Nelder-Mead User's Manual

Michaël BAUDIN

Abstract

In this document, we present the Nelder-Mead component provided in Scilab. The introduction gives a brief overview of the optimization features of the component and present an introductory example. Then we present some theory associated with the simplex, a geometric concept which is central in the Nelder-Mead algorithm. We present several method to compute an initial simplex. Then we present Spendley's et al. fixed shape unconstrained optimization algorithm. Several numerical experiments are provided, which shows how this algorithm performs on well-scaled and badly scaled quadratics. In the final section, we present the Nelder-Mead variable shape unconstrained optimization algorithm. Several numerical experiments are presented, where some of these are counter examples, that is cases where the algorithms fails to converge on a stationnary point. In the appendix of this document, the interested reader will find a bibliography of simplex-based algorithms, along with an analysis of the various implementations which are available in several programming languages.

Version 0.4
October 2009

Contents

Notations

n	number of variables
$\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$	the unknown
$\mathbf{x}_0 \in \mathbb{R}^n$	the initial guess
$\mathbf{v} \in \mathbb{R}^n$	a vertex
$S = \{\mathbf{v}_i\}_{i=1,m}$	a complex, where $m \geq n+1$ is the number of vertices
$S = \{\mathbf{v}_i\}_{i=1,n+1}$	a simplex (with $n+1$ vertices)
$(\mathbf{v}_i)_j$	the j-th component of the i-th vertex
S_0	the initial simplex
S_k	the simplex at iteration k
$\mathbf{v}_i^{(k)}$	the vertex i at iteration k
$f_i^k = f\left(\mathbf{v}_i^(k)\right)$	the function value of the vertex i at iteration k
$f : \mathbb{R}^n \to \mathbb{R}$	the cost function

Fig. 1 : Notations used in this document

Chapter 1

Introduction

In this introductory chapter, we make an overview of simplex-based algorithms. We present the main features of the *neldermead* component, and show how to use the component with a simple example.

1.1 Overview

The Nelder-Mead simplex algorithm [29], published in 1965, is an enormously popular search method for multidimensional unconstrained optimization. The Nelder-Mead algorithm should not be confused with the (probably) more famous simplex algorithm of Dantzig for linear programming. The Nelder-Mead algorithm is especially popular in the fields of chemistry, chemical engineering, and medicine. Two measures of the ubiquity of the Nelder-Mead algorithm are that it appears in the best-selling handbook Numerical Recipes and in Matlab. In [46], Virginia Torczon writes: "Margaret Wright has stated that over fifty percent of the calls received by the support group for the NAG software library concerned the version of the Nelder-Mead simplex algorithm to be found in that library". No derivative of the cost function is required, which makes the algorithm interesting for noisy problems.

The Nelder-Mead algorithm falls in the more general class of direct search algorithms. These methods use values of f taken from a set of sample points and use that information to continue the sampling. The Nelder-Mead algorithm maintains a simplex which are approximations of an optimal point. The vertices are sorted according to the objective function values. The algorithm attemps to replace the worst vertex with a new point, which depends on the worst point and the centre of the best vertices.

The goal of this toolbox is to provide a Nelder-Mead (1965) direct search optimization method to solve the following unconstrained optimization problem

$$\min f(\mathbf{x}) \tag{1.1}$$

where $\mathbf{x} \in \mathbb{R}^n$, n is the number of optimization parameters and f is the objective function $f : \mathbb{R}^n \to \mathbb{R}$. In order to solve the unconstrained optimization problem, the Nelder-Mead algorithm uses a

variable shape simplex. The toolbox also provide Spendley, Hext and Himsworth's algorithm [43] (1962), which uses a fixed shape simplex. Historically, the algorithm created by Nelder and Mead was designed as an improvement on Spendley's et al. algorithm. The Box complex algorithm [4] (1965), which is an extension of Spendley's et al. algorithm, solves the following constrained problem

$$\min f(\mathbf{x}) \tag{1.2}$$

$$\ell_i \leq x_i \leq u_i, \qquad i = 1, n \tag{1.3}$$

$$g_j(\mathbf{x}) \geq 0, \qquad j = 1, m \tag{1.4}$$

$$\tag{1.5}$$

where m is the number of nonlinear, positive constraints and $\ell_i, u_i \in \mathbb{R}^n$ are the lower and upper bounds of the variables.

The Nelder-Mead algorithm may be used in the following optimization context :

- there is no need to provide the derivatives of the objective function,
- the number of parameters is small (up to 10-20),
- there are bounds and/or non linear constraints.

The internal design of the system is based on the following components.

- The "neldermead" component provides various simplex-based algorithms and manages for Nelder-Mead specific settings, such as the method to compute the initial simplex and the specific termination criteria.
- The "fminsearch" component provides a Scilab commands which aims at behaving as Matlab's fminsearch. Specific terminations criteria, initial simplex and auxiliary settings are automatically configured so that the behavior of Matlab's fminsearch is exactly reproduced.
- The "optimset" and "optimget" components provide Scilab commands to emulate their Matlab counterparts.
- The "nmplot" component provides features to produce directly output pictures for Nelder-Mead algorithm.

The current toolbox is based on (and therefore requires) the following components.

- The "optimbase" component provides an abstract class for a general optimization component, including the number of variables, the minimum and maximum bounds, the number of non linear inequality constraints, the logging system, various termination criteria, the cost function, etc...

- The "optimsimplex" component provides a class to manage a simplex made of an arbitrary number of vertices, including the computation of a simplex by various methods (axes, regular, Pfeffer's, randomized bounds), the computation of the size by various methods (diameter, sigma +, sigma-, etc...) and many algorithms to perform reflections and shrinkages.

The following is a list of features the Nelder-Mead algorithm currently provides :

- manage various simplex initializations
  - initial simplex given by user,
  - initial simplex computed with a length and along the coordinate axes,
  - initial regular simplex computed with Spendley et al. formula
  - initial simplex computed by a small perturbation around the initial guess point
- manage cost function
  - optionnal additionnal argument
  - direct communication of the task to perform : cost function or inequality constraints
- manage various termination criteria
  - maximum number of iterations,
  - tolerance on function value (relative or absolute),
  - tolerance on x (relative or absolute),
  - tolerance on standard deviation of function value (original termination criteria in [3]),
  - maximum number of evaluations of cost function,
  - absolute or relative simplex size,
- manage the history of the convergence, including :
  - the history of function values,
  - the history of optimum point,
  - the history of simplices,
  - the history of termination criterias,
- provide a plot command which allows to graphically see the history of the simplices toward the optimum,
- provide query functions for
  - the status of the optimization process,

  - the number of iterations,
  - the number of function evaluations,
  - the status of execution,
  - the function value at initial point,
  - the function value at optimal point,
  - etc...

- Spendley et al. fixed shaped algorithm,
- Kelley restart based on simplex gradient,
- O'Neill restart based on factorial search around optimum,
- Box-like method managing bounds and nonlinear inequality constraints based on arbitrary number of vertices in the simplex.

1.2 How to use the Toolbox

The design of the toolbox is based on the creation of a new token by the *neldermead_new* function. The Nelder-Mead object associated with this token can then be configured with *neldermead_configure* and queried with *neldermead_cget*. For example, the *neldermead_configure* command allows to configure the number of variables, the objective function and the initial guess.

The main command of the toolbox is the *neldermead_search* command, which solves the optimization problem. After an optimization has been performed, the *neldermead_get* command allows to retrieve the optimum $x^\star$, as well as other parameters, such as the number of iterations performed, the number of evaluations of the function, etc...

Once the optimization is finished, the *neldermead_destroy* function deletes the object.

1.3 An example

In the following example, we search the minimum of the 2D Rosenbrock function [39], defined by

$$f(x_1, x_2) = 100(x_2 - x_1)^2 + (1 - x_1)^2 \tag{1.6}$$

The following Scilab script allows to find the solution of the problem. We begin by defining the function *rosenbrock* which computes the Rosenbrock function. The traditionnal initial guess $(-1.2, 1.0)$ is used, which corresponds to the "-x0" key. The initial simplex is computed along the axes with a length equal to 0.1. We want to use the Nelder-Mead algorithm with variable simplex size is used, which corresponds to the "variable" value of the "-method" option. The verbose mode is enabled so that messages are generated during the algorithm. After the optimization is performed, the optimum is retrieved with quiery features.

```
function y = rosenbrock (x)
  y = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
endfunction
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-x0",[-1.2 1.0]');
nm = neldermead_configure(nm,"-simplex0method","axes");
nm = neldermead_configure(nm,"-simplex0length",0.1);
nm = neldermead_configure(nm,"-method","variable");
nm = neldermead_configure(nm,"-verbose",1);
nm = neldermead_configure(nm,"-function",rosenbrock);
nm = neldermead_search(nm);
xopt = neldermead_get(nm,"-xopt")
fopt = neldermead_get(nm,"-fopt")
status = neldermead_get(nm,"-status")
nm = neldermead_destroy(nm);
```

This produces the following output.

```
-->nm = neldermead_search(nm);
Function Evaluation #1 is [24.2] at [-1.2 1]
Function Evaluation #1 is [24.2] at [-1.2 1]
Function Evaluation #2 is [8.82] at [-1.1 1]
Function Evaluation #3 is [16.4] at [-1.2 1.1]
Step #1 : order
=============================================================
Iteration #1 (total = 1)
Function Eval #3
Xopt : -1.1 1
Fopt : 8.820000e+000
DeltaFv : 1.538000e+001
Center : -1.1666667 1.0333333
Size : 1.414214e-001
Vertex #1/3 : fv=8.820000e+000, x=-1.100000e+000 1.000000e+000
Vertex #2/3 : fv=1.640000e+001, x=-1.200000e+000 1.100000e+000
Vertex #3/3 : fv=2.420000e+001, x=-1.200000e+000 1.000000e+000
Reflect
xbar=-1.15 1.05
Function Evaluation #4 is [5.62] at [-1.1 1.1]
xr=[-1.1 1.1], f(xr)=5.620000
Expand
Function Evaluation #5 is [4.428125] at [-1.05 1.15]
xe=-1.05 1.15, f(xe)=4.428125
  > Perform Expansion
Sort
[...]
=============================================================
Iteration #56 (total = 56)
Function Eval #98
Xopt : 0.6537880 0.4402918
Fopt : 1.363828e-001
DeltaFv : 1.309875e-002
Center : 0.6788120 0.4503999
Size : 6.945988e-002
Vertex #1/3 : fv=1.363828e-001, x=6.537880e-001 4.402918e-001
Vertex #2/3 : fv=1.474625e-001, x=7.107987e-001 4.799712e-001
Vertex #3/3 : fv=1.494816e-001, x=6.718493e-001 4.309367e-001
Reflect
xbar=0.6822933 0.4601315
Function Evaluation #99 is [0.1033237] at [0.6927374 0.4893262]
xr=[0.6927374 0.4893262], f(xr)=0.103324
Expand
Function Evaluation #100 is [0.1459740] at [0.7031815 0.5185210]
xe=0.7031815 0.5185210, f(xe)=0.145974
  > Perform reflection
Sort
=============================================================
Iteration #57 (total = 57)
Function Eval #100
Xopt : 0.6927374 0.4893262
Fopt : 1.033237e-001
DeltaFv : 4.413878e-002
Center : 0.6857747 0.4698631
Size : 6.262139e-002
Vertex #1/3 : fv=1.033237e-001, x=6.927374e-001 4.893262e-001
Vertex #2/3 : fv=1.363828e-001, x=6.537880e-001 4.402918e-001
Vertex #3/3 : fv=1.474625e-001, x=7.107987e-001 4.799712e-001
Terminate with status : maxfuneval
-->xopt = neldermead_get(nm,"-xopt")
 xopt  =
```

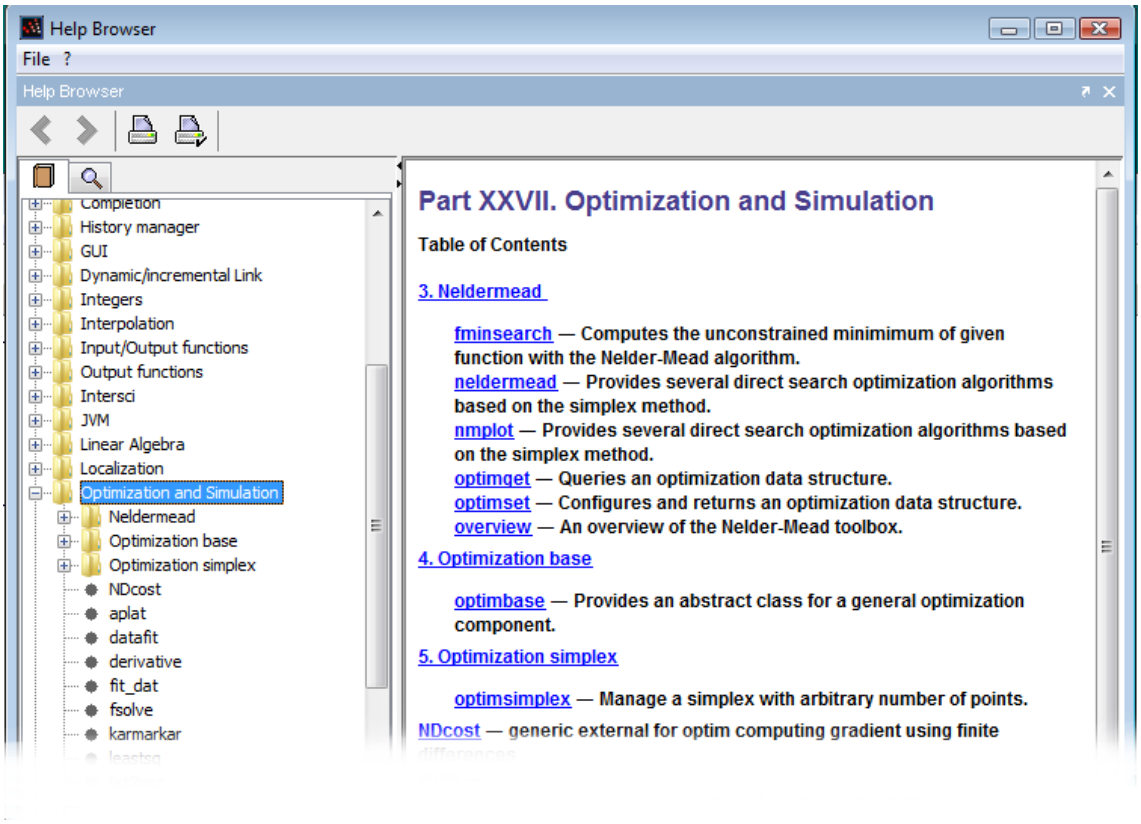

Fig. 1.1 : Built-in help for the Nelder-Mead component

```
    0.6927374
    0.4893262

-->fopt = neldermead_get(nm,"-fopt")
 fopt  =

    0.1033237

-->status = neldermead_get(nm,"-status")
 status  =

 maxfuneval
```

1.4 Help, demonstrations and unit tests

For a complete presentation of the functions and options, the reader should consult the help which is provided with the component. The main menu of the help associated with the optimization module is presented in figures 1.1 and 1.2. The corresponding pages provide a complete documentation for the corresponding functions, as well as many sample uses.

Several demonstrations are provided with the component. These are available from the "Demonstration" menu of the Scilab console and are presented in figure 1.3.

