown in Figure 14.6. The method seems to converge to a local optimum.

Possible improvements

The standard deviation, σ, can be adapted to improve convergence. The rule of thumb is the so-called "one-fifth-rule" that aims at tuning the standard deviation σ for keeping the improvement probability around 20%. In the $(\mu/\mu + \lambda)$-ES variant, the μ

parents are kept until the next `tell` call, where the selection of the μ parents of the next generation is made out of $\mu + \lambda$ points. Hence one never risks to lose the best point, at the cost of a higher risk to converge prematurely to a location different from the global optimum. These two variations are implemented according to the ask & tell logic inside the `OMD_toolbox` [Con 09].

14.5.3. *Example 3: Steepest descent*

The steepest descent method consists in descending in the direction opposite to the gradient:

$$x_0 \text{ given} \tag{14.5}$$

$$x_i = x_{i-1} - \alpha_i \nabla f(x_{i-1}) \,, \quad i \geq 1 \tag{14.6}$$

where α_i is the size of the displacement that minimizes the function in this direction:

$$\alpha_i = \underset{\alpha}{\text{argmin}}\, f(x_{i-1} - \alpha \nabla f(x_{i-1})) \tag{14.7}$$

This step is called "*line search*" and is often replaced by simpler heuristics, like using a constant α_i. The variant we implement here consists in reducing α_i because, for arbitrarily small steps and unless we are at a stationary point, it is guaranteed that the algorithm will progress [MIN 86]:

– set α_i to the initial value $\alpha_0 > 0$ (given by the user);

– divide α_i by two ($\alpha_i \leftarrow \alpha_i/2$) while

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

A composite stopping criterion is implemented:
$\text{dist}(x_i, x_{i-1}) \leq \varepsilon$
or maximum number of iterations reached
or maximum number of objective function evaluations reached
Such a condition is necessary because, during the adaptation of α_i (the line search), the number of function evaluations is not known in advance.

Implementation

The constructor initializes an `mlist` of type `steepdesc`:

```
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
```

The parameters are `x0`, the initial point, `step`, the initial value of parameter α_i at the beginning of each iteration (the current value is `_step`), `eps`, the tolerance over the x_i's (written ε in mathematical format), as well as the usual inner variables. Notice that the gradient (`_dy`) is also stored.

The `ask` method returns the initial point `x0` during the first iteration, and later the points given by (14.6):