The following script shows where the demonstration scripts are available from the Scilab

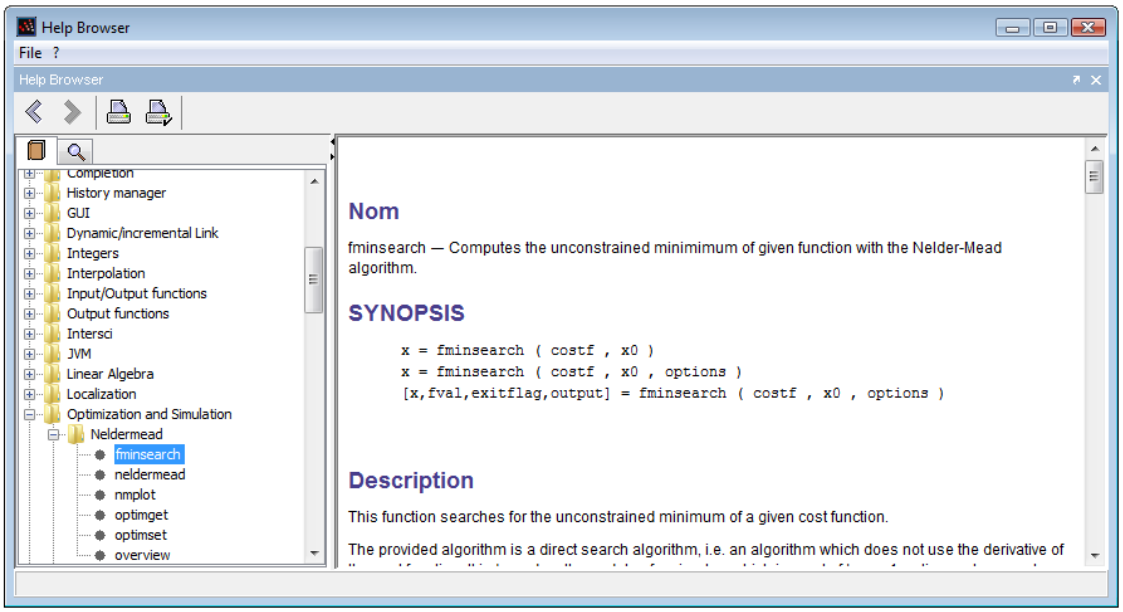

Fig. 1.2 : Built-in help for the *fminsearch* function

installation directory.

```
-->cd SCI/modules/optimization/demos/neldermead
 ans  =

 D:\Programs\SCFD8E~1\modules\optimization\demos\neldermead

-->ls *.sce
 ans  =

!nmplot_rosenbrock.sce          !
!                               !
!nmplot_rosenbrock.fixed.sce    !
!                               !
!nmplot_quadratic.fixed.sce     !
!                               !
!nmplot_mckinnon2.sce           !
!                               !
!nmplot_mckinnon.sce            !
!                               !
!nmplot_han2.sce                !
!                               !
!nmplot_han1.sce                !
!                               !
!nmplot_boxproblemA.sce         !
!                               !
!neldermead_rosenbrock.sce      !
!                               !
!neldermead.dem.sce             !
!                               !
!fminsearch.sce                 !
```

These components were developped based on unit tests, which are provided with Scilab. These unit tests are located in the "SCI/modules/optimization/tests/unit_tests" directory, under the "neldermead", "optimsimplex" and "optimbase" directories. Each unit test correspond to a .tst file. These tests are covering most (if not all) the features provided by the components. This is why there are a good source of information on how to use the functions.

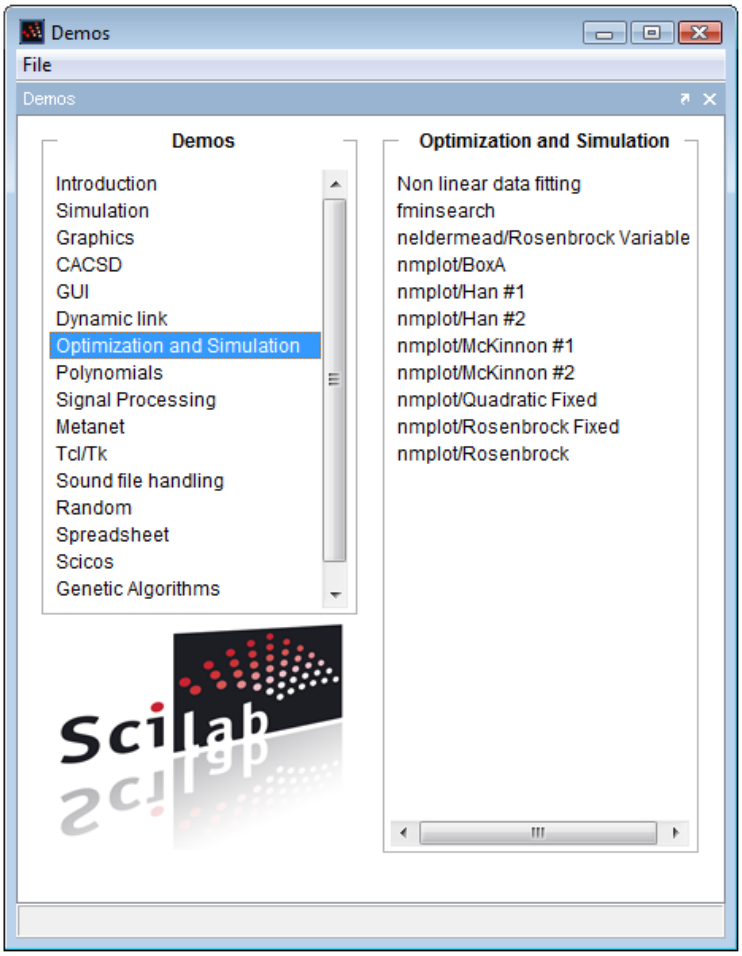

Fig. 1.3 : Built-in demonstration scripts for the Nelder-Mead component

Chapter 2

Simplex theory

In this section, we present the various definitions connected to simplex algorithms. We introduce several methods to measure the size of a simplex, including the oriented length. We present several methods to compute an initial simplex, that is, the regular simplex used by Spendley et al., the axis-by-axis simplex, Pfeffer's simplex and the randomized bounds simplex.

2.1 The simplex

Definition 2.1.1 *(*Simplex*) A* simplex S *in* $\mathbb{R}^n$ *is the convex hull of* $n+1$ *vertices, that is, a simplex* $S = \{\mathbf{v}_i\}_{i=1,n+1}$ *is defined by its* $n+1$ *vertices* $\mathbf{v}_i \in \mathbb{R}^n$ *for* $i = 1, n+1$.

The j-th coordinate of the i-th vertex $\mathbf{v}_i \in \mathbb{R}^n$ is denoted by $(\mathbf{v}_i)_j \in \mathbb{R}$.

Box extended the Nelder-Mead algorithm to handle bound and non linear constraints [4]. To be able to manage difficult cases, he uses a *complex* made of $m \geq n+1$ vertices.

Definition 2.1.2 *(*Complex*) A* complex S *in* $\mathbb{R}^n$ *is a set of* $m \geq n+1$ *vertices, that is, a simplex* $S = \{\mathbf{v}_i\}_{i=1,m}$ *is defined by its* m *vertices* $\mathbf{v}_i \in \mathbb{R}^n$ *for* $i = 1, m$.

In this chapter, we will state clearly when the definition and results can only be applied to a simplex or to a more general a complex.

We assume that we are given a cost function $f : \mathbb{R}^n \to \mathbb{R}$. Each vertex $\mathbf{v}_i$ is associated with a function value

$$f_i = f(\mathbf{v}_i) \text{ for } i = 1, m. \tag{2.1}$$

For any complex, the vertices can be sorted by increasing function values

$$f_1 \leq f_2 \leq \ldots \leq f_n \leq f_m. \tag{2.2}$$

The sorting order is not precisely defined neither in Spendley's et al paper [43] nor in Nelder and Mead's [29]. In [19], the sorting rules are defined precisely to be able to state a theoretical convergence result. In practical implementations, though, the ordering rules have no measurable influence.

2.2 The size of the complex

Several methods are available to compute the size of a complex.

In this section, we use the euclidian norm $\|.\|_2$ the defined by

$$\|\mathbf{v}\|_2 = \sum_{j=1,n} (v_j)^2. \tag{2.3}$$

Definition 2.2.1 *(*Diameter*) The simplex diameter $diam(S)$ is defined by*

$$diam(S) = \max_{i,j=1,m} \|\mathbf{v}_i - \mathbf{v}_j\|_2. \tag{2.4}$$

In practical implementations, computing the diameter requires two nested loops over the vertices of the simplex, i.e. requires m^2 operations. This is why authors generally prefer to use lengths which are less expensive to compute.

Definition 2.2.2 *(*Oriented length*) The two oriented lengths $\sigma_-(S)$ and $\sigma_+(S)$ are defined by*

$$\sigma_+(S) = \max_{i=2,m} \|\mathbf{v}_i - \mathbf{v}_1\|_2 \qquad \text{and} \qquad \sigma_-(S) = \min_{i=2,m} \|\mathbf{v}_i - \mathbf{v}_1\|_2. \tag{2.5}$$

Proposition 2.2.3 *The diameter and the maximum oriented length satisfy the following inequalities*

$$\sigma_+(S) \leq diam(S) \leq 2\sigma_+(S). \tag{2.6}$$

Proof We begin by proving that

$$\sigma_+(S) \leq diam(S). \tag{2.7}$$

This is directly implied by the inequality

$$\begin{aligned} \max_{i=2,m} \|\mathbf{v}_i - \mathbf{v}_1\|_2 &\leq \max_{i=1,m} \|\mathbf{v}_i - \mathbf{v}_1\|_2 && (2.8)\\ &\leq \max_{i,j=1,m} \|\mathbf{v}_i - \mathbf{v}_j\|_2, && (2.9) \end{aligned}$$

which concludes the first part of the proof. We shall now proove the inequality

$$diam(S) \leq 2\sigma_+(S). \tag{2.10}$$

We decompose the difference $\mathbf{v}_i - \mathbf{v}_j$ into

$$\mathbf{v}_i - \mathbf{v}_j = (\mathbf{v}_i - \mathbf{v}_1) + (\mathbf{v}_1 - \mathbf{v}_j). \tag{2.11}$$

Hence,

$$\|\mathbf{v}_i - \mathbf{v}_j\|_2 \leq \|\mathbf{v}_i - \mathbf{v}_1\|_2 + \|\mathbf{v}_1 - \mathbf{v}_j\|_2. \tag{2.12}$$

We take the maximum over i and j, which leads to

$$\max_{i,j=1,m} \|\mathbf{v}_i - \mathbf{v}_j\|_2 \leq \max_{i=1,m} \|\mathbf{v}_i - \mathbf{v}_1\|_2 + \max_{j=1,m} \|\mathbf{v}_1 - \mathbf{v}_j\|_2 \quad (2.13)$$

$$\leq 2 \max_{i=1,m} \|\mathbf{v}_i - \mathbf{v}_1\|_2. \quad (2.14)$$

With the definitions of the diameter and the oriented length, this immediately prooves the inequality 2.10. ∎

In Nash's book [25], the size of the simplex $s_N(S)$ is measured based on the 1-norm and is defined by

$$s_N(S) = \sum_{i=2,m} \|\mathbf{v}_i - \mathbf{v}_1\|_1 \quad (2.15)$$

where the 1-norm is defined by

$$\|\mathbf{v}_i\|_1 = \sum_{j=1,n} |(\mathbf{v}_i)_j|. \quad (2.16)$$

The *optimsimplex_size* function provides all these size algorithms. In the following example, we create an axis-by-axis simplex with length unity and compute its length by several methods.

```
xx0 = [0.0 0.0];
si = optimsimplex_new ( "axes" , x0 );
methodlist = [
"sigmaplus"
"sigmaminus"
"Nash"
"diameter"
];
for i = 1:size(methodlist,"*")
  m = methodlist ( i );
  ss = optimsimplex_size ( si , m );
  mprintf ( "%s: %f\n", m , ss );
end
optimsimplex_destroy(si)
```

The previous script produces the following output.

```
sigmaplus: 1.000000
sigmaminus: 1.000000
Nash: 2.000000
diameter: 1.414214
```

We check that the diameter is equal to $diam(S) = \sqrt{2}$. We see that inequality 2.6 is satisfied since $\sigma_+(S) = 1 \leq \sqrt{2} \leq 2 = 2\sigma_+(S)$.

2.3 The initial simplex

While most of the theory can be developed without being very specific about the initial simplex, it plays a very important role in practice. All approaches are based on the initial guess $\mathbf{x}_0 \in \mathbb{R}^n$ and create a geometric shape based on this point.

In this section, we present the various approach to design the initial simplex. In the first part, we emphasize the importance of the initial simplex in optimization algorithms. Then we present the regular simplex by Spendley et al., the axis-by-axis simplex, the randomized bounds approach by Box and Pfeffer's simplex.

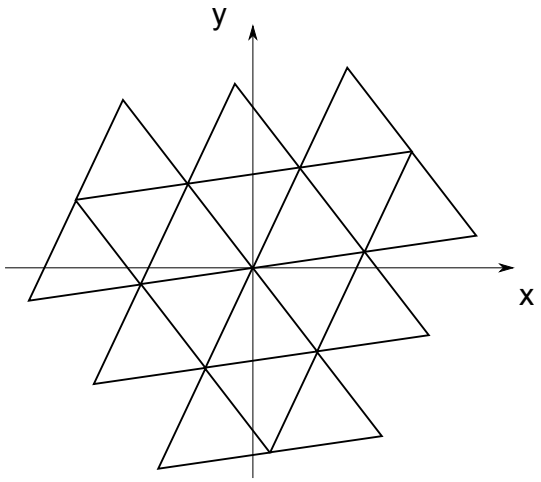

Fig. 2.1 : Typical pattern with fixed-shape Spendley's et al algorithm

2.3.1 Importance of the initial simplex

The initial simplex is particularily important in the case of Spendley's et al method, where the shape of the simplex is fixed during the iterations. Therefore, the algorithm can only go through points which are on the pattern defined by the initial simplex. The pattern presented in figure 2.1 is typical a fixed-shape simplex algorithm (see [46], chapter 3, for other patterns of a direct search method). If, by chance, the pattern is so that the optimum is close to one point defined by the pattern, the number of iteration may be small. On the contrary, the number of iterations may be large if the pattern does not come close to the optimum.

The variable-shape simplex algorithm designed by Nelder and Mead is also very sensitive to the initial simplex. One of the problems is that the initial simplex should be consistently scaled with respect to the unknown $\mathbf{x}$. In "An investigation into the efficiency of variants on the simplex method" [33], Parkinson and Hutchinson explored several improvements of Nelder and Mead's algorithm. First, they investigate the sensitivity of the algorithm to the initial simplex. Two parameters were investigated, that is, the initial length and the orientation of the simplex. The conclusion of their study with respect to the initial simplex is the following. "The orientation of the initial simplex has a significant effect on efficiency, but the relationship can be too sensitive for an automatic predictor to provide sufficient accuracy at this time."

Since no initial simplex clearly improves on the others, in practice, it may be convenient to try different approaches.

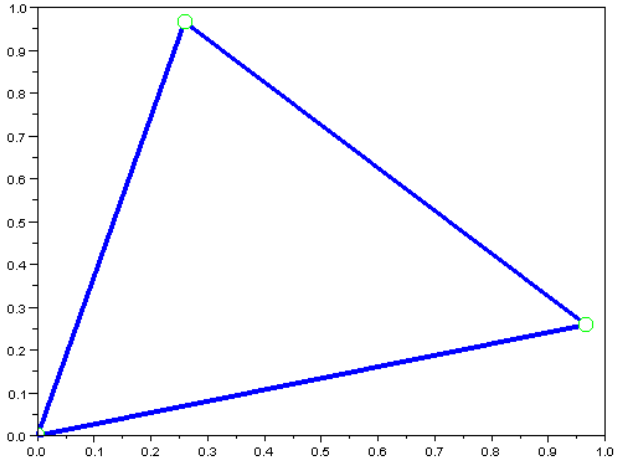

Fig. 2.2 : Regular simplex in 2 dimensions

2.3.2 Spendley's et al regular simplex

In their paper [43], Spendley, Hext and Himsworth use a regular simplex with given size $\ell > 0$. We define the parameters $p, q > 0$ as

$$p = \frac{1}{n\sqrt{2}}\left(n - 1 + \sqrt{n+1}\right), \tag{2.17}$$

$$q = \frac{1}{n\sqrt{2}}\left(\sqrt{n+1} - 1\right). \tag{2.18}$$

We can now define the vertices of the simplex $S = \{\mathbf{x}_i\}_{i=1,n+1}$. The first vertex of the simplex is the initial guess

$$\mathbf{v}_1 = \mathbf{x}_0. \tag{2.19}$$

The other vertices are defined by

$$(\mathbf{v}_i)_j = \begin{cases} (\mathbf{x}_0)_j + \ell p, & \text{if } j = i-1, \\ (\mathbf{x}_0)_j + \ell q, & \text{if } j \neq i-1, \end{cases} \tag{2.20}$$

for vertices $i = 2, n+1$ and components $j = 1, n$, where $\ell \in \mathbb{R}$ is the length of the simplex and satisfies $\ell > 0$. Notice that this length is the same for all the edges which keeps the simplex regular.

The regular simplex is presented in figure 2.2.

In the following Scilab session, we define a regular simplex with the *optimsimplex_new* function.

```
x0 = [0.0 0.0];
si = optimsimplex_new ( "spendley" , x0 );
methodlist = [
"sigmaplus"
"sigmaminus"
"diameter"
];
for i = 1:size(methodlist,"*")
  m = methodlist ( i );
  ss = optimsimplex_size ( si , m );
  mprintf ( "%s: %f\n", m , ss );
end
optimsimplex_destroy(si);
```

The previous script produces the following output.

```
sigmaplus: 1.000000
sigmaminus: 1.000000
diameter: 1.000000
```

We check that the three sizes $diam(S)$, $\sigma_+(S)$ and $\sigma_-(S)$ are equal, as expected from a regular simplex.

2.3.3 Axis-by-axis simplex

A very efficient and simple approach leads to an axis-by-axis simplex. This simplex depends on a vector of positive lengths $\mathbf{l} \in \mathbb{R}^n$. The first vertex of the simplex is the initial guess

$$\mathbf{v}_1 = \mathbf{x}_0. \tag{2.21}$$

The other vertices are defined by

$$(\mathbf{v}_i)_j = \begin{cases} (\mathbf{x}_0)_j + \ell_j, \text{ if } j = i-1, \\ (\mathbf{x}_0)_j, \text{ if } j \neq i-1, \end{cases} \tag{2.22}$$

for vertices $i = 2, n+1$ and components $j = 1, n$.

This type of simplex is presented in figure 2.3, where $\ell_1 = 1$ and $\ell_2 = 2$. The axis-by-axis simplex is used in the Nelder-Mead algorithm provided in Numerical Recipes in C [37]. As stated in [37], the length vector $\mathbf{l}$ can be used as a guess for the characteristic length scale of the problem.

2.3.4 Randomized bounds

Assume that the variable $\mathbf{x} \in \mathbb{R}^n$ is bounded so that

$$m_j \leq x_j \leq M_j, \tag{2.23}$$

for $j = 1, n$, where $m_j, M_j \in \mathbb{R}$ are minimum and maximum bounds and $m_j \leq M_j$. A method suggested by Box in [4] is based on the use of pseudo-random numbers. Let $\{\theta_{i,j}\}_{i=1,n+1,j=1,n} \in [0,1]$ be a sequence of random numbers uniform in the interval $[0,1]$. The first vertex of the simplex is the initial guess

$$\mathbf{v}_1 = \mathbf{x}_0. \tag{2.24}$$

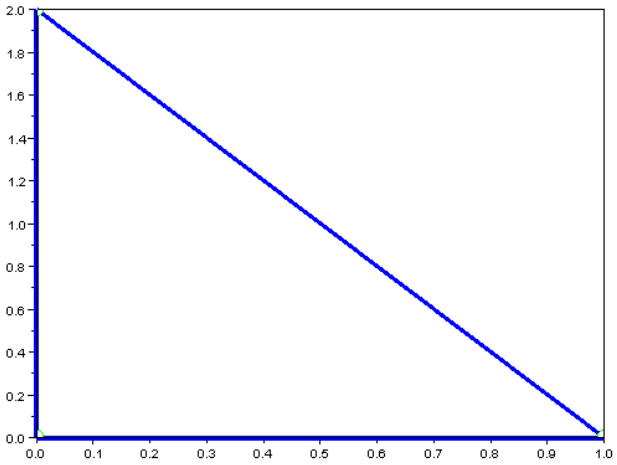

Fig. 2.3 : Axis-based simplex in 2 dimensions – Notice that the length along the x axis is 1 while the length along the y axis is 2.

The other vertices are defined by

$$(\mathbf{v}_i)_j = m_j + \theta_{i,j}(M_j - m_j), \tag{2.25}$$

for vertices $i = 2, n+1$ and components $j = 1, n$.

2.3.5 Pfeffer's method

This initial simplex is used in the function *fminsearch* and presented in [7]. According to [7], this simplex is due to L. Pfeffer at Stanford. The goal of this method is to scale the initial simplex with respect to the characteristic lengths of the problem. This allows, for example, to manage cases where $x_1 \approx 1$ and $x_2 \approx 10^5$. As we are going to see, the scaling is defined with respect to the initial guess $\mathbf{x}_0$, with an axis-by-axis method.

The method proceeds by defining $\delta_u, \delta_z > 0$, where δ_u is used for usual components of $\mathbf{x}_0$ and δ_z is used for the case where one component of $\mathbf{x}_0$ is zero. The default values for δ_u and δ_z are

$$\delta_u = 0.05 \qquad \text{and} \qquad \delta_z = 0.0075. \tag{2.26}$$

The first vertex of the simplex is the initial guess

$$\mathbf{v}_1 = \mathbf{x}_0. \tag{2.27}$$

The other vertices are defined by

$$(\mathbf{v}_i)_j = \begin{cases} (\mathbf{x}_0)_j + \delta_u(\mathbf{x}_0)_j, \text{ if } j = i-1 \text{ and } (\mathbf{x}_0)_{j-1} \neq 0, \\ \delta_z, \text{ if } j = i-1 \text{ and } (\mathbf{x}_0)_{j-1} = 0, \\ (\mathbf{x}_0)_j, \text{ if } j \neq i-1, \end{cases} \tag{2.28}$$

for vertices $i = 2, n+1$ and components $j = 1, n$.

2.4 The simplex gradient

In this section, we present the simplex gradient and proove that this gradient is an approximation of the gradient of the objective function, provided that the condition of the matrix of simplex directions. We derive the forward simplex gradient.

2.4.1 Matrix of simplex directions

We consider here simplices made of $m = n + 1$ vertices only. This allows to define the matrix of simplex directions as presented in the following definition.

Definition 2.4.1 *(*Matrix of simplex directions*) Assume that S is a set of $m = n + 1$ vertices. The $n \times n$ matrix of simplex directions $D(S)$ is defined by*

$$D(S) = (\mathbf{v}_2 - \mathbf{v}_1, \mathbf{v}_2 - \mathbf{v}_1, \ldots, \mathbf{v}_{n+1} - \mathbf{v}_1). \tag{2.29}$$

We define by $\{\mathbf{d}_i\}_{i=1,n}$ the columns of the $n \times n$ matrix $D(S)$, i.e.

$$D(S) = (\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n). \tag{2.30}$$

We say that the simplex S is nonsingular *if the matrix $D(S)$ is* nonsingular*. We define the* simplex condition *as the l^2 condition number of the matrix of simplex directions $\kappa(D(S))$.*

The directions $\mathbf{d}_i$ can be seen as *offsets*, leading from the first vertex to each vertex $\mathbf{v}_i$, i.e.

$$\mathbf{v}_i = \mathbf{v}_1 + \mathbf{d}_1, \text{ for } i = 1, n. \tag{2.31}$$

Example (*A non degenerate simplex*) Consider the axis-by-axis simplex, with first vertex at origin and lengths unity. The vertices are defined by

$$\mathbf{v}_1 = (0, 0)^T, \qquad \mathbf{v}_2 = (1, 0)^T, \qquad \mathbf{v}_3 = (0, 1)^T, \tag{2.32}$$

so that the matrix of simplex directions is given by

$$D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{2.33}$$

Such a matrix has a unity condition number.

The following Scilab session uses the *optimsimplex* component to generate a axis-by-axis simplex and computes the matrix of directions with the *optimsimplex_dirmat* function.

```
x0 = [0.0 0.0];
si = optimsimplex_new ( "axes" , x0 );
D = optimsimplex_dirmat ( si )
k = cond(D)
optimsimplex_destroy(si)
```

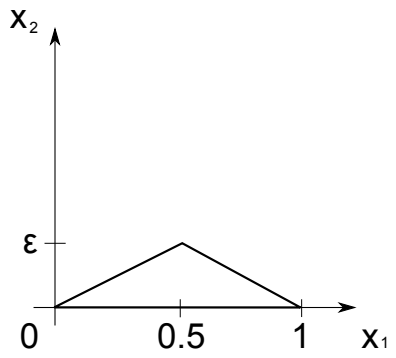

Fig. 2.4 : A "flat" simplex in 2 dimensions

The previous script produces the following output.

```
-->D = optimsimplex_dirmat ( si )
 D  =
    1.      0.
    0.      1.
-->k = cond(D)
 k  =
    1.
```

We check that an axis-by-axis simplex has a very low condition number. □

Example (*A degenerate simplex*) In this example, we show that a flat simplex is associated with a high condition number. Consider a flat simplex, defined by its vertices:

$$\mathbf{v}_1 = (0,0)^T, \qquad \mathbf{v}_2 = (1,0)^T, \qquad \mathbf{v}_3 = (1/2,\epsilon)^T, \tag{2.34}$$

with $\epsilon = 10^{-10}$. This simplex is presented in figure 2.4.

```
coords = [
0.0  0.0
1.0  0.0
0.5  1.e-10
];
si = optimsimplex_new ( coords );
D = optimsimplex_dirmat ( si )
k = cond(D)
optimsimplex_destroy(si);
```

The previous script produces the following output.

```
-->D = optimsimplex_dirmat ( si )
 D  =
    1.      0.5
    0.      1.000D-10
-->k = cond(D)
 k  =
    1.250D+10
```

We see that a flat simplex is associated with a high condition number. Indeed, a low condition number of the matrix of directions is an indication of the non-degeneracy of the simplex. □

There is a close relationship between the oriented length $\sigma_+(S)$ and the l^2 norm of the matrix of directions $D(S)$ as prooved in the following proposition.

Proposition 2.4.2 *Let S be a simplex and consider the euclidian norm $\|.\|$. Therefore,*

$$\|\mathbf{d}_i\| \leq \sigma_+(S) \leq \|D\|, \tag{2.35}$$

for all $i = 1, \ldots, n$.

Proof It is easy to prove that

$$\|\mathbf{d}_i\| \leq \sigma_+(S). \tag{2.36}$$

Indeed, the definition of the oriented length $\sigma_+(S)$ in the case where there are $n+1$ vertices is

$$\begin{aligned} \sigma_+(S) &= \max_{i=2,n+1} \|\mathbf{v}_i - \mathbf{v}_1\|_2 & (2.37)\\ &= \max_{i=1,n} \|\mathbf{d}_i\|_2, & (2.38) \end{aligned}$$

which concludes the first part of the proof.

We shall now proove that

$$\sigma_+(S) \leq \|D\|. \tag{2.39}$$

The euclidian norm is so that ([10], section 2.3.1, "Definitions"),

$$\|D\mathbf{x}\| \leq \|D\|\|\mathbf{x}\|, \tag{2.40}$$

for any vector $\mathbf{x} \in \mathbb{R}^n$. We choose the specific vector $\mathbf{x}$ which has zeros components, except for the i-th row, which is equal to 1, i.e. $\mathbf{x} = (0, \ldots, 0, 1, 0, \ldots, 0)^T$. With this particular choice of $\mathbf{x}$ we have the properties $D\mathbf{x} = \mathbf{d}_i$ and $\|\mathbf{x}\| = 1$, so that the previous inequality becomes

$$\|\mathbf{d}_i\| \leq \|D\|, \tag{2.41}$$

for all $i = 1, \ldots, n$. We can now take the maximum of the left hand-size of 2.41 and get the oriented length $\sigma_+(S)$, which concludes the proof. ∎

Example In the following Scilab session, we define a new simplex by its coordinates, so that the matrix of directions is not symetric and that the edges do not have unit lengths.

```
coords = [
0.0 0.0
1.0 0.5
1.0 2.0
];
si = optimsimplex_new ( coords );
D = optimsimplex_dirmat ( si )
for i=1:2
  nd = norm(D(1:2,i),2);
  mprintf( "||d_%d||=%f\n",i,nd)
end
ss = optimsimplex_size ( si , "sigmaplus" );
mprintf( "sigma_+(S)=%f\n",ss);
normmatrix = norm(D);
mprintf( "||D||=%f\n",normmatrix);
optimsimplex_destroy(si);
```

The previous script produces the following output.

```
||d_1||=1.118034
||d_2||=2.236068
sigma_+(S)=2.236068
||D||=2.422078
```

This result is consistent with the inequality 2.35. □

2.4.2 Taylor's formula

The simplex gradient proposition that we shall proove in the next section assumes that the gradient $\mathbf{g}$ of the function f satisfies a Lipshitz condition. The following proposition presents a result satisfied by such functions. In order to simplify the notations, we denote by $\|.\|$ the euclidian norm.

Proposition 2.4.3 *Assume that $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable and assume that its gradient $\mathbf{g}$ is defined and continuous. Let $\mathbf{x} \in \mathbb{R}^n$ be a given point and $\mathbf{p} \in \mathbb{R}^n$ a vector. Assume that the gradient $\mathbf{g}$ is Lipshitz continuous in a neighbourhood of $\mathbf{x}$ and $\mathbf{x}+\mathbf{p}$ with Lipshitz constant L. Then*

$$|f(\mathbf{x}+\mathbf{p}) - f(\mathbf{x}) - \mathbf{p}^T\mathbf{g}(\mathbf{x})| \leq \frac{1}{2}L\|\mathbf{p}\|^2. \tag{2.42}$$

Proof We can write Taylor's expansion of f in a neighbourhood of $\mathbf{x}$

$$f(\mathbf{x}+\mathbf{p}) = f(\mathbf{x}) + \int_0^1 \mathbf{p}^T\mathbf{g}(\mathbf{x}+t\mathbf{p})dt. \tag{2.43}$$

By definition of the Lipshitz condition on $\mathbf{g}$, we have

$$\|\mathbf{g}(\mathbf{x}) - \mathbf{g}(\mathbf{y})\| \leq L\|\mathbf{x}-\mathbf{y}\|, \tag{2.44}$$

for $\mathbf{x}$ and $\mathbf{y}$ in that neighbourhood. Assume that $t \in [0,1]$ and use the particular point $\mathbf{y} = \mathbf{x}+t\mathbf{p}$. We get

$$\|\mathbf{g}(\mathbf{x}+t\mathbf{p}) - \mathbf{g}(\mathbf{x})\| \leq tL\|\mathbf{p}\|. \tag{2.45}$$

We now use equality 2.43, substract the term $\mathbf{p}^T\mathbf{g}(\mathbf{x})$ and get

$$f(\mathbf{x}+\mathbf{p}) - f(\mathbf{x}) - \mathbf{p}^T\mathbf{g}(\mathbf{x}) = \int_0^1 \mathbf{p}^T\left(\mathbf{g}(\mathbf{x}+t\mathbf{p}) - \mathbf{g}(\mathbf{x})\right)dt. \tag{2.46}$$

Therefore,

$$\begin{aligned} \left|f(\mathbf{x}+\mathbf{p}) - f(\mathbf{x}) - \mathbf{p}^T\mathbf{g}(\mathbf{x})\right| &= \left|\int_0^1 \mathbf{p}^T\left(\mathbf{g}(\mathbf{x}+t\mathbf{p}) - \mathbf{g}(\mathbf{x})\right)dt\right| && (2.47)\\ &\leq \int_0^1 \|\mathbf{p}\|\,\|\mathbf{g}(\mathbf{x}+t\mathbf{p}) - \mathbf{g}(\mathbf{x})\|\,dt && (2.48) \end{aligned}$$

We plug 2.45 into the previous equality and get

$$\begin{aligned} \left|f(\mathbf{x}+\mathbf{p}) - f(\mathbf{x}) - \mathbf{p}^T\mathbf{g}(\mathbf{x})\right| &\leq \int_0^1 Lt\|\mathbf{p}\|^2 dt && (2.49)\\ &\leq \frac{1}{2}L\|\mathbf{p}\|^2, && (2.50) \end{aligned}$$

which concludes the proof. ∎

2.4.3 Forward difference simplex gradient

Finite difference formulas are a common tool to compute the numerical derivative of a function. In this section, we introduce the simplex gradient, which allows to compute an approximation of the gradient of the cost function. As we are going to see, this approximation is more accurate when the simplex has a low condition number.

We denote by $\delta(S)$ the vector of objective function differences

$$\delta(S) = (f(\mathbf{v}_2) - f(\mathbf{v}_1), f(\mathbf{v}_3) - f(\mathbf{v}_1), \ldots, f(\mathbf{v}_{n+1}) - f(\mathbf{v}_1))^T. \tag{2.51}$$

As with classical finite difference formulas, the vector of function can be used to compute the simplex gradient.

Definition 2.4.4 *(*Simplex gradient*) Let S be a non singular simplex. The* simplex gradient $\overline{\mathbf{g}}(S)$ *is the unique solution of the linear system of equations*

$$D(S)^T \overline{\mathbf{g}}(S) = \delta(S). \tag{2.52}$$

By hypothesis, the simplex S is nonsingular so that the linear system of equations has a unique solution, which is equal to

$$\overline{\mathbf{g}}(S) = (D(S)^T)^{-1} \delta(S). \tag{2.53}$$

By hypothesis, the matrix $D(S)$ is non singular, therefore the transpose of the inverse is equal to the inverse of the transpose ([10], section 2.1.3, "Matrix Inverse"), i.e. $(D(S)^T)^{-1} = (D(S)^{-1})^T$. We denote by $D(S)^{-T}$ the inverse of the transpose so that the previous equality becomes

$$\overline{\mathbf{g}}(S) = D(S)^{-T} \delta(S). \tag{2.54}$$

In practice, the matrix of simplex direction is not inverted and the solution of 2.52 is computed directly, using classical linear algebra libraries, like Lapack for example.

The simplex gradient is an approximation of the gradient $\mathbf{g}$ of the function f, as presented in the following proposition.