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

The `tell` method should manage two optimizer states, either the returned point corresponds to a new iteration or to an adaptation step of 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
```

Within this function, `y` is assumed to be a list with the objective function value in the first field and the gradient in the second field.

The `best` method returns the current point:

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

Test

The `steepdesc` optimizer is run on the same Branin-Hoo function with the following initial parameters:

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

Note that, as we do not fix the maximum number of the objective function evaluations, `eval`, its default value is $+\infty$ (cf. the constructor implementation).

Besides the handling of the $(f(x_i), \nabla f(x_i))$ couple within a list, the ask & tell optimization loop introduces no novelties:

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

Figure 14.7 shows a `steepdesc` trajectory. Note that, in general, the optimizer converges to local optima.

Possible improvements

The line search of `steepdesc` should be improved with a strategy that allows to take larger steps if necessary, e.g., Goldstein's rule.

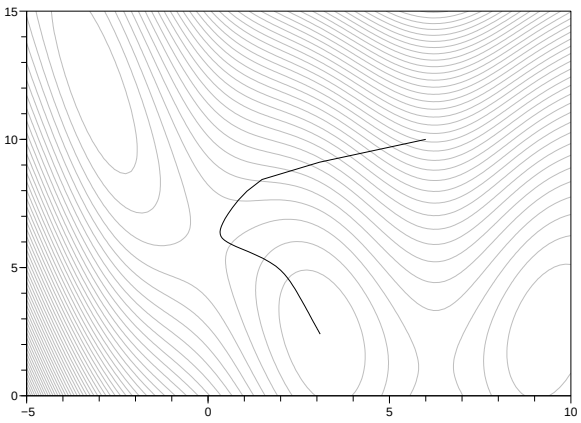


Figure 14.7. *Example of a* `steepdesc` *execution with 100 iterations on the Branin-Hoo function.*

14.6. The Simplex method

14.6.1. *Principle*

The Simplex algorithm, also called Nelder-Mead algorithm [NEL 65], is a derivative-free non-linear optimization algorithm. It should not be mistaken with the simplex method used in linear programming.

The simplex method uses $k+1$ trial points (where k represents the dimension of the decision variable $\overrightarrow{x}$) for defining an improvement direction of the objective function. Such $k+1$ vertices geometrical figure is called a simplex.

14.6.2. *Presentation of the method*

First, the algorithm chooses the $k+1$ values of the decision variables $\overrightarrow{x}$ at random. The objective functions of these points is then evaluated. The simplex will henceforth be updated by deletion of the least efficient vertex and replacement by a new point which is evaluated. The generation of the new point is based on three rules that aim at *i)* creating an improvement direction and *ii)* having a large step size.

The simplex rules are now introduced using the following notation.

– W: worst (rejected) point.

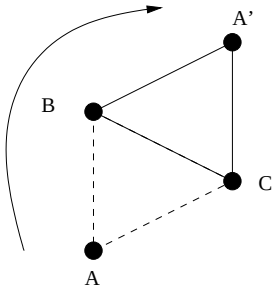


Figure 14.8. *Generation of a new point by symmetry.*

– B: best point.
– N: second best point.
– $\bar{C}$: barycenter of the remaining points.
– α: reflexion coefficient ($\alpha = 1$ by default).
– γ: expansion coefficient ($\gamma = 2$ by default).
– β^+: positive contraction coefficient ($\beta^+ = 0.5$ by default).
– β^-: negative contraction coefficient ($\beta^- = 0.5$ by default).

The simplex rules are:

Rule 1. Reject the worst solution by reflection. A new realization R of the decision variable $\overrightarrow{x}$ is calculated by homothety of the rejected position about the barycenter of the simplex points but the worst (see Figure 14.8).
Reflection: $R = \bar{C} + \alpha \cdot \left(\bar{C} - W\right)$

Rule 2. Expand the displacement towards the objective function improvement zone. This rule permits to increase the step size along favorable search directions.
Expansion: $E = \bar{C} + \gamma \cdot \left(\bar{C} - W\right)$

Rule 3. Contract the displacement if this has been done towards a non favorable zone. This rule allows not to go back to the previous position and prevents oscillations within the same iteration between two bad points.
Positive contraction: $C_+ = \bar{C} + \beta^+ \cdot \left(\bar{C} - W\right)$
Negative contraction: $C_- = \bar{C} - \beta^- \cdot \left(\bar{C} - W\right)$
Expansions and contractions are sketched in 2D in Figure 14.6.2.

A flow chart of the algorithm, showing how these rules are architectured, is given in Figure 14.9. Figure 14.10 illustrates on a simple 2D landscape how the simplex may move and, on the average of several iterations, find the right search direction.

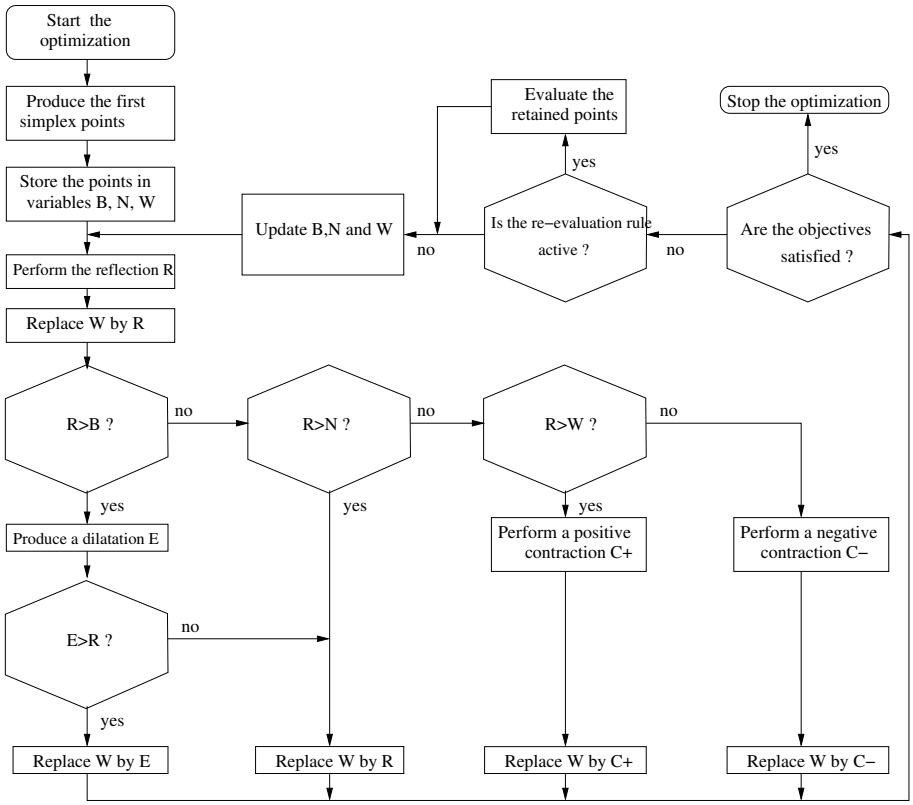


Figure 14.9. *Flow chart of the simplex method algorithm.*

14.6.3. *Implementation*

Our Ask & Tell implementation of the Nelder and Mead algorithm uses a "step by step" version of the simplex algorithm. This method has the following prototype in Scilab:

```
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)
```

The different parameters of the optimization method have the following meaning:

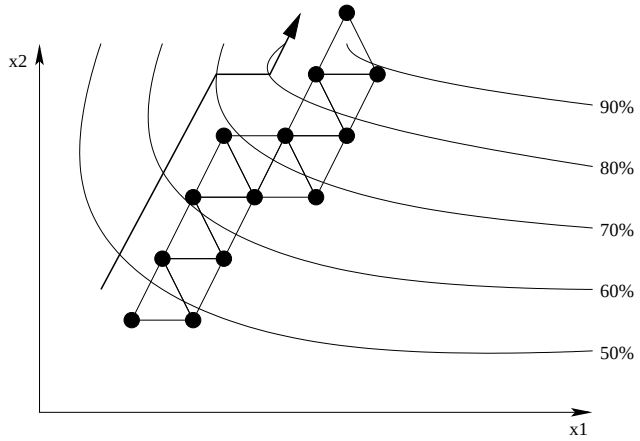


Figure 14.10. *Path of a simplified simplex method (only reflections are made). The algorithm correctly finds an uphill direction (the function is maximized here).*

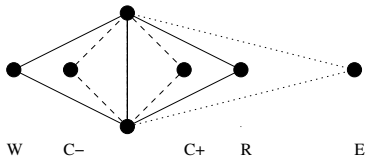


Figure 14.11. *Expansion and contraction steps of the simplex method.*

– `f_current`: objective value of x_current. If nm_mode='init', then f_current is a vector of $n+1$ components corresponding to the coordinates of the simplex;

– `x_current`: the initial simplex ($n+1$ columns vectors) or the last created vertex whose objective function has been calculated (1 column vector), depending on the value of nm_mode;

– `data_current`: current state of the optimization method. This parameter should be void at the first iteration (nm_mode='init');

– `nm_mode`: operation mode of the Nelder & Mead "step by step" method. Admissible values are:
- `'init'`: for the initial iteration;
- `'run'`: for all the remaining iterations;
- `'exit'`: for retrieving the best solution.

– `Log`: a boolean. If it is %T, the information is displayed during the optimization run (this is an optional parameter set to %F by default);

– kelley_restart: a boolean. If it is %T, the simplex is re-initialized around the current best point when the simplex aspect ratio reaches a maximum value (this is an optional parameter set to %F by default);

– kelley_alpha: a scalar related to the shape of the current simplex. It controls the maximum amount of simplex degeneracy allowed during the optimization (this is an optional parameter set to 1e-4 by default);

– x_next: a parameter vector for which the values of the objective function should be calculated or the best solution already found by the Nelder & Mead method (if nm_mode equals 'init', x_next corresponds to a $n + 1$ columns vector);

– data_next: structure containing the state of the Nelder & Mead method. This structure should be transmitted to the function at each iteration;

– eval_Func: the number of evaluations of the objective function (this parameter is optional);

– f_hist: the best objective function value at each iteration (this parameter is optional);

– x_hist: the points making the simplex at each iteration ($n + 1$ column vector, optional).

We will use now this function to define a Nelder & Mead algorithm complying with the Ask & Tell formalism.

The relevant parameters of the algorithm are:

– ItMX: maximum number of iterations;

– x0: starting point of the optimization (the initial simplex will be instantiated around x0);

– upper and lower: the upper and lower bounds of the optimization variables;

– simplex_relsize: the relative size of the simplex that will be defined around x0;

The constructor of the nmomd method is the following:

```
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
```

We now overload the ask and tell functions.
At the first iteration, the ask function generates the initial simplex. Later, the value `x_init` is the variables vector for which the objective function should be calculated.

```
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
```

The `tell` function updates the parameter vector `x_init` with the objective function values just calculated. It then updates the internal structure of the `nmomd` object, i.e., the number of remaining iterations is decremented, the stopping boolean and the method state are updated.

```
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
```

Finally, a method for retrieving the best parameter vector found as well as the corresponding objective function value is defined.