Proposition 2.4.5 *Let S be a simplex. Let the gradient $\mathbf{g}$ be Lipshitz continuous in a neighbourhood of S with Lipshitz constant L. Consider the euclidian norm $\|.\|$. Then, there is a constant $K > 0$, depending only on L such that*

$$\|\mathbf{g}(\mathbf{v}_1) - \overline{\mathbf{g}}(S)\|_2 \leq K \kappa(S) \sigma_+(S). \tag{2.55}$$

Proof We can write the difference between the simplex gradient and the gradient in the following form

$$\overline{\mathbf{g}}(S) - \mathbf{g}(\mathbf{v}_1) = D(S)^{-T} \left(D(S)^T \overline{\mathbf{g}}(S) - D(S)^T \mathbf{g}(\mathbf{v}_1) \right). \tag{2.56}$$

We now plug the simplex gradient definition 2.52 into the previous equality and get

$$\overline{\mathbf{g}}(S) - \mathbf{g}(\mathbf{v}_1) = D(S)^{-T} \left(\delta(S) - D(S)^T \mathbf{g}(\mathbf{v}_1)\right). \tag{2.57}$$

The fact that the euclidian norm $\|.\|$ satisfies the inequality

$$\|AB\| \leq \|A\|\|B\|, \tag{2.58}$$

for any matrices A and B with suitable number of rows and columns ([10], section 2.3, "Matrix Norms") plays an important role in the results that we are going to derive. Indeed, we can compute the euclidian norm of both sides of equation 2.57 and get

$$\|\overline{\mathbf{g}}(S) - \mathbf{g}(\mathbf{v}_1)\| = \left\|D(S)^{-T} \left(\delta(S) - D(S)^T \mathbf{g}(\mathbf{v}_1)\right)\right\|. \tag{2.59}$$

Therefore,

$$\|\overline{\mathbf{g}}(S) - \mathbf{g}(\mathbf{v}_1)\| \leq \left\|D(S)^{-T}\right\| \left\|\delta(S) - D(S)^T \mathbf{g}(\mathbf{v}_1)\right\|. \tag{2.60}$$

The suite of the proof is based on the computation of the right-hand side of equation 2.60, that is, the computation of the norm of the vector $\delta(S) - D(S)^T \mathbf{g}(\mathbf{v}_1)$.

By hypothesis, the gradient $\mathbf{g}$ is Lipshitz continuous in a neighbourhood of S. By proposition 2.4.3, we have

$$\left|f(\mathbf{v}_1 + \mathbf{d}_i) - f(\mathbf{v}_1) - \mathbf{d}_i^T \mathbf{g}(\mathbf{v}_1)\right| \leq \frac{1}{2} L \|\mathbf{d}_i\|^2, \tag{2.61}$$

for $i = 1, n$. By definition of the direction $\mathbf{d}_i$, we have $\mathbf{v}_1 + \mathbf{d}_i = \mathbf{v}_i$ for $i = 1, n$. By proposition 2.4.2, we have $\|\mathbf{d}_j\| \leq \sigma_+(S)$ for all $j = 1, n$. Hence,

$$\left|f(\mathbf{v}_i) - f(\mathbf{v}_1) - \mathbf{d}_i^T \mathbf{g}(\mathbf{v}_1)\right| \leq \frac{1}{2} L \sigma_+(S)^2, \tag{2.62}$$

We can use this to compute the euclidian norm of the vector $\delta(S) - D^T \mathbf{g}(\mathbf{v}_1)$. Using ineguality 2.62, the square of the norm of this vector is

$$\begin{aligned}
\left\|\delta(S) - D^T \mathbf{g}(\mathbf{v}_1)\right\|^2 &= \sum_{i=1,n} \left(f(\mathbf{v}_i) - f(\mathbf{v}_1) - \mathbf{d}_i^T \mathbf{g}(\mathbf{v}_1)\right)^2 && (2.63)\\
&\leq \sum_{i=1,n} \left(\frac{1}{2} L \sigma_+(S)^2\right)^2 && (2.64)\\
&\leq n \left(\frac{1}{2} L \sigma_+(S)^2\right)^2 && (2.65)
\end{aligned}$$

which finally implies

$$\left\|\delta(S) - D^T \mathbf{g}(\mathbf{v}_1)\right\|^2 \leq \frac{1}{2} \sqrt{n} L \sigma_+(S)^2. \tag{2.66}$$

Let us define the constant $K = \frac{1}{2}\sqrt{n}L$. The previous inequality becomes

$$\left\|\delta(S) - D^T\mathbf{g}(\mathbf{v}_1)\right\|^2 \leq K\sigma_+(S)^2. \tag{2.67}$$

We can now plug the previous equality into inequality 2.60 and get

$$\|\overline{\mathbf{g}}(S) - \mathbf{g}(\mathbf{v}_1)\| \leq K\left\|D(S)^{-T}\right\|\sigma_+(S)^2. \tag{2.68}$$

By proposition 2.4.2, we have $\sigma_+(S) \leq \|D\|$, so that the previous inequality is transformed into

$$\|\overline{\mathbf{g}}(S) - \mathbf{g}(\mathbf{v}_1)\| \leq K\left\|D(S)^{-T}\right\|\|D(S)\|\sigma_+(S). \tag{2.69}$$

The l^2 norm of the matrix $D(S)$ is the largest eigenvalue of the matrix $D(S)^TD(S)$, so that the norm is not affected by transposition, which implies that $\left\|D(S)^{-T}\right\| = \|D(S)^{-1}\|$. The condition number of the matrix of direction $\kappa(S)$ is equal to $\|D(S)^{-1}\|\,\|D(S)\|$ ([10], section 2.7.2, "Condition"), which concludes the proof. ∎

Example (*Simplex gradient with a non-degenerate simplex*) In the following Scilab session, we define the function $f(\mathbf{x}) = x_1^2 + x_2^2$, where $\mathbf{x} \in \mathbb{R}^2$. The exact gradient of this function is $\mathbf{g} = (x_1, x_2)^T$. We create an axis-by-axis simplex based on the relatively small length $\ell = 10^{-3}$. This simplex defines a rectangular triangle, similar to the one presented in figure 2.3, but with smaller edges.

```
function y = myfunction ( x )
  y = x(1)^2 + x(2)^2
endfunction
x0 = [1.0 1.0];
len = 1.e-3;
si = optimsimplex_new ( "axes" , x0 , myfunction , len );
sg = optimsimplex_gradientfv ( si );
mprintf ( "Simplex Gradient=(%f %f)^T\n",sg(1),sg(2));
eg = [2 * x0(1) 2 * x0(2)].';
mprintf ( "Exact Gradient=(%f %f)^T\n",eg(1),eg(2));
err = norm(sg-eg)/norm(eg);
mprintf ( "Relative Error = %e\n",err);
err = norm(sg-eg);
mprintf ( "Absolute Error = %e\n",err);
D = optimsimplex_dirmat ( si );
k = cond(D);
mprintf ( "k(D)=%f\n",k);
ss = optimsimplex_size ( si );
mprintf ( "sigma_+(D)=%e\n",ss);
optimsimplex_destroy(si);
```

The previous script produces the following output.

```
Simplex Gradient=(2.001000 2.001000)^T
Exact Gradient=(2.000000 2.000000)^T
Absolute Error = 1.414214e-003
k(D)=1.000000
sigma_+(D)=1.000000e-003
```

We check that the inequality 2.55 gives an accurate measure of the approximation. Indeed, since the Lipshitz constant for the gradient $\mathbf{g}$ is $L = 2$, we have the constant $K = \sqrt{2}$. □

Example (*Simplex gradient with a simplex close to degenerate*) We consider what happens when an axis-by-axis simplex is transformed into a degenerate simplex. This situation is presented in

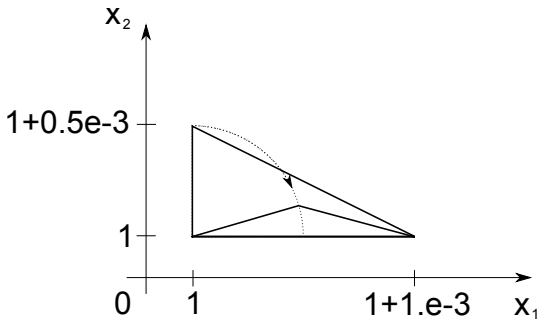

Fig. 2.5 : An axis-by-axis simplex which degenerates into a "flat" simplex in 2 dimensions.

θ (°)	$\sigma_+(S)$	$\|\overline{\mathbf{g}}(S) - \mathbf{g}(\mathbf{v}_1)\|$	$\kappa(S)$
90.000000	1.000000e-003	1.118034e-003	2.000000e+000
10.000000	1.000000e-003	2.965584e-003	1.432713e+001
1.000000	1.000000e-003	2.865807e-002	1.432397e+002
0.100000	1.000000e-003	2.864799e-001	1.432395e+003
0.010000	1.000000e-003	2.864789e+000	1.432394e+004
0.001000	1.000000e-003	2.864789e+001	1.432394e+005

figure 2.5, where the third vertex moves on a circle with radius $0.5.10^{-3}$ toward the center of an edge. Therefore the simplex degenerates and its condition number increases dramatically.

In the following Scilab script, we create a simplex as presented in figure 2.5. We use decreasing values of the angle θ between the two directions, starting from $\theta = 90$ (°) and going down to $\theta = 0.001$ (°). Then we compute the gradient and the absolute error, as well as the condition number and the size of the simplex.

```
R = 0.5e-3
coords = [
  1.0 1.0
  1.0+1.e-3 1.0
];
for theta = [90.0 10.0 1.0 0.1 0.01 0.001]
  C(1,1) = 1.0 + R * cos(theta*%pi/180);
  C(1,2) = 1.0 + R * sin(theta*%pi/180);
  coords(3,1:2) = C;
  si = optimsimplex_new ( coords , myfunction );
  sg = optimsimplex_gradientfv ( si );
  eg = [2 * x0(1) 2 * x0(2)].';
  err = norm(sg-eg);
  D = optimsimplex_dirmat ( si );
  k = cond(D);
  ss = optimsimplex_size ( si );
  mprintf ( "%f %e %e %e\n",theta , ss , err , k);
  optimsimplex_destroy(si);
end
```

The results are presented in table 2.4.3.

We see that while the oriented length $\sigma_+(S)$ is constant, the simplex gradient is more and more inaccurate as the condition number $\kappa(S)$ is increasing. □

2.5 References and notes

The section 2.4.3 and some elements of the section 2.2 are taken from Kelley's book [18], "Iterative Methods for Optimization". While this book focus on Nelder-Mead algorithm, Kelley gives a broad view on optimization and present other algorithms for noisy functions, like implicit filtering, multidirectional search and the Hooke-Jeeves algorithm.

The section 2.4.2, which present Taylor's formula with a Lisphitz continous gradient is based on [15], "Elements of Analysis, Geometry, Topology", section "Mean Value Theorem".

Chapter 3

Spendley's et al. method

In this chapter, we present Spendley, Hext and Himsworth algorithm [43] for unconstrained optimization.

We begin by presenting a global overview of the algorithm. Then we present various geometric situations which might occur during the algorithm. In the second section, we present several numerical experiments which allow to get some insight in the behavior of the algorithm on some simple situations. The two first cases are involving only 2 variables and are based on a quadratic function. The last numerical experiment explores the behavior of the algorithm when the number of variables increases.

3.1 Introduction

In this section, we present Spendley's et al algorithm for unconstrained optimization. This algorithm is based on the iterative update of a simplex. At each iteration, either a reflection of a shrink step is performed, so that the shape of the simplex does not change during the iterations. Then we present various geometric situations which might occur during the algorithm. This allows to understand when exactly a reflection or a shrink is performed in practice.

3.1.1 Overview

The goal of Spendley's et al. algorithm is to solve the following unconstrained optimization problem

$$\min f(\mathbf{x}) \tag{3.1}$$

where $\mathbf{x} \in \mathbb{R}^n$, n is the number of optimization parameters and f is the objective function $f : \mathbb{R}^n \to \mathbb{R}$.

This algorithms is based on the iterative update of a *simplex* made of $n+1$ points $S = \{\mathbf{v}_i\}_{i=1,n+1}$. Each point in the simplex is called a *vertex* and is associated with a function value $f_i = f(\mathbf{v}_i)$ for $i = 1, n+1$.

The vertices are sorted by increasing function values so that the *best* vertex has index 1 and the *worst* vertex has index $n+1$

$$f_1 \leq f_2 \leq \ldots \leq f_n \leq f_{n+1}. \tag{3.2}$$

The $\mathbf{v}_1$ vertex (resp. the $\mathbf{v}_{n+1}$ vertex) is called the *best* vertex (resp. *worst*), because it is associated with the lowest (resp. highest) function value. As we are going to see, the *next-to-worst* vertex $\mathbf{v}_n$ has a special role in this algorithm.

The centroid of the simplex $\overline{\mathbf{x}}(j)$ is the center of the vertices where the vertex $\mathbf{v}_j$ has been excluded. This centroid is

$$\overline{\mathbf{x}}(j) = \frac{1}{n} \sum_{i=1,n+1,i\neq j} \mathbf{v}_i. \tag{3.3}$$

The algorithm makes use of one coefficient $\rho > 0$, called the reflection factor. The standard value of this coefficient is $\rho = 1$. The algorithm attempts to replace some vertex $\mathbf{v}_j$ by a new vertex $\mathbf{x}(\rho, j)$ on the line from the vertex $\mathbf{v}_j$ to the centroid $\overline{\mathbf{x}}(j)$. The new vertex $\mathbf{x}(\rho, j)$ is defined by

$$\mathbf{x}(\rho, j) = (1+\rho)\overline{\mathbf{x}}(j) - \rho\mathbf{v}_j. \tag{3.4}$$

3.1.2 Algorithm

In this section, we analyze Spendley's et al algorithm, which is presented in figure 3.1.

At each iteration, we compute the centroid $\overline{\mathbf{x}}(n+1)$ where the worst vertex $\mathbf{v}_{n+1}$ has been excluded. This centroid is

$$\overline{\mathbf{x}}(n+1) = \frac{1}{n} \sum_{i=1,n} \mathbf{v}_i. \tag{3.5}$$

We perform a reflection with respect to the worst vertex $\mathbf{v}_{n+1}$, which creates the reflected point $\mathbf{x}_r$ defined by

$$\mathbf{x}_r = \mathbf{x}(\rho, n+1) = (1+\rho)\overline{\mathbf{x}}(n+1) - \rho\mathbf{v}_{n+1} \tag{3.6}$$

We then compute the function value of the reflected point as $f_r = f(\mathbf{x}_r)$. If the function value f_r is better than the worst function value f_{n+1}, i.e. if $f_r < f_{n+1}$, then the worst vertex $\mathbf{v}_{n+1}$ is rejected from the simplex and the reflected point $\mathbf{x}_r$ is accepted. If the reflection point does not improve the function value f_{n+1}, we consider the centroid $\overline{\mathbf{x}}(n)$, i.e. the centroid where the next-to-worst vertex $\mathbf{v}_n$ has been excluded. We then consider the reflected point $\mathbf{x}'_r$, computed from the next-to-worst vertex $\mathbf{v}_n$ and the centroid $\overline{\mathbf{x}}(n)$. We compute the function value $f'_r = f(\mathbf{x}'_r)$. If the function value f'_r improves over the worst function value f_{n+1}, then the worst vertex $\mathbf{v}_{n+1}$ is rejected from the simplex and the new reflection point $\mathbf{x}'_r$ is accepted.

At that point of the algorithm, neither the reflection with respect to $\mathbf{v}_{n+1}$ nor the reflection with respect to $\mathbf{v}_n$ were able to improve over the worst function value f_{n+1}. Therefore, the algorithm shrinks the simplex toward the best vertex $\mathbf{v}_1$. That last step uses the shrink coefficient $0 < \sigma < 1$. The standard value for this coefficient is $\sigma = \frac{1}{2}$.

Compute an initial simplex S_0
Sorts the vertices S_0 with increasing function values
$S \leftarrow S_0$
while $\sigma(S) > tol$ **do**
  $\overline{x} \leftarrow \overline{\mathbf{x}}(n+1)$ {Compute the centroid}
  $\mathbf{x}_r \leftarrow \mathbf{x}(\rho, n+1)$ {Reflect with respect to worst}
  $f_r \leftarrow f(\mathbf{x}_r)$
  if $f_r < f_{n+1}$ **then**
    Accept $\mathbf{x}_r$
  else
    $\overline{x} \leftarrow \overline{\mathbf{x}}(n)$ {Compute the centroid}
    $\mathbf{x}'_r \leftarrow \mathbf{x}(\rho, n)$ {Reflect with respect to next-to-worst}
    $f'_r \leftarrow f(\mathbf{x}'_r)$
    if $f'_r < f_{n+1}$ **then**
      Accept $\mathbf{x}'_r$
    else
      Compute the vertices $\mathbf{v}_i = \mathbf{v}_1 + \sigma(\mathbf{v}_i - \mathbf{v}_1)$ for $i = 2, n+1$ {Shrink}
      Compute $f_i = f(\mathbf{v}_i)$ for $i = 2, n+1$
    end if
  end if
  Sort the vertices of S with increasing function values
end while

Fig. 3.1 : Spendley's et al. algorithm

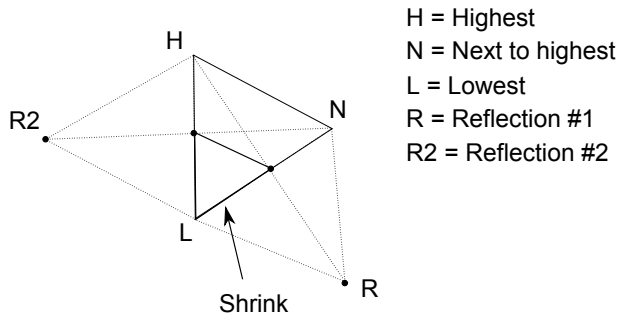

Fig. 3.2 : Spendley et al. simplex moves

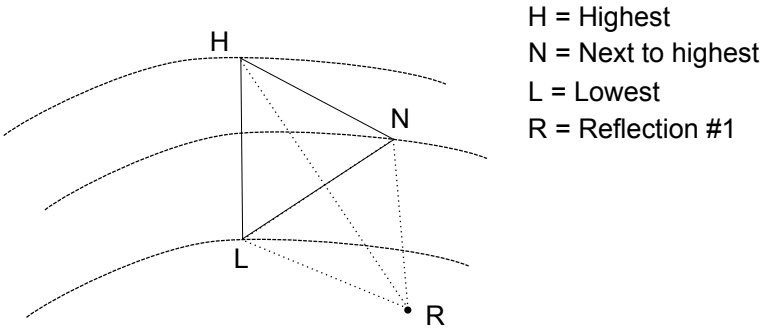

Fig. 3.3 : Spendley et al. simplex moves – Reflection with respect to highest point

3.1.3 Geometric analysis

The figure 3.2 presents the various moves of the Spendley et al. algorithm. It is obvious from the picture that the algorithm explores a pattern which is entirely determined from the initial simplex.

In Spendley's et al. original paper, the authors use a regular simplex, where the edges all have the same length. In practice, however, any non degenerate simplex can be used.

The various situations in which these moves are performed are presented in figures 3.3, 3.4 and 3.5.

The basic move is the reflection step, presented in figure 3.3 and 3.4. These two figures show that Spendley's et al. algorithm is based on a discretization of the parameter space. The optimum is searched on that grid, which is based on regular simplices. When no move is possible to improve the situation on that grid, a shrink step is necessary, as presented in figure 3.5.

In the situation of figure 3.5, neither the reflection #1 or reflection #2 have improved the simplex. Diminishing the size of the simplex by performing a shrink step is the only possible move because the simplex has vertices which are located across the valley. This allows to refine the discretization grid on which the optimum is searched.

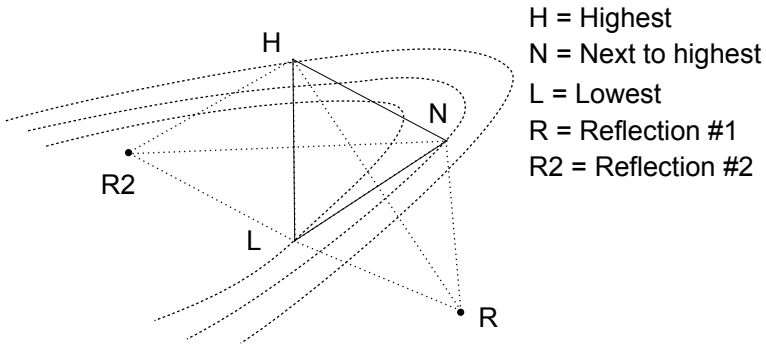

Fig. 3.4 : Spendley et al. simplex moves – Reflection with respect to next-to-highest point. It may happen that the next iteration is a shrink step.

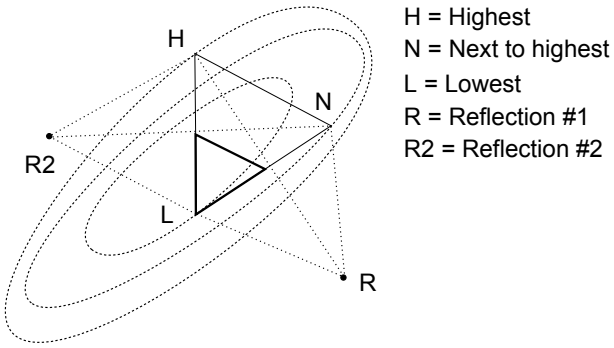

Fig. 3.5 : Spendley et al. simplex moves – The shrink step is the only possible move.

3.1.4 General features of the algorithm

From the performance point of viewn when a reflection step is performed, only 1 or 2 function evaluations are required. Instead, when a shrink step is performed, there are n function evaluations required. In practice, reflection steps are performed when the simplex is away from the optimum. When the simplex is closer to the optimum, or enters in a narrow valley, shrink steps are used.

As stated in [42], the main feature of Spendley's et al. algorithm is that the simplex can vary in size, but not in shape. As we are going to see in the numerical experiments, this leads to a slow convergence when a narrow valley is encountered. In that situation, the shrink steps are required, which leads to a large number of iterations and function evaluations.

In fact, the Spendley's et al. algorithm is a pattern search algorithm [45]. This is a consequence of the fact that the search pattern used in the method is constant. Therefore, the design never degenerates. As stated in [45], "under very mild assumptions on f, these simple heuristics provide enough structure to guarantee global convergence. This is not the case for the Nelder-Mead algorithm, which might converge to non-stationnary points [21, 14, 12, 46]. In all cases, the difficulty is that a sequence of simplices produced by the Nelder-Mead simplex method can come arbitrarily close to degeneracy.

3.2 Numerical experiments

In this section, we present some numerical experiments with Spendley's et al. algorithm. The first numerical experiments involves one quadratic function in 2 dimensions. The second experiment is based on a badly scaled quadratic in 2 dimension. In the third experiment, we analyze the behavior of the algorithm with respect to the number of variables.

3.2.1 Quadratic function

The function we try to minimize is the following quadratic in 2 dimensions

$$f(x_1, x_2) = x_1^2 + x_2^2 - x_1 x_2. \tag{3.7}$$

The stopping criteria is based on the relative size of the simplex with respect to the size of the initial simplex

$$\sigma_+(S) < tol \times \sigma_+(S_0). \tag{3.8}$$

The oriented length $\sigma_+(S)$ is defined by

$$\sigma_+(S) = \max_{i=2,n+1} \|\mathbf{v}_i - \mathbf{v}_1\|_2 \tag{3.9}$$

where $\|.\|_2$ is the euclidian norm defined by

$$\|\mathbf{x}\|_2 = \sum_{i=1,n} x_i^2. \tag{3.10}$$

In this experiment, we use $tol = 10^{-8}$ as the relative tolerance on simplex size.

The initial simplex is a regular simplex with length unity.

The following Scilab script performs the optimization.

```
function y = quadratic (x)
  y = x(1)^2 + x(2)^2 - x(1) * x(2);
endfunction
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-function",quadratic);
nm = neldermead_configure(nm,"-x0",[2.0 2.0]');
nm = neldermead_configure(nm,"-maxiter",100);
nm = neldermead_configure(nm,"-maxfunevals",300);
nm = neldermead_configure(nm,"-tolxmethod","disabled");
nm = neldermead_configure(nm,"-tolsimplexizerelative",1.e-8);
nm = neldermead_configure(nm,"-simplex0method","spendley");
nm = neldermead_configure(nm,"-method","fixed");
nm = neldermead_configure(nm,"-verbose",1);
nm = neldermead_configure(nm,"-verbosetermination",0);
nm = neldermead_search(nm);
neldermead_display(nm);
nm = neldermead_destroy(nm);
```

The numerical results are presented in table 3.6.

Iterations	49
Function Evaluations	132
$\mathbf{x}_0$	$(2.0, 2.0)$
Relative tolerance on simplex size	10^{-8}
Exact $\mathbf{x}^\star$	$(0., 0.)$
Computed $\mathbf{x}^\star$	$(2.169e-10, 2.169e-10)$
Exact $f(\mathbf{x}^\star)$	0.
Computed $f(\mathbf{x}^\star)$	$4.706e-20$

Fig. 3.6 : Numerical experiment with Spendley's et al. method on the quadratic function $f(x_1, x_2) = x_1^2 + x_2^2 - x_1 x_2$

The various simplices generated during the iterations are presented in figure 3.7. The method use reflections in the early iterations. Then there is no possible improvement using reflections and shrinking is necessary. That behavior is an illustration of the discretization which has already been discussed.

The figure 3.8 presents the history of the oriented length of the simplex. The length is updated step by step, where each step corresponds to a shrink in the algorithm.

The convergence is quite fast in this case, since less than 60 iterations allow to get a function value lower than 10^{-15}, as shown in figure 3.9.

3.2.2 Badly scaled quadratic function

The function we try to minimize is the following quadratic in 2 dimensions

$$f(x_1, x_2) = a x_1^2 + x_2^2, \tag{3.11}$$

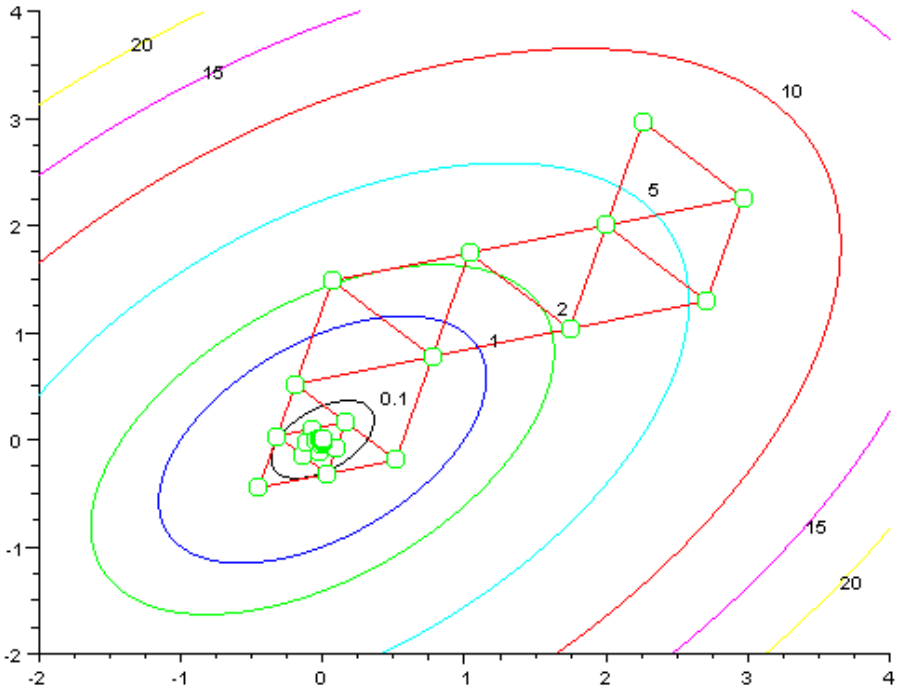

Fig. 3.7 : Spendley et al. numerical experiment – History of simplex

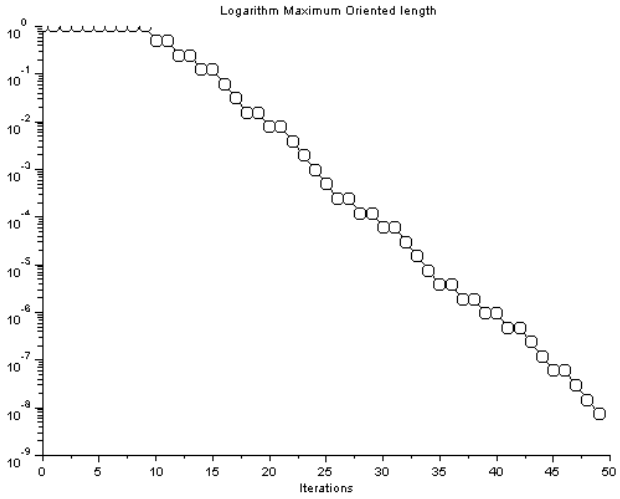

Fig. 3.8 : Spendley et al. numerical experiment – History of logarithm of the size of the simplex

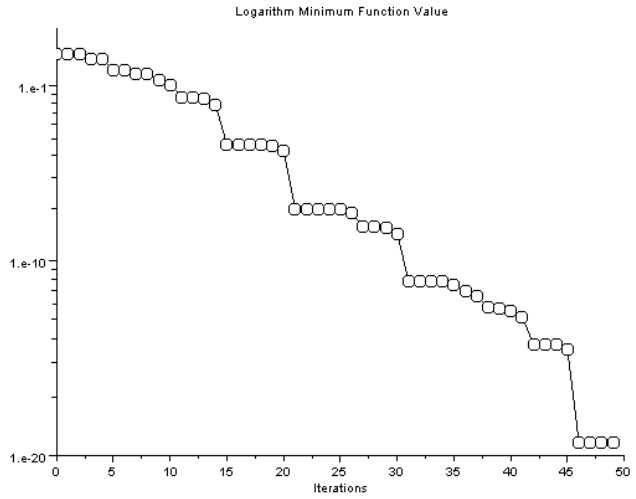

Fig. 3.9 : Spendley et al. numerical experiment – History of logarithm of function

where $a > 0$ is a chosen scaling parameter. The more a is large, the more difficult the problem is to solve with the simplex algorithm. Indeed, let us compute the Hessian matrix associated with the cost function. We have

$$\mathbf{H} = \begin{pmatrix} 2a & 0 \\ 0 & 2 \end{pmatrix}. \tag{3.12}$$

Therefore, the eigenvalues of the Hessian matrix are $2a$ and 2, which are stricly positive if $a > 0$. Hence, the cost function is stricly convex and there is only one global solution, that is $\mathbf{x}^\star = (0, 0, \ldots, 0)^T$. The ratio between these two eigenvalues is a. This leads to an elongated valley, which is extremely narrow when a is large.

The stopping criteria is based on the relative size of the simplex with respect to the size of the initial simplex

$$\sigma_+(S) < tol \times \sigma_+(S_0). \tag{3.13}$$

In this experiment, we use $tol = 10^{-8}$ as the relative tolerance on simplex size.

We set the maximum number of function evaluations to 400. The initial simplex is a regular simplex with length unity.

The following Scilab script allows to perform the optimization.

```
a = 100;
function y = quadratic (x)
  y = a * x(1)^2 + x(2)^2;
endfunction
nm = nmplot_new ();
nm = nmplot_configure(nm,"-numberofvariables",2);
nm = nmplot_configure(nm,"-function",quadratic);
nm = nmplot_configure(nm,"-x0",[10.0 10.0]');
nm = nmplot_configure(nm,"-maxiter",400);
nm = nmplot_configure(nm,"-maxfunevals",400);
```

```
nm = nmplot_configure(nm,"-tolxmethod","disabled");
nm = nmplot_configure(nm,"-tolsimplexizerelative",1.e-8);
nm = nmplot_configure(nm,"-simplex0method","spendley");
nm = nmplot_configure(nm,"-method","fixed");
nm = nmplot_configure(nm,"-verbose",1);
nm = nmplot_configure(nm,"-verbosetermination",0);
nm = nmplot_configure(nm,"-simplexfn","rosenbrock.fixed.history.simplex.txt");
nm = nmplot_configure(nm,"-fbarfn","rosenbrock.fixed.history.fbar.txt");
nm = nmplot_configure(nm,"-foptfn","rosenbrock.fixed.history.fopt.txt");
nm = nmplot_configure(nm,"-sigmafn","rosenbrock.fixed.history.sigma.txt");
nm = nmplot_search(nm);
nmplot_display(nm);
nm = nmplot_destroy(nm);
```

The numerical results are presented in table 3.6, where the experiment is presented for $a = 100$. We can check that the number of function evaluations is equal to its maximum limit, even if the value of the function at optimum is very inaccurate ($f(\mathbf{x}^\star) \approx 0.08$).

Iterations	340
Function Evaluations	400
a	100.0
$\mathbf{x}_0$	$(10.0, 10.0)$
Relative tolerance on simplex size	10^{-8}
Exact $\mathbf{x}^\star$	$(0., 0.)$
Computed $\mathbf{x}^\star$	$(0.001, 0.2)$
Computed $f(\mathbf{x}^\star)$	0.08

Fig. 3.10 : Numerical experiment with Spendley's et al. method on a badly scaled quadratic function

The various simplices generated during the iterations are presented in figure 3.11. The method use reflections in the early iterations. Then there is no possible improvement using reflections, so that shrinking is necessary. But the repeated shrink steps makes the simplex very small, leading to a large number of iterations. This is a limitation of the method, which is based on a simplex which can vary its size, but not its shape.