```
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
```

14.6.4. *Example*

We will now apply the Ask & Tell version of the Nelder & Mead method to the Branin-Hoo function (see paragraph 14.5.1). We start by initializing the parameters of the method:

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

The search domain is also defined:

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

An initial point is randomly selected from the search domain:

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

And the optimization method is started:

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

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

We recover the best parameter vector and the corresponding objective function values:

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

An example run of this program is shown in Figure 14.12.

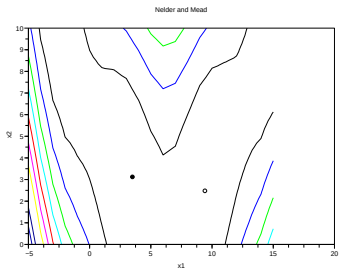
14.7. Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [HAN 01, HAN 03, HAN 04, HAN 06] (used in Chapter 12) is a state-of-the-art stochastic search method.

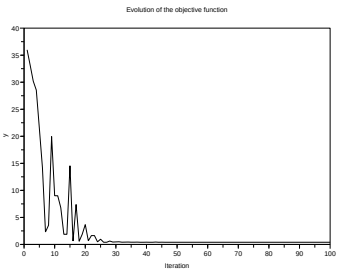
14.7.1. *Principle*

New candidate solutions, $\vec{x}_i$, are sampled according to a *multi-normal* distribution of mean $\vec{m}$, and covariance matrix the product of $\underline{\underline{C}}$ with the square of a step size σ,

$$\vec{x}_i = \vec{m} + \sigma \times \mathcal{N}_i\left(\vec{0}, \underline{\underline{C}}\right) \qquad \text{for } i = 1, \ldots, \lambda \ . \tag{14.8}$$



(a) Initial (dark) and final point (bright).



(b) Evolution of the best values of the objective functions vs. the number of iterations.

Figure 14.12. *Example of a run of the Ask & Tell version of the Nelder & Mead algorithm on Branin's function.*

The λ new points created using Eq. 14.8 follow $\mathcal{N}\left(\vec{m}, \sigma^2 \underline{\underline{C}}\right)$. Typically, λ is of the order of 10 although, in some cases, larger values ($\lambda \gg 100$) may be advisable. Besides the starting point and the stopping criterion, the population size $\lambda > 3$ is the main parameter of the algorithm.

In the $(\mu/\mu, \lambda)$-CMA-ES, the best $\mu < \lambda$ newly generated solutions are selected to update the mean $\vec{m}$ and the other distribution parameters σ and $\underline{\underline{C}}$. These updates rely on two principles.

– The time evolution of the mean solution $\vec{m}$ in the search space is analyzed. The covered distance determines changes of the step-size σ. The direction shapes the covariance matrix $\underline{\underline{C}}$. The time evolution is captured in a so-called evolution path and is computed in a momentum equation.

– The distribution shape, that is the covariance matrix $\underline{\underline{C}}$, is updated such that *successful steps* become more likely to be sampled again. All updates are unbiased, given random selection.

The $(\mu/\mu, \lambda)$-CMA-ES turns out to be a robust and efficient stochastic search algorithm.

– Sampling $\lambda \gg 1$ new solutions and the $(\mu/\mu, \lambda)$ selection are robust to noisy evaluations and help not getting trapped into local minima. Increasing the population size λ can further improve the capacity of locating global optima [HAN 04].

– The adaptation of the covariance matrix is particularly helpful on ill-conditioned functions, while the performance on well-conditioned functions is not affected. As a

drawback, the internal computational complexity is in n^2 (for the sampling of a general multi-variate normal distribution), where n is the search space dimension.

– The step-size control allows a fast increase of the sample variance, which is essential to prevent premature convergence, and later in the search a fast decrease to accurately converge to the optimum.

14.7.2. *Implementation and Interfaces*

The implementation follows very closely the descriptions given in [HAN 04, HAN 06, HAN 09]. We will not detail the basic `%cma_ask`, `%cma_tell` and `%cma_best` functions, which have been sufficiently illustrated in the preceeding examples. Other important functions are now briefly discussed.

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

In the first case, `cma_new` returns its parameters. In the second case it returns a new optimizer, assigned to `es`. At least two mandatory parameters in `param` need to be set by the user, as will be seen below. A full `param struct` is returned which contains the actually used parameters.

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

allows to plot optimization runs. Data visualization can be very insightful and, in many cases, helps to improve the problem formulation.

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

provide a functional interface to the object-oriented implementation, similar to the built-in Scilab function `optim`. `cma_optim_restarted` implements restarts, where λ is increased for each restart by a factor of two [AUG 05].

14.7.3. *Examples*

In this section we discuss a few examples from scratch. A cma-object is generated with the function `cma_new(param)`. The `struct param` has two mandatory fields, `x0` and `sigma0`. First, we get a parameter `struct` filled with default values:

```
-->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]
```

The returned parameter struct contains a few short comments in the `readme` field explaining also the role of `x0` and `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"
```

Parameters `x0` and `sigma0` do not have default values. All other optional parameters have default values and do not need to be part of the input `struct` for `cma_new`. We continue with a short example minimizing the 8-D Rosenbrock function. The initialization reads

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

`sigma0` was deliberately chosen too small. It should be such that the optimum is expected to be within `x0` ± `3*sigma0`, in our case `sigma0` = `0.5` would be appropriate. If different values are appropriate for different coordinates, the optional parameter `opt.scaling_of_variables` can be used.

We define the Rosenbrock function[2]

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

The following optimization iterations resembles previous examples given in this chapter.

```
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);
```

Here, the function `ask` delivers *a list* of solution vectors (since Version 0.99), where solutions are *column vectors*. The output from the example looks as follows.

```
(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
```

The "Axis Ratio" (5-th column) is the square root of the condition number of the covariance matrix $\underline{\underline{C}}$. Large values indicate an ill-conditioned problem. Observing a final condition number

2. For demonstration purposes, we have subtracted 10^{-5} from the original function value such that solutions close to the optimum yield negative values. The CMA-ES is invariant to adding a constant to the function value and has other important invariance properties, see [HAN 01, HAN 06].

of about $60^2 \approx 4 \times 10^3$, the Rosenbrock function appears to be moderately ill-conditioned. Condition numbers up to 10^7 are not uncommon in practice. The numerics involved allow to handle condition numbers up to 10^{14}.

Useful output can be found in the `es.out struct`.

```
-->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
```

The final mean solution is delivered in `es.out.solutions.mean.x`. The termination reasons were small deviations in the function values:

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

Figure 14.13 shows the output graphics that was generated while running the example. The plot can also be generated by calling

```
-->cma_plot
```

after the run has finished. `cma_plot` reads data from files written in `tell()`. In the lower left subplot of Figure 14.13, we find that the ill-conditioning of the Rosenbrock function is related to one long axis in the distribution (one large eigenvalue). This means that one search direction yields much quicker progress than any other direction. In the upper right, the evolution of the parameters reveals the structure of the Rosenbrock function.

To control the frequency of output figures plotting and files writing, the "modulo" options can be changed, `plotmodulo` or `logmodulo` respectively.

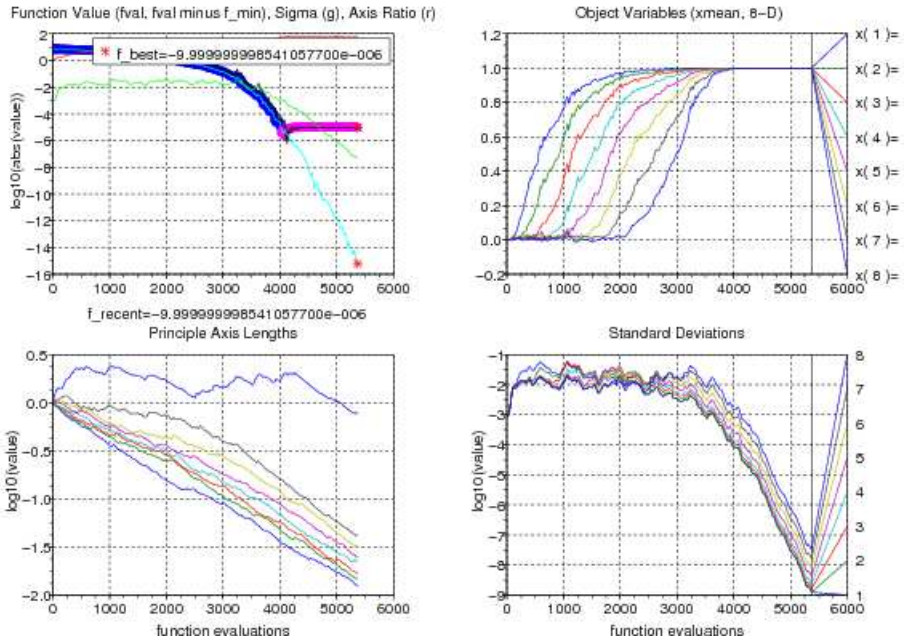


Figure 14.13. *Output of* `cma_plot` *from an optimization run on the 8-D Rosenbrock function, where* 10^{-5} *was subtracted from the original function value.* ***Upper left****: The function value (thick line in blue and magenta) becomes negative after about* 4000 *function evaluations (*400 *iterations). The cyan line (final value* 10^{-15}*) shows the difference to the best achieved function value and reveals a continuing improvement, as the line is decreasing until termination.* ***Lower left****: The principle axes lengths of the sample distribution (in log-scale) adapt to the minimized function. They reveal the (local) structure of the underlying optimization problem.* ***Upper right****: the variables move from zero to one in an ordered fashion. The global optimum is at* $x_i = 1$ *for all* $i = 1, \ldots, 8$*.* ***Lower right****: the standard deviations quickly increase during the first iterations by about one order of magnitude from* 10^{-3} *to* 10^{-2}*. They also vary during the optimization by a factor of about three and quickly decrease during the final convergence phase after* 4000 *function evaluations.*

```
-->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
```

For running the loop quietly, `logmodulo` and `displaymodulo` would be set to zero, limiting the output information to a minimum.

14.8. Ask & Tell formalism for uncertainty handling

14.8.1. *The additional data created by simulation uncertainties*

The underlying assumption of the optimization methods studied so far, and any other general optimization algorithm, is that it is possible to assign a stationary performance value to any decision vector, i.e. if we say $y = f(x)$ then y is a unique vector of scalars. Within such a *deterministic* formulation, the optimization becomes the search for the vector x which has the best performance vector y. However, in industrial contexts and in real life in general, such a deterministic performance measure is rarely possible due to the existence of ill-known factors (prices, boundary conditions, material state, demand, ...): physical systems have an inherent variability and there are practical and theoretical limitations to collect and handle all the necessary information to describe a system. The practical consequence is that the decision vectors are no longer associated to single objective values but to sets of possible performances.

14.8.1.0.1. The simple case: removal of uncertainties by a statistical transformation.

A first step consists in transforming the original single or multi-objective problem into another one, based on some pertinent criteria for uncertainty handling (typically robustness, reliability or risk) and the characteristics of the uncertainty sources [Sal 09]. Some examples of this approach can be found in Chapters 9, 10 and 11 in this book.

Coming back to the basic Ask & Tell formalism introduced in section 14.2, we notice that, in the presence of uncertainty, every decision vector x generates a set of objective instances. If the criteria for uncertainty handling allow to reduce such sets to single values, e.g. some random variable central moments, such values can be sent directly to the `tell` procedure. In such a case we get:

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

where r is a function that assesses the new criteria and returns a vector of scalars z upon which the quality order is built inside tell. Noticing that z=r(f(x)), the above optimization loop is, in terms of programming, equivalent to the deterministic Ask & Tell pattern discussed earlier in this Chapter.

In the context of Monte Carlo simulations for example, y is typically a set of samples of the performance distribution and r is the value of a statistical estimator.

Notice that we have not yet specified whether the uncertainty is attached to x or to $f(x)$ or both. We are only assuming at this stage that the simulator f(x) is able to return a set. For instance, if $y = f(x)$ is a random variable whose probability distribution function (PDF) may be fully known for every x, the optimization problem entails a sequence of pairwise comparisons of PDFs, which can be transformed to mean value comparisons if the adopted criterion, programmed in r, establishes so. Function r should then calculate the mean, μ = z = r(y), leaving the rest of the processing (store x and z, make decisions based on them) to tell. If the resulting PDF is not fully known, which is the general case, μ could be estimated via $\bar{y} = \frac{1}{n}\sum_i^n y_i$.