In figure 3.12, we analyze the behavior of the method with respect to scaling. We check that the method behave poorly when the scaling is bad. The convergence speed is slower and slower and impractical when $a > 10$

3.2.3 Sensitivity to dimension

In this section, we try to study the convergence of the Spendley et al. algorithm with respect to the number of variables, as presented by Han & Neumann in [13]. We emphasize, though, that Han & Neumann present their numerical experiment with the Nelder-Mead algorithm, while we present in this section the Spendley et al. algorithm. The function we try to minimize is the following quadratic function in n-dimensions

$$f(\mathbf{x}) = \sum_{i=1,n} x_i^2. \tag{3.14}$$

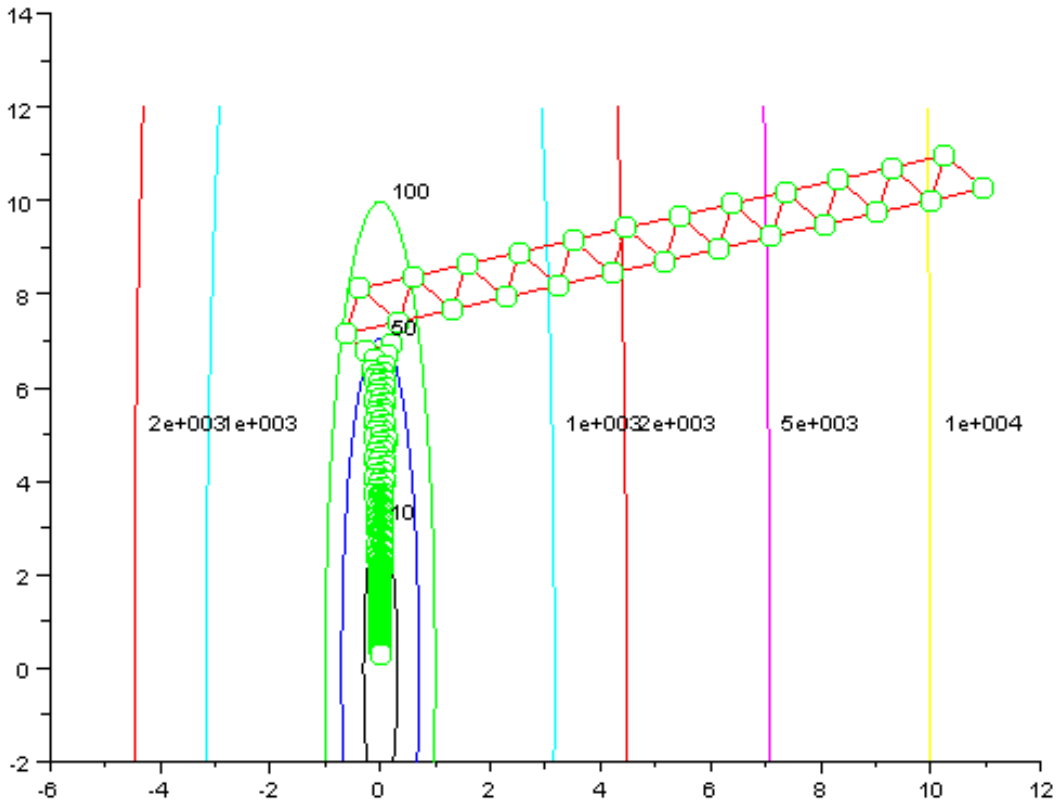

Fig. 3.11 : Spendley et al. numerical experiment with $f(x_1, x_2) = ax_1^2 + x_2^2$ and $a = 100$ – History of simplex

a	Function evaluations	Computed $f(\mathbf{x}^\star)$
1.0	160	$2.35e-18$
10.0	222	$1.2e-17$
100.0	400	0.083
1000.0	400	30.3
10000.0	400	56.08

Fig. 3.12 : Numerical experiment with Spendley's et al. method on a badly scaled quadratic function

The initial guess is the origin $\mathbf{x}_0 = (0, 0, \ldots, 0)^T$, which is also the global solution of the problem. We have $f(\mathbf{x}_0) = 0$ so that this vertex is never updated during the iterations. The initial simplex is computed with a random number generator. The first vertex of the initial simplex is the origin. The other vertices are uniform in the $[-1, 1]$ interval. An absolute termination criteria on the size of the simplex is used, that is, the algorithm is stopped when the inequality $\sigma_+(S_k) \leq 10^{-8}$ is satisfied.

For this test, we compute the rate of convergence as presented in Han & Neuman [13]. This rate is defined as

$$\rho(S_0, n) = \limsup_{k\to\infty} \left(\prod_{i=0,k-1} \frac{\sigma(S_{i+1})}{\sigma(S_i)} \right)^{1/k}, \tag{3.15}$$

where k is the number of iterations. That definition can be viewed as the geometric mean of the ratio of the oriented lengths between successive simplices. This definition implies

$$\rho(S_0, n) = \limsup_{k\to\infty} \left(\frac{\sigma(S_k)}{\sigma(S_0)} \right)^{1/k}, \tag{3.16}$$

If k is the number of iterations required to obtain convergence, as indicated by the termination criteria, the rate of convergence is practically computed as

$$\rho(S_0, n, k) = \left(\frac{\sigma(S_k)}{\sigma(S_0)} \right)^{1/k}. \tag{3.17}$$

The following Scilab script allows to perform the optimization.

```
function y = quadratic (x)
  y = x(:)' * x(:);
endfunction
//
// myoutputcmd --
//   This command is called back by the Nelder-Mead
//   algorithm.
// Arguments
//   state : the current state of the algorithm
//     "init", "iter", "done"
//   data : the data at the current state
//     This is a tlist with the following entries:
//     * x : the optimal vector of parameters
//     * fval : the minimum function value
//     * simplex : the simplex, as a simplex object
//     * iteration : the number of iterations performed
//     * funccount : the number of function evaluations
//     * step : the type of step in the previous iteration
//
function myoutputcmd ( state , data , step )
  global STEP_COUNTER
  STEP_COUNTER(step) = STEP_COUNTER(step) + 1
endfunction

// OptimizeHanNeumann --
//    Perform the optimization and returns the object
// Arguments
//    N : the dimension
function nm = OptimizeHanNeumann ( N )
  global STEP_COUNTER
  STEP_COUNTER("init") = 0;
  STEP_COUNTER("done") = 0;
  STEP_COUNTER("reflection") = 0;
  STEP_COUNTER("expansion") = 0;
  STEP_COUNTER("insidecontraction") = 0;
```

```
  STEP_COUNTER("outsidecontraction") = 0;
  STEP_COUNTER("expansion") = 0;
  STEP_COUNTER("shrink") = 0;
  STEP_COUNTER("reflectionnext") = 0;

  x0 = zeros(N,1);
  nm = neldermead_new ();
  nm = neldermead_configure(nm,"-numberofvariables",N);
  nm = neldermead_configure(nm,"-function",quadratic);
  nm = neldermead_configure(nm,"-x0",x0);
  nm = neldermead_configure(nm,"-maxiter",10000);
  nm = neldermead_configure(nm,"-maxfunevals",10000);
  nm = neldermead_configure(nm,"-tolxmethod","disabled");
  nm = neldermead_configure(nm,"-tolsimplexizeabsolute",1.e-8);
  nm = neldermead_configure(nm,"-tolsimplexizerelative",0);
  nm = neldermead_configure(nm,"-simplex0method","given");
  coords0(1,1:N) = zeros(1,N);
  coords0(2:N+1,1:N) = 2 * rand(N,N) - 1;
  nm = neldermead_configure(nm,"-coords0",coords0);
  nm = neldermead_configure(nm,"-method","fixed");
  nm = neldermead_configure(nm,"-verbose",0);
  nm = neldermead_configure(nm,"-verbosetermination",0);
  nm = neldermead_configure(nm,"-outputcommand",myoutputcmd);
  //
  // Perform optimization
  //
  nm = neldermead_search(nm);
endfunction

for N = 1:10
  nm = OptimizeHanNeumann ( N );
  niter = neldermead_get ( nm , "-iterations" );
  funevals = neldermead_get ( nm , "-funevals" );
  simplex0 = neldermead_get ( nm , "-simplex0" );
  sigma0 = optimsimplex_size ( simplex0 , "sigmaplus" );
  simplexopt = neldermead_get ( nm , "-simplexopt" );
  sigmaopt = optimsimplex_size ( simplexopt , "sigmaplus" );
  rho = ( sigmaopt / sigma0 ) ^ ( 1 / niter );
  //mprintf ( "%d %d %d %e\n" , N , funevals , niter , rho );
  mprintf("%d %s\n",N, strcat(string(STEP_COUNTER)," "))
  nm = neldermead_destroy(nm);
end
```

The figure 3.13 presents the type of steps which are performed for each number of variables. We see that the algorithm mostly performs shrink steps.

The figure 3.14 presents the number of function evaluations depending on the number of variables. We can see that the number of function evaluations increases approximately linearly with the dimension of the problem in figure 3.15. A rough rule of thumb is that, for $n = 1, 20$, the number of function evaluations is equal to $30n$: most iterations are shrink steps and approximately 30 iterations are required, almost independently of n.

The table 3.14 also shows the interesting fact that the convergence rate is almost constant and very close to $1/2$. This is a consequence of the shrink steps, which are dividing the size of the simplex at each iteration by 2.

3.3 Conclusion

We saw in the first numerical experiment that the method behave reasonably when the function is correctly scaled. When the function is badly scaled, as in the second numerical experiment, the Spendley et al. algorithm produces a large number of function evaluations and converges very slowly. This limitation occurs with even moderate badly scaled functions and generates a very slow method in these cases.

n	#Iterations	# Reflections / High	# Reflection / Next to High	#Shrink
1	27	0	0	26
2	28	0	0	27
3	30	2	0	27
4	31	1	1	28
5	29	0	0	28
6	31	2	0	28
7	29	0	0	28
8	29	0	0	28
9	29	0	0	28
10	29	0	0	28
11	29	0	0	28
12	29	0	0	28
13	31	0	2	28
14	29	0	0	28
15	29	0	0	28
16	31	0	1	29
17	30	0	0	29
18	30	0	0	29
19	31	0	1	29
20	32	2	0	29

Fig. 3.13 : Numerical experiment with Spendley et al method on a generalized quadratic function – Number of iterations and types of steps performed

n	Function Evaluations	Iterations	$\rho(S_0, n)$
1	81	27	0.513002
2	112	28	0.512532
3	142	29	0.524482
4	168	28	0.512532
5	206	31	0.534545
6	232	29	0.512095
7	262	30	0.523127
8	292	30	0.523647
9	321	30	0.523647
10	348	29	0.512095
11	377	29	0.512095
12	406	29	0.512095
13	435	29	0.512095
14	464	29	0.512095
15	493	29	0.512095
16	540	30	0.511687
17	570	30	0.511687
18	600	30	0.511687
19	630	30	0.511687
20	660	30	0.511687

Fig. 3.14 : Numerical experiment with Spendley et al. method on a generalized quadratic function

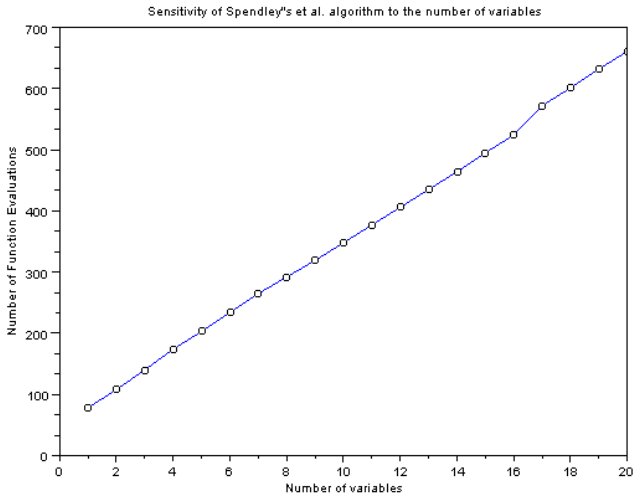

Fig. 3.15 : Spendley et al. numerical experiment – Number of function evaluations depending on the number of variables

In the last experiment, we have explored what happens when the number of iterations is increasing. In this experiment, the rate of convergence is close to $1/2$ and the number of function evaluations is a linear function of the number of variables (approximately $30n$).

Chapter 4

Nelder-Mead method

In this chapter, we present Nelder and Mead's [29] algorithm. We begin by the analysis of the algorithm, which is based on a variable shape simplex. Then, we present geometric situations where the various steps of the algorithm are used. In the third part, we present the rate of convergence toward the optimum of the Nelder-Mead algorithm. This part is mainly based on Han and Neumann's paper [13], which makes use of a class of quadratic functions with a special initial simplex. The core of this chapter is the analysis of several numerical experiments which have been performed with the neldermead component. We analyze the behavior of the algorithm on quadratic functions and present several counter examples where the Nelder-Mead algorithm is known to fail.

4.1 Introduction

In this section, we present the Nelder-Mead algorithm for unconstrained optimization. This algorithm is based on the iterative update of a simplex. Then we present various geometric situations which might occur during the algorithm.

4.1.1 Overview

The goal of the Nelder and Mead algorithm is to solve the following unconstrained optimization problem

$$\min f(\mathbf{x}) \tag{4.1}$$

where $\mathbf{x} \in \mathbb{R}^n$, n is the number of optimization parameters and f is the objective function $f : \mathbb{R}^n \to \mathbb{R}$.

The Nelder-Mead method is an improvement over the Spendley's et al. method with the goal of allowing the simplex to vary in *shape*, and not only in *size*, as in Spendley's et al. algorithm.

This algorithms is based on the iterative update of a *simplex* made of $n+1$ points $S = \{\mathbf{v}_i\}_{i=1,n+1}$. Each point in the simplex is called a *vertex* and is associated with a function value $f_i = f(\mathbf{v}_i)$ for $i = 1, n+1$.

The vertices are sorted by increasing function values so that the *best* vertex has index 1 and the *worst* vertex has index $n+1$

$$f_1 \leq f_2 \leq \ldots \leq f_n \leq f_{n+1}. \tag{4.2}$$

The $\mathbf{v}_1$ vertex (resp. the $\mathbf{v}_{n+1}$ vertex) is called the *best* vertex (resp. *worst*), because it is associated with the lowest (resp. highest) function value.

The centroid of the simplex $\overline{\mathbf{x}}(j)$ is the center of the vertices where the vertex $\mathbf{v}_j$ has been excluded. This centroid is

$$\overline{\mathbf{x}}(j) = \frac{1}{n} \sum_{i=1,n+1,i\neq j} \mathbf{v}_i. \tag{4.3}$$

The algorithm makes use of one coefficient $\rho > 0$, called the reflection factor. The standard value of this coefficient is $\rho = 1$. The algorithm attempts to replace some vertex $\mathbf{v}_j$ by a new vertex $\mathbf{x}(\rho, j)$ on the line from the vertex $\mathbf{v}_j$ to the centroid $\overline{\mathbf{x}}(j)$. The new vertex $\mathbf{x}(\rho, j)$ is defined by

$$\mathbf{x}(\rho, j) = (1+\rho)\overline{\mathbf{x}}(j) - \rho\mathbf{v}_j. \tag{4.4}$$

4.1.2 Algorithm

In this section, we analyze the Nelder-Mead algorithm, which is presented in figure 4.1.

The Nelder-Mead algorithm makes use of four parameters: the coefficient of reflection ρ, expansion χ, contraction γ and shrinkage σ. When the expansion or contraction steps are performed, the shape of the simplex is changed, thus "adapting itself to the local landscape" [29].

These parameters should satisfy the following inequalities [29, 19]

$$\rho > 0, \qquad \chi > 1, \qquad \chi > \rho, \qquad 0 < \gamma < 1 \qquad \text{and} \qquad 0 < \sigma < 1. \tag{4.5}$$

The standard values for these coefficients are

$$\rho = 1, \qquad \chi = 2, \qquad \gamma = \frac{1}{2} \qquad \text{and} \qquad \sigma = \frac{1}{2}. \tag{4.6}$$

In [18], the Nelder-Mead algorithm is presented with other parameter names, that is $\mu_r = \rho$, $\mu_e = \rho\chi$, $\mu_{ic} = -\gamma$ and $\mu_{oc} = \rho\gamma$. These coefficients must satisfy the following inequality

$$-1 < \mu_{ic} < 0 < \mu_{oc} < \mu_r < \mu_e. \tag{4.7}$$

At each iteration, we compute the centroid $\overline{\mathbf{x}}(n+1)$ where the worst vertex $\mathbf{v}_{n+1}$ has been excluded. This centroid is

$$\overline{\mathbf{x}}(n+1) = \frac{1}{n} \sum_{i=1,n} \mathbf{v}_i. \tag{4.8}$$

Compute an initial simplex S_0
Sorts the vertices S_0 with increasing function values
$S \leftarrow S_0$
while $\sigma(S) > tol$ **do**
  $\overline{x} \leftarrow \overline{x}(n+1)$
  $x_r \leftarrow x(\rho, n+1)$ {Reflect}
  $f_r \leftarrow f(x_r)$
  if $f_r < f_1$ **then**
    $x_e \leftarrow x(\rho\chi, n+1)$ {Expand}
    $f_e \leftarrow f(x_e)$
    if $f_e < f_r$ **then**
      Accept x_e
    else
      Accept x_r
    end if
  else if $f_1 \leq f_r < f_n$ **then**
    Accept x_r
  else if $f_n \leq f_r < f_{n+1}$ **then**
    $x_c \leftarrow x(\rho\gamma, n+1)$ {Outside contraction}
    $f_c \leftarrow f(x_c)$
    if $f_c < f_r$ **then**
      Accept x_c
    else
      Compute the points $x_i = x_1 + \sigma(x_i - x_1)$, $i = 2, n+1$ {Shrink}
      Compute $f_i = f(\mathbf{v}_i)$ for $i = 2, n+1$
    end if
  else
    $x_c \leftarrow x(-\gamma, n+1)$ {Inside contraction}
    $f_c \leftarrow f(x_c)$
    if $f_c < f_{n+1}$ **then**
      Accept x_c
    else
      Compute the points $x_i = x_1 + \sigma(x_i - x_1)$, $i = 2, n+1$ {Shrink}
      Compute $f_i = f(\mathbf{v}_i)$ for $i = 2, n+1$
    end if
  end if
  Sort the vertices of S with increasing function values
end while

Fig. 4.1 : Nelder-Mead algorithm – Standard version

We perform a reflection with respect to the worst vertex $\mathbf{v}_{n+1}$, which creates the reflected point $\mathbf{x}_r$ defined by

$$\mathbf{x}_r = \mathbf{x}(\rho, n+1) = (1+\rho)\overline{\mathbf{x}}(n+1) - \rho\mathbf{v}_{n+1} \tag{4.9}$$

We then compute the function value of the reflected point as $f_r = f(\mathbf{x}_r)$.

From that point, there are several possibilities, which are listed below. Most steps try to replace the worst vertex $\mathbf{v}_{n+1}$ by a better point, which is computed depending on the context.

- In the case where $f_r < f_1$, the reflected point $\mathbf{x}_r$ were able to improve (i.e. reduce) the function value. In that case, the algorithm tries to expand the simplex so that the function value is improved even more. The expansion point is computed by

  $$\mathbf{x}_e = \mathbf{x}(\rho\chi, n+1) = (1+\rho\chi)\overline{\mathbf{x}}(n+1) - \rho\chi\mathbf{v}_{n+1} \tag{4.10}$$

  and the function is computed at this point, i.e. we compute $f_e = f(\mathbf{x}_e)$. If the expansion point allows to improve the function value, the worst vertex $\mathbf{v}_{n+1}$ is rejected from the simplex and the expansion point $\mathbf{x}_e$ is accepted. If not, the reflection point $\mathbf{x}_r$ is accepted.

- In the case where $f_1 \leq f_r < f_n$, the worst vertex $\mathbf{v}_{n+1}$ is rejected from the simplex and the reflected point $\mathbf{x}_r$ is accepted.

- In the case where $f_n \leq f_r < f_{n+1}$, we consider the point

  $$\mathbf{x}_c = \mathbf{x}(\rho\gamma, n+1) = (1+\rho\gamma)\overline{\mathbf{x}}(n+1) - \rho\gamma\mathbf{v}_{n+1} \tag{4.11}$$

  is considered. If the point $\mathbf{x}_c$ is better than the reflection point $\mathbf{x}_r$, then it is accepted. If not, a shrink step is performed, where all vertices are moved toward the best vertex $\mathbf{v}_1$.

- In other cases, we consider the point

  $$\mathbf{x}_c = \mathbf{x}(-\gamma, n+1) = (1-\gamma)\overline{\mathbf{x}}(n+1) + \gamma\mathbf{v}_{n+1}. \tag{4.12}$$

  If the point $\mathbf{x}_c$ is better than the worst vertex $\mathbf{x}_{n+1}$, then it is accepted. If not, a shrink step is performed.

The algorithm from figure 4.1 is the most popular variant of the Nelder-Mead algorithm. But the original paper is based on a "greedy" expansion, where the expansion point is accepted if it is better than the best point (and not if it is better than the reflection point). This "greedy" version is implemented in AS47 by O'Neill in [31] and the corresponding algorithm is presented in figure 4.2.

```
[...]

  x_e ← x(ρχ, n + 1) {Expand}
  f_e ← f(x_e)
  if f_e < f_1 then
    Accept x_e
  else
    Accept x_r
  end if

[...]
```

Fig. 4.2 : Nelder-Mead algorithm – Greedy version

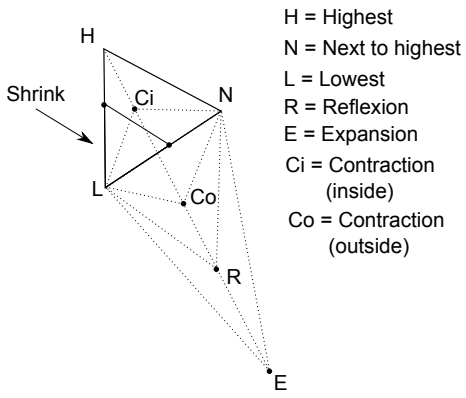

Fig. 4.3 : Nelder-Mead simplex steps

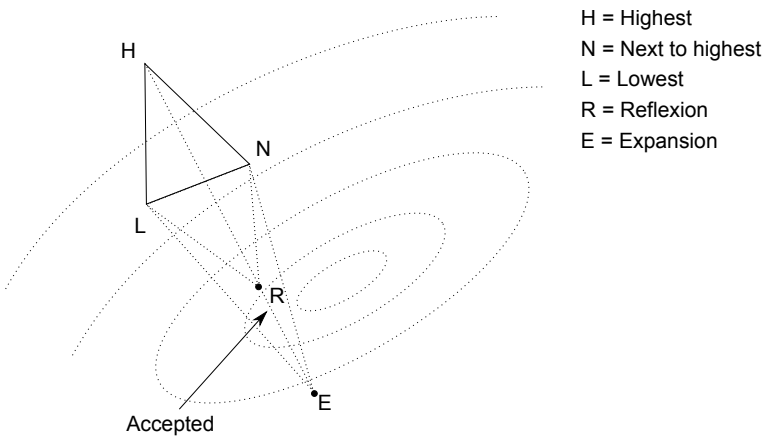

Fig. 4.4 : Nelder-Mead simplex moves – Reflection

4.2 Geometric analysis

The figure 4.3 presents the various moves of the simplex in the Nelder-Mead algorithm.

The figures 4.4 to 4.9 present the detailed situations when each type of step occur. We emphasize that these figures are not the result of numerical experiments. These figures been created in order to illustrate the following specific points of the algorithm.

- Obviously, the expansion step is performed when the simplex is far away from the optimum. The direction of descent is then followed and the worst vertex is moved into that direction.
- When the reflection step is performed, the simplex is getting close to an valley, since the expansion point does not improve the function value.
- When the simplex is near the optimum, the inside and outside contraction steps may be performed, which allows to decrease the size of the simplex. The figure 4.6, which illustrates the inside contraction step, happens in "good" situations. As presented in section 4.5.4, applying repeatedly the inside contraction step can transform the simplex into a degenerate simplex, which may let the algorithm converge to a non stationnary point.
- The shrink steps (be it after an outside contraction or an inside contraction) occurs only in very special situations. In practical experiments, shrink steps are rare.

4.3 Automatic restarts

In this section, we describe an algorithm which enables the user to perform automatic restarts when a search has failed. A condition is used to detect that a false minimum has been reached. We describe the automatic restart algorithm as well as the conditions used to detect a false minimum.

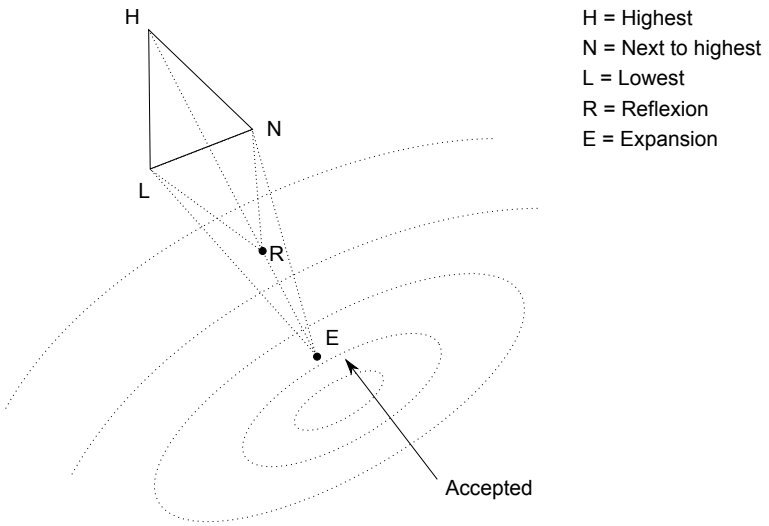

Fig. 4.5 : Nelder-Mead simplex moves – Expansion

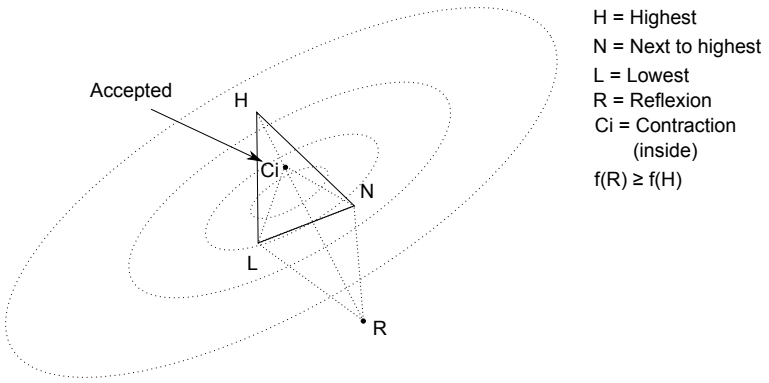

Fig. 4.6 : Nelder-Mead simplex moves - Inside contraction

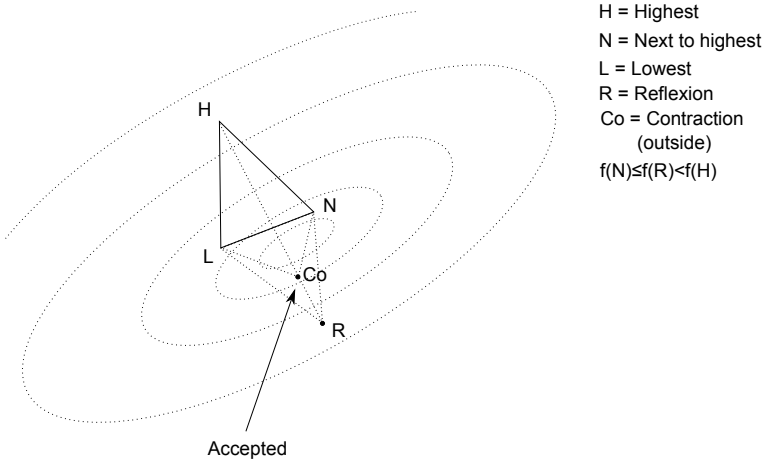

Fig. 4.7 : Nelder-Mead simplex moves – Outside contraction

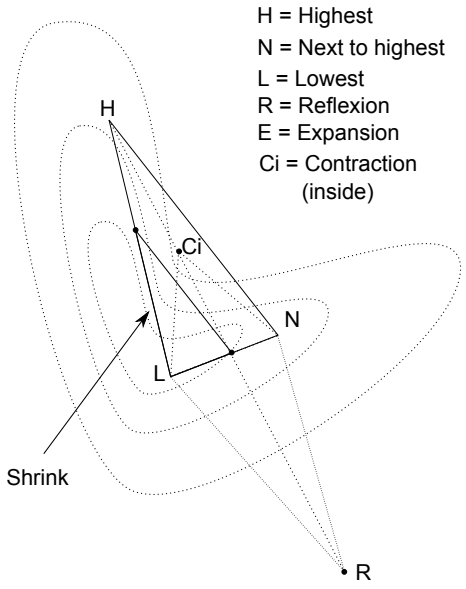

Fig. 4.8 : Nelder-Mead simplex moves – Shrink after inside contraction.

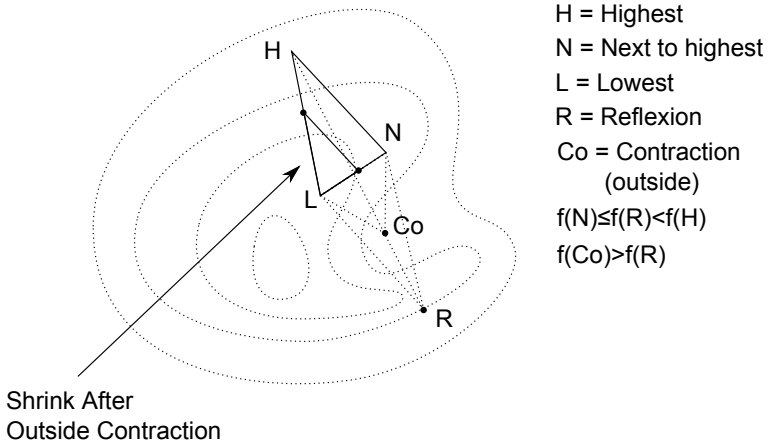

Fig. 4.9 : Nelder-Mead simplex moves – Shrink after outside contraction

4.3.1 Automatic restart algorithm

In this section, we present the automatic restart algorithm.

The goal of this algorithm is to detect that a false minimum has been found, a situation which may occur with the Nelder-Mead algorithm, as we are going to see in the numerical experiments section. These problems are known by practitionners since decades and several authors have tried to detect and solve this specific problem.

In 1971, O'Neill published a fortran 77 implementation of the Nelder-Mead algorithm [31]. In order to check that the algorithm has converged, a factorial test is used. This test will be detailed later in this section. If a false minimum is found by this test, O'Neill suggests to restart the algorithm.

In 1998, Mc Kinnon [21] showed a simple objective function for which the Nelder-Mead algorithm fails to converge to a minimum and, instead, converge to a non-stationnary point. In this numerical experiment, the simplex degenerates toward a single point. In 1999, Kelley [17] shows that restarting the algorithm allows to converge toward the global minimum. In order to detect the convergence problem, Kelley adapted the sufficient decrease condition which is classical in the frameword of gradient-based algorithms. When this condition is met, the algorithm is stopped and a restart should be performed.

Scilab provides an automatic restart algorithm, which allows to detect that a false optimum has been reached and that a new search must be performed. The algorithm is based on a loop where a maximum number of restarts is allowed. The default maximum number of restarts is 3, which means that the maximum number of searches is 4.

After a search has been performed, a condition is computed to know whether a restart must be performed. There are two conditions which are implemented:

- O'Neill factorial test,
- Kelley's stagnation condition.

We will analyze these tests later in this section.

Notice that the automatic restarts are available whatever the simplex algorithm, be it the Nelder-Mead variable shape simplex algorithm, Spendley's et al. fixed shape simplex algorithm or any other algorithm. This is because the automatic restart is a loop programmed above the optimization algorithm.

The automatic restart algorithm is presented in 4.10. Notice that, if a false minimum is detected after the maximum number of restart has been reached, the status is set to "maxrestart".

4.3.2 O'Neill factorial test

In this sectin, we present O'Neill's factorial test. This algorithm is given a vector of lengths, stored in the *step* variable. It is also given a small value ϵ, which is an step length relative to the *step* variable. The algorithm is presented in figure 4.11.

$restartnb \leftarrow 0$
$reached \leftarrow FALSE$
for $i = 1$ to $restartmax + 1$ **do**
  $search()$
  $istorestart = istorestart()$
  if $NOT(istorestart)$ **then**
    $reached \leftarrow TRUE$ {Convergence}
    $BREAK$
  end if
  if $i < restartmax$ **then**
    $restartnb \leftarrow restartnb + 1$ {A restart is going to be performed}
  end if
end for
if $reached$ **then**
  printf ("Convergence reached after
else
  printf ("Convergence not reached after maximum
  $status \leftarrow "maxrestart"$
end if

Fig. 4.10 : Nelder-Mead algorithm – Automatic restart algorithm.

$\mathbf{x} \leftarrow \mathbf{x}^{\star}$
$istorestart = FALSE$
for $i = 1$ to n **do**
  $\delta = step(i) * \epsilon$
  if $\delta == 0.0$ **then**
    $\delta = \epsilon$
  end if
  $x(i) = x(i) + \delta$
  $fv = f(x)$
  if $fv < fopt$ **then**
    $istorestart = TRUE$
    break
  end if
  $x(i) = x(i) - \delta - \delta$
  $fv = f(x)$
  if $fv < fopt$ **then**
    $istorestart = TRUE$
    break
  end if
  $x(i) = x(i) + \delta$
end for

Fig. 4.11 : O'Neill's factorial test

O'Neill's factorial test requires a large number of function evaluations, namely 2^n function evaluations. In O'Neill's implementation, the parameter ϵ is set to the constant value $1.e-3$. In Scilab's implementation, this parameter can be customized, thanks to the *-restarteps* option. Its default value is *%eps*, the machine epsilon. In O'Neill's implementation, the parameter *step* is equal to the vector of length used in order to compute the initial simplex. In Scilab's implementation, the two parameters are different, and the *step* used in the factorial test can be customized with the *-restartstep* option. Its default value is 1.0, which is expanded into a vector with size n.

4.3.3 Kelley's stagnation detection

In this section, we present Kelley's stagnation detection, which is based on the simplex gradient, which definition has been presented in chapter 2.