However, it turns out that since $\bar{y}$ is also a random variable whose value depends on the available samples y_i, the use of the transformation r is somewhat naive.

14.8.1.0.2. Example of Monte Carlo simulations.

We now illustrate how uncertainties change the optimization program by giving an example of Monte Carlo simulations embedded into a Scilab code. Different scenarios of uncertainty handling may occur, namely *a)* uncertainty attached to the decision vectors x, *b)* noisy objective functions and *c)* the combination of the previous cases. We can build a simulator for uncertainty handling with a generic interface F(x,u) based on a deterministic simulator f(x), considering for the case *a)* f(x+ux), for the case *b)* f(x)+uf and finally for the case *c)* f(x+ux)+uf, where u is a vector of realizations of random variables, which is conveniently split into two subvectors ux and uf for simulating the uncertainty associated to x and to f, respectively. As an example, consider the Branin-Hoo function introduced in section 14.5.1. We are going to optimize $f(x_1, x_2)$ adding uncertainty to its argument and outcome, so that it becomes $F_u(x_1, x_2, u) = f(x_1 + u_{x_1}, x_2 + u_{x_2}) + u_f$. Vector $u = (u_{x_1}, u_{x_2}, u_f)$ is a random vector such that $u_{x_1} \sim U(-1, 1)$, $u_{x_2} \sim U(-1, 1)$ and $u_f \sim N(0, 1)$. If we want to perform one iteration of minimization of the mean of the Branin-Hoo function with the naive deterministic-like approach, we write

```
x = ask(opt)
u = [grand(nmc,1,'unf',-1,1), grand(nmc,1,'unf',-1,1),...
     grand(nmc,1,'nor',0,1)]
y = mean( branin(x(1)*ones(budget,1)+u(:,1),...
          x(2)*ones(budget,1)+u(:,2))+u(:,3) )
opt = tell(opt,x,y)
```

where nmc is the number of Monte Carlo realizations of the random vector u. Notice that branin returns a column vector of nmc entries corresponding to nmc evaluations of the function for a fixed x and nmc realizations of u. The effect of varying nmc on the estimation of $E[F_u]$ is illustrated in Figure 14.14.

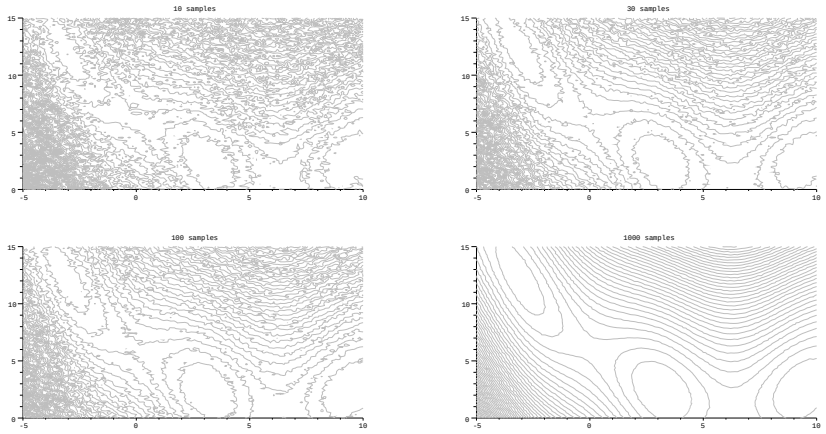


Figure 14.14. *Effect of the number of Monte Carlo simulations on the average estimation of the Branin function with uncertainties. From left to right and top to bottom, 10, 30, 100 and 1000 samples.*

14.8.1.0.3. The general case : ranking of statistical estimators.

Statistical estimations introduce a new level of uncertainty into the problem that has been neglected in the two previous paragraphs. A vector x_1 can be classified as better or worse than another vector x_2 depending on several factors such as the size of the samples of $f(x_1)$ and $f(x_2)$, the type of statistical estimation used in the comparison and the variances of such estimators. Note that many rank-based optimizers (e.g., evolution strategies, the simplex) do not need to know the real values of the different y to work. They only need a procedure to rank the decision vectors well enough to guide the search towards the optimum. When comparing two points in the search space, for example two successive iterates x_1 and x_2, rank-based optimizers do not need to know the values of $f(x_1)$ and $f(x_2)$, they just need to know if $f(x_1) < f(x_2)$.

As an example of ranking under uncertainty, consider the use of statistical hypothesis tests for comparing the outcomes. Let Y_1 and Y_2 denote the random variables associated to the outcomes of $f(x_1)$ and $f(x_1)$, and let $\bar{y_1}$ and $\bar{y_2}$ denote their respective estimated means. In most real situations, the Monte Carlo simulation budget allows to perform only a few simulations, so the $\bar{y}$ estimates are prone to lie significantly far from the real means. In such a case the use of statistical hypothesis tests may represent an additional safety for ensuring good rankings. If we opt for a parametric test, it suffices to estimate the means, $\bar{y_1}$ and $\bar{y_2}$, and the variances of Y_1 and Y_2 using known sample sizes n_1 and n_2, respectively, to draw a conclusion. By contrast, the use of non-parametric tests requires to handle the whole populations comprising n_1 observations of Y_1 and n_2 samples of Y_2 to apply the test. The reader is referred to Chapter 9 for examples of how theses tests can actually apply to optimization.

14.8.2. *An Ask & Tell pattern accounting for simulation uncertainties*

The preceding discussion shows how the Ask & Tell programming template should evolve to account for uncertainties in the optimization formulation: it should be able to handle sets of performance values of different sizes; the ranking of alternatives should allow advanced comparisons beyond the lines

```
if y < this._y then
    this._x = x
    this._y = y
end
```

which were used in the previous examples to update the `tell` function; finally, the `stop` procedure needs to know how many simulations were run during the Monte Carlo simulations to halt the execution loop when the total number of simulations reaches its maximum. Similarly, it could be interesting to let the optimizer control the number of Monte Carlo simulations. In order to illustrate how these features can be integrated into the Ask & Tell formalism, we revisit the random search algorithm described in section 14.5.1 and add the following features:

1) Decoupled ranking: the ranking procedure is performed outside the `tell` procedure by a new function (`rankpop`). `tell` now receives a vector of ranks instead of a vector of objective functions values. In this way, we can account for uncertainties and keep using the rank-based optimizers presented earlier in this chapter without changing their inner structure.

2) Extended parameters passing: for decoupling the ranking of solution points, the `ask` procedure should return the new `x` vectors to be evaluated along with the current `xopt` and its corresponding sample of vector of performance (or use an archive to recover already calculated performances).

3) Control of the number of objective function evaluations: the deterministic formulation of the `ask` and `tell` functions assumes that only one evaluation of the objective functions is performed for every vector. When optimizing with uncertainties, the update of the optimizer should account for a variable number of evaluations for each alternative.

4) Addition of an archive: in order to avoid reevaluating old search points, the optimizer can be assisted by an external memory that stores the points already visited in previous iterations.

Let us now look at the modifications of the `ask` and `tell` functions through the simple example of a random search.

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

```
  this._y = %inf
endfunction

function [x,max_budget] = %rsearchu_ask(this)
  x = (this.xmax - this.xmin) .* grand(1, this.d, 'def') +...
  this.xmin
  // output the new point and the current best one
  x = [x ; this._x]
  max_budget = this.iter;
endfunction

function this = %rsearchu_tell(this, x, y, varargin)
  // sort the new points
  [sorted,k] = gsort(y(:,1),'g','i');
  this._x = x(k(1),:)
  this._y = y(k(1),:)
  // update budget with an optional additional argument
  if(argn(2) > 3) then
    used_budget = varargin(1);
  else
    used_budget = 1;
  end
  this.iter = this.iter - used_budget;
  this.stop = this.stop | this.iter <= 0
endfunction
```

There are mainly two modifications to the `ask` function with respect to the deterministic case. Firstly, in order to allow ranking the alternatives outside `tell`, it is necessary to return the new points generated along with those already stored as best ones (line `x = [x ; this._x]`). Secondly, the optimization loop can be stopped using the number of points to be evaluated, the number of iterations of the loop or the number of evaluations of the objective function. These criteria are not necessarily proportional since the number of Monte Carlo simulations performed for assessing each search point can be variable. Hence, by returning `this.iter`, the programmer has an additional argument available for deciding when to stop the optimization and/or the Monte Carlo simulations.

The `tell` function is completed accordingly with a fourth, optional, parameter, `varargin`, that allows to decrement the total simulation budget by quantities other than 1 (the default). The structure of `tell` assumes that the first column of `y` is the rank of the associated row of `x`, and the other columns allow to store any other optional information (e.g., an estimate of performance value). In this way, all the information related to the already calculated analyzed points can be kept by the optimizer (independently of the archive) along the run.

The functions for overloading `ask` and `tell` need to be modified to comply with the new structure. They become:

```
function [x,max_budget] = ask(this)
```

```
  execstr('[x,max_budget] = %' + typeof(this) + '_ask(this)')
endfunction

function this = tell(this, x, y, varargin)
  sargin = '';
  if (argn(2) > 3) then
    for i = 1 : length(varargin)
      sargin = sargin + ',' + 'varargin(' + string(i) +')';
    end
  end
  execstr('this = %' + typeof(this) +...
  '_tell(this, x, y' + sargin + ')')
endfunction
```

In order to use the new "rsearchu" optimizer, we keep the same parameters as for rsearch except the number of iterations opt.iter which is set to 500 and we add a new parameter, mcs, that indicates the maximal number of Monte Carlo simulations to perform for each point. We also add two vectors xlast and ylast for storing the visited points:

```
opt = rsearch()
opt.d = 2
opt.xmin = [-5, 10]
opt.xmax = [0, 15]
opt.iter = 500
mcs = 50;
xlast = [];
ylast = [];
```

The subsequent "ask & tell" optimization loop is written:

```
while ~opt.stop

  [x,max_budget] = ask(opt);

  // Seach archive for already calculated points
  // if x(i,:) has already been calculated,
  //    output the pendant in xold along with yold.
  // otherwise copy x(i,:) into xnew
  [xnew,xold,yold] = memory_search(x,xlast,ylast).

  s_xnew = size(xnew,1);
  used_budget = 0;
  // Determine how many evaluations can be
  //allocated per vector (row)
```

```
  budget = min(mcs,ceil(max_budget/s_xnew));
  // Create variables for counting the vectors evaluated
  x_evaltd =0;
  // Evaluate by MC simulations
  for i = 1 : s_xnew
    if max_budget - used_budget > 0 then
      u = [grand(budget,1,'unf',-1,1),...
           grand(budget,1,'unf',-1,1),...
           grand(budget,1,'nor',0,1)];
      y = branin(xnew(i,1)*ones(budget,1)+u(:,1),...
          xnew(i,2)*ones(budget,1)+u(:,2))+u(:,3);
      ynew = [ynew ; mean(y)];
      x_evaltd = x_evaltd + 1;
      used_budget = used_budget + budget;
    end
  end

  // Update memory and optimizer
  xnew = xnew(1:x_evaltd,:);
  xlast = [xlast ; xnew];
  ylast = [ylast ; ynew];
  ycurrent = [yold;ynew];
  xcurrent = [xold;xnew];
  ranks = rankpop(ycurrent);
  opt = tell(opt, xcurrent, [ranks,ycurrent], used_budget);

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

An example of the outcome of this program is given in Figure 14.15. To keep the example simple, only the estimated mean of the Monte Carlo samples is stored in the archive. Other characteristics of the Monte Carlo samples could be stored (the number of simulations performed, the variance, ... , and even all the y's of the Monte Carlo simulations, at a high memory cost). In some cases, it could be interesting to complete the evaluation of the current best with new samples, for instance if it has been the best for a long time, to rule out biased samples.

The code of the `rankpop` function is not provided. An example of non trivial ranking can be found in Chapter 9, where the ranking is based on hypothesis testing.

14.9. Concluding remarks

This Chapter has presented a programming pattern for optimization algorithms. It is called "ask & tell" after the name of the two main pattern functions. Examples of optimization methods (random search, simplex, evolution strategies, CMA-ES) written in Scilab in such an object-oriented fashion have been given. Although it may seem as if ask & tell makes programming more complex, it enables

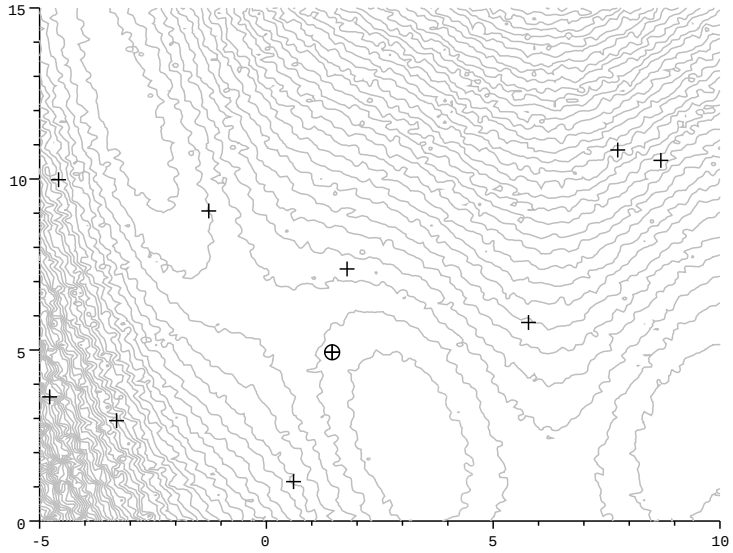


Figure 14.15. *Example run of the Ask & Tell algorithm on an uncertain version of the Branin-Hoo function, 50 Monte Carlo simulations per point. The best point found after testing 10 points is enclosed within a circle. The contour lines were also calculated with 50 Monte Carlo simulation.*

– more versatile optimizers, assembled from several basic optimizers (e.g., the multistart strategy of Section 14.4),

– decoupling the calls to the optimization algorithm from the calls to the simulation which, in turns, allows multi-fidelity strategies and distributed computing,

– coupling the optimization and statistical estimations strategies for handling noisy functions without changing the optimization algorithms.

14.10. Bibliography

[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.

[Con 09] Consortium OMD, "OMD Toolbox for Scilab", www.scilab.org, (see contributions), 2009.

[HAN 01] Hansen N., Ostermeier A., "Completely derandomized self-adaptation in evolution strategies", *Evolutionary Computation*, vol. 9, num. 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, num. 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.

[HAR 05] Hart W. E., Krasnogor N., Smith J., "Recent Advances in Memetic Algorithms", *Series: Studies in Fuzziness and Soft Computing*, vol. 166, Springer, 2005.

[INR 09] INRIA, Digiteo, ENPC, "Scilab : a free software for scientific computing", www.scilab.org, 2009.

[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.

[MIN 86] Minoux M., *Mathematical programming: Theory and Algorithms*, Wiley, 1986.

[NEL 65] Nelder J. A., Mead R., "A Simplex method for function minimization", *Computer Journal*, vol. 7, p. 308-313, 1965.

[Sal 09] Salazar Aponte D. E., Rocco S. C. M., Galván B., "On Uncertainty and Robustness in Evolutionary Optimization-based MCDM", Ehrgott M., Fonseca C., Gandibleux X., Hao J.-K., Sevaux M., Eds., *Evolutionary Multi-Criterion Optimization. Proceedings of the 5th International Conference, EMO 2009*, vol. 5467 of *Lectures Notes in Computer Science*, p. 51–65, Springer, 2009.

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