C.T. Kelley described in [17] a method to detect stagnation of Nelder-Mead's algorithm. In order to detect the convergence problem, Kelley adapted the sufficient decrease condition which is classical in the frameword of gradient-based algorithms. When this condition is met, the algorithm is stopped and a restart should be performed. We first present the sufficient decrease condition in the context of line search methods. We then present the stagnation condition and a variant of this condition.

Line search and sufficient decrease condition

Before presenting the stagnation criteria suggested by Kelley, it is worthwhile to consider a general gradient-based optimization algorithm and to analyse the way to compute the step length.

Consider an optimization algorithm where the update of the current point $\mathbf{x}^k \in \mathbb{R}^n$ is based on the iteration

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{4.13}$$

where $\mathbf{p}_k \in \mathbb{R}^n$ is the direction and $\alpha_k > 0$ is the step length. Assume that the direction $\mathbf{p}_k$ is given and that α_k is unknown. The problem is to find the minimizer of the one dimensional function Φ defined by the equality

$$\Phi(\alpha) = f(\mathbf{x}_k + \alpha p_k), \tag{4.14}$$

for all $\alpha > 0$.

During the computation of the step length α, there is a tradeoff between reducing sufficiently the function value and not spending too much time in doing so. Line search methods aims at providing an efficient solution for this problem. Several algorithms can be designed in order to find such an optimal α, but all rely on a set of conditions which allows to know when to stop the algorithm. Many line search algorithms are based on the Goldstein-Armijo condition [15, 9], which requires that

$$f(\mathbf{x}_k + \alpha p_k) \leq f(\mathbf{x}_k) + c\alpha \nabla f_k^T \mathbf{p}_k, \tag{4.15}$$

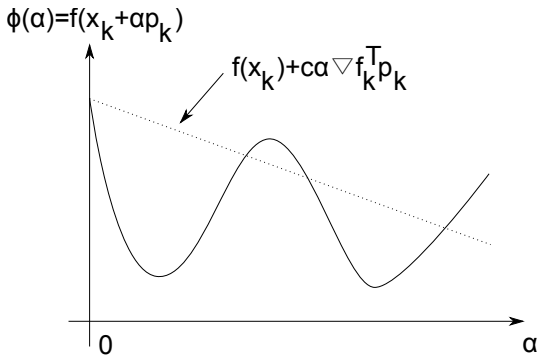

Fig. 4.12 : Sufficient decrease condition

where $c \in (0,1)$ is a given parameter. This condition is presented in figure 4.12. The term $f_k^T \mathbf{p}_k$ is the directionnal derivative of the objective function f along the direction $\mathbf{p}_k$. The Goldstein-Armijo condition ensures that the step length is not too large by requiring that the reduction in f be proportional to the step length α and the directional derivative $f_k^T \mathbf{p}_k$. In practice, the parameter c is often chosen as $c = 10^{-4}$. This implies that the line $f(\mathbf{x}_k) + c\alpha \nabla f_k^T \mathbf{p}_k$ has a slightly decreasing slope, i.e. the condition is rather loose and accept many values of α.

In many line search methods, the Goldstein-Armijo condition is used in combination with another condition, which ensures that the step length α is not too small. This is the additionnal requirement of the Wolfe conditions, also called the curvature condition. We will not detail this further, because the curvature condition is not used in Kelley's stagnation detection criteria.

Stagnation criteria

Let us denote by S_k the simplex at iteration k. We make the assumption that the initial simplex S^0 is nondegenerate, i.e. the condition number of the matrix of simplex directions $\kappa(D(S))$ is finite. We denote by $k \geq 0$ the index of the current iteration. Let us denote by f_1^k the function value at the best vertex $\mathbf{v}_1^{(k)}$, i.e. $f_1^k = f\left(\mathbf{v}_1^{(k)}\right)$.

The derivation is based on the following assumptions.

Assumption 4.3.1 *For all iterations k,*

- *the simplex S_k is nondegenerate,*
- *the vertices are ordered by increasing function value, i.e.*

$$f_1^k \leq f_2^k \leq \ldots \leq f_{n+1}^k, \tag{4.16}$$

- *the best function value is strictly decreasing, i.e. $f_1^{k+1} < f_1^k$.*

If no shrink step occurs in the Nelder-Mead algorithm, then the best function value is indeed decreasing.

Kelley defines a sufficient decrease condition which is analalogous to the sufficient decrease condition for gradient-base algorithms. This condition requires that the $k+1$st iteration satisfies

$$f_1^{k+1} - f_1^k < -c\|\overline{\mathbf{g}}(S_k)\|^2, \tag{4.17}$$

where $\overline{\mathbf{g}}(S_k)$ is the simplex gradient associated with the simplex S_k and $c > 0$ is a small parameter. A typical choice in line-search methods is $c = 10^{-4}$. Kelley suggest in [17] to use 4.17 as a test to detect the stagnation of the Nelder-Mead algorithm.

For consistency, we reproduce below a proposition already presented in chapter 2.

Proposition 4.3.2 *Let S be a simplex. Let the gradient $\mathbf{g}$ be Lipshitz continuous in a neighbourhood of S with Lipshitz constant L. Consider the euclidian norm $\|.\|$. Then, there is a constant $K > 0$, depending only on L such that*

$$\|\mathbf{g}(\mathbf{v}_1) - \overline{\mathbf{g}}(S)\|_2 \leq K\kappa(S)\sigma_+(S). \tag{4.18}$$

The stagnation detection criteria is based on the following proposition.

Proposition 4.3.3 *Let a sequence of simplices $\{S_k\}_{k\geq 0}$ satisfy assumption 4.3.1. Assume that the sequence $\{f_1^k\}_{k\geq 0}$ is bounded from below. Let the gradient $\mathbf{g}$ of the objective function be Lipshitz continuous in a neighbourhood of $\{S_k\}_{k\geq 0}$ with Lipshitz constant L. Assume that the constant K_k, defined in proposition 4.3.2 is bounded. Assume that the sufficient decrease condition 4.17 is satisfied and that the simplices are so that*

$$\lim_{k\to\infty} \kappa(S_k)\sigma_+(S_k) = 0. \tag{4.19}$$

Therefore, if the best vertex in the simplices converges towards $\mathbf{v}_1^\star$, then $\mathbf{g}(\mathbf{v}_1^\star) = 0$.

Essentially, the proposition states that the condition 4.17 is necessary to get the convergence of the algorithm towards a stationnary point.

Notice that, since the simplex condition number $\kappa(S_k)$ satisfies $\kappa(S_k) \geq 1$, then the the equality 4.19 implies that the size of the simplices converges towards 0.

Proof We first proove that the sequence of simplex gradients $\{\overline{\mathbf{g}}(S_k)\}_{k\geq 0}$ converges toward 0. Notice that the sufficient decrease condition 4.17 can be written as

$$\|\overline{\mathbf{g}}(S_k)\| < \frac{1}{\sqrt{c}}\sqrt{f_1^k - f_1^{k+1}}, \tag{4.20}$$

where the right hand side is positive, by the assumption 4.3.1. By hypothesis, f is uniformly bounded from below and the sequence $\{f_1^k\}_{k\geq 0}$ is stricly decreasing by assumption 4.3.1. Therefore, the sequence $\{f_1^k\}_{k\geq 0}$ converges, which implies that the the sequence $\{f_1^k - f_1^{k+1}\}_{k\geq 0}$ converges to 0. Hence, the inequality 4.20 implies that the sequence $\{\overline{\mathbf{g}}(S_k)\}_{k\geq 0}$ converges towards 0.

Assume that $\mathbf{v}_1^\star$ is an accumulation point of the best vertex of the simplices. We now proove that $\mathbf{v}_1^\star$ is a critical point of the objective function, i.e. we proove that the sequence $\{\mathbf{g}(\mathbf{v}_1^k)\}_{k\geq 0}$ converges towards 0. Notice that we can write the gradient as the sum

$$\mathbf{g}(\mathbf{v}_1^k) = \big(\mathbf{g}(\mathbf{v}_1^k) - \overline{\mathbf{g}}(S_k)\big) + \overline{\mathbf{g}}(S_k), \tag{4.21}$$

which implies

$$\|\mathbf{g}(\mathbf{v}_1^k)\| \leq \|\mathbf{g}(\mathbf{v}_1^k) - \overline{\mathbf{g}}(S_k)\| + \|\overline{\mathbf{g}}(S_k)\|. \tag{4.22}$$

By proposition 4.3.2, there is a constant $K_k > 0$, depending on L and k, such that

$$\|\mathbf{g}(\mathbf{v}_1^k) - \overline{\mathbf{g}}(S_k)\|_2 \leq K_k \kappa(S_k)\sigma_+(S_k). \tag{4.23}$$

By hypothesis, the sequence $\{K_k\}_{k\geq 0}$ is bounded, so that there exists a $K > 0$ so that the inequality $K_k \leq K$, which implies

$$\|\mathbf{g}(\mathbf{v}_1^k) - \overline{\mathbf{g}}(S_k)\|_2 \leq K \kappa(S_k)\sigma_+(S_k). \tag{4.24}$$

We plug the previous inequality into 4.22 and get

$$\|\mathbf{g}(\mathbf{v}_1^k)\| \leq K \kappa(S_k)\sigma_+(S_k) + \|\overline{\mathbf{g}}(S_k)\|. \tag{4.25}$$

We have already prooved that the sequence $\{\overline{\mathbf{g}}(S_k)\}_{k\geq 0}$ converges towards 0. Moreover, by hypothesis, the sequence $\{\kappa(S_k)\sigma_+(S_k)\}_{k\geq 0}$ converges towards 0. Therefore, we have

$$\lim_{k\to\infty} \mathbf{g}(\mathbf{v}_1^k) = 0, \tag{4.26}$$

which concludes the proof. ∎

Kelley also states a similar theorem which involves noisy functions. These functions are of the form

$$f(\mathbf{x}) = \tilde{f}(\mathbf{x}) + \phi(\mathbf{x}), \tag{4.27}$$

where $\tilde{f}$ is smooth and ϕ is a bounded low-amplitude perturbation. The result is that, if the noise function ϕ has a magnitude smaller than $\sigma_+(S)$, then the proposition 4.3.3 still holds.

A variant of the stagnation criteria

In his book [18], C.T. Kelley suggest a slightly different form for the stagnation criteria 4.17. This variant is based on the fact that the Armijo-Goldstein condition

$$f(\mathbf{x}_k + \alpha p_k) \leq f(\mathbf{x}_k) + c\alpha \nabla f_k^T \mathbf{p}_k, \tag{4.28}$$

distinguish the parameter $c = 10^{-4}$ and the step length $\alpha_k > 0$. In the simplex algorithm, there is no such step length, so that the step length α must be incorporated into the parameter c, which leads to the condition

$$f_1^{k+1} - f_1^k < -c\|\overline{\mathbf{g}}(S_k)\|^2, \tag{4.29}$$

with $c = 10^{-4}$. Now, at the first iteration, the simplex diameter $\sigma_+(S_0)$ might be much smaller that the simplex gradient $\|\overline{\mathbf{g}}(S_k)\|$ so that the previous condition may fail. Kelley address this problem by modifying the previous condition into

$$f_1^{k+1} - f_1^k < -c\frac{\sigma_+(S_0)}{\|\overline{\mathbf{g}}(S_0)\|}\|\overline{\mathbf{g}}(S_k)\|^2. \tag{4.30}$$

4.4 Convergence properties on a quadratic

In this section, we reproduce one result presented by Han and Neumann [13], which states the rate of convergence toward the optimum on a class of quadratic functions with a special initial simplex. Some additional results are also presented in the Phd thesis by Lixing Han [12]. We study a generalized quadratic and use a particular initial simplex. We show that the vertices follow a recurrence equation, which is associated with a characteristic equation. The study of the roots of these characteristic equations give an insight of the behavior of the Nelder-Mead algorithm when the dimension n increases.

Let us suppose than we want to minimize the function

$$f(\mathbf{x}) = x_1^2 + \ldots + x_n^2 \tag{4.31}$$

with the initial simplex

$$S_0 = \left[\mathbf{0}, \mathbf{v}_1^{(0)}, \ldots, \mathbf{v}_n^{(0)}\right] \tag{4.32}$$

With this choice of the initial simplex, the best vertex remains fixed at $\mathbf{0} = (0, 0, \ldots, 0)^T \in \mathbb{R}^n$. As the cost function 4.31 is strictly convex, the Nelder-Mead method never performs the *shrink* step. Therefore, at each iteration, a new simplex is formed by replacing the worst vertex $\mathbf{v}_n^{(k)}$, by a new, better vertex. Assume that the Nelder-Mead method generates a sequence of simplices $\{S_k\}_{k\geq 0}$ in $\mathbb{R}^n$, where

$$S_k = \left[\mathbf{0}, \mathbf{v}_1^{(k)}, \ldots, \mathbf{v}_n^{(n)}\right] \tag{4.33}$$

We wish that the sequence of simplices $S_k \to \mathbf{0} \in \mathbb{R}^n$ as $k \to \infty$. To measure the progress of convergence, Han and Neumann use the oriented length $\sigma_+(S_k)$ of the simplex S_k, defined by

$$\sigma_+(S) = \max_{i=2,m} \|\mathbf{v}_i - \mathbf{v}_1\|_2. \tag{4.34}$$

We say that a sequence of simplices $\{S_k\}_{k\geq 0}$ converges to the minimizer $\mathbf{0} \in \mathbb{R}^n$ of the function in equation 4.31 if $\lim_{k\to\infty} \sigma_+(S_k) = 0$.

We measure the rate of convergence defined by

$$\rho(S_0, n) = \limsup_{k\to\infty} \left(\sum_{i=0,k-1} \frac{\sigma(S_{i+1})}{\sigma(S_i)} \right)^{1/k}. \tag{4.35}$$

That definition can be viewed as the geometric mean of the ratio of the oriented lengths between successive simplices and the minimizer 0. This definition implies

$$\rho(S_0, n) = \limsup_{k\to\infty} \left(\frac{\sigma(S_{k+1})}{\sigma(S_0)} \right)^{1/k}. \tag{4.36}$$

According to the definition, the algorithm is convergent if $\rho(S_0, n) < 1$. The larger the $\rho(S_0, n)$, the slower the convergence. In particular, the convergence is very slow when $\rho(S_0, n)$ is close to 1. The analysis is based on the fact that the Nelder-Mead method generates a sequence of simplices in $\mathbb{R}^n$ satisfying

$$S_k = \left[\mathbf{0}, \mathbf{v}^{(k+n-1)}, \ldots, \mathbf{v}^{(k+1)}, \mathbf{v}^{(k)}\right], \tag{4.37}$$

where $\mathbf{0}, \mathbf{v}^{(k+n-1)}, \ldots, \mathbf{v}^{(k+1)}, \mathbf{v}^{(k)} \in \mathbb{R}^n$ are the vertices of the $k-th$ simplex, with

$$f(\mathbf{0}) < f\left(\mathbf{v}^{(k+n-1)}\right) < f\left(\mathbf{v}^{(k+1)}\right) < f\left(\mathbf{v}^{(k)}\right), \tag{4.38}$$

for $k \geq 0$.

To simplify the analysis, we consider that only one type of step of the Nelder-Mead method is applied repeatedly. This allows to establish recurrence equations for the successive simplex vertices. As the shrink step is never used, and the expansion steps is never used neither (since the best vertex is already at 0), the analysis focuses on the outside contraction, inside contraction and reflection steps.

The centroid of the n best vertices of S_k is given by

$$\begin{aligned}
\overline{\mathbf{v}}^{(k)} &= \frac{1}{n}\left(\mathbf{v}^{(k+1)} + \ldots + \mathbf{v}^{(k+n-1)} + \mathbf{0}\right) && (4.39)\\
&= \frac{1}{n}\left(\mathbf{v}^{(k+1)} + \ldots + \mathbf{v}^{(k+n-1)}\right) && (4.40)\\
&= \frac{1}{n}\sum_{i=1,n-1} \mathbf{v}^{(k+i)} && (4.41)
\end{aligned}$$

4.4.1 With default parameters

In this section, we analyze the roots of the characteristic equation with *fixed*, standard inside and outside contraction coefficients.

Outside contraction

If the outside contraction step is repeatedly performed with $\mu_{oc} = \rho\gamma = \frac{1}{2}$, then

$$\mathbf{v}^{(k+n)} = \overline{\mathbf{v}}^{(k)} + \frac{1}{2}\left(\overline{\mathbf{v}}^{(k)} - \mathbf{v}^{(k)}\right). \tag{4.42}$$

By plugging the definition of the centroid 4.41 into the previous equality, we find the recurrence formula

$$2n\mathbf{v}^{(k+n)} - 3\mathbf{v}^{(k+1)} - \ldots - 3\mathbf{v}^{(k+n-1)} + n\mathbf{v}^{(k)} = 0. \tag{4.43}$$

The associated characteristic equation is

$$2n\mu^n - 3\mu^{n-1} - \ldots - 3\mu + n = 0. \tag{4.44}$$

Inside contraction

If the inside contraction step is repeatedly performed with $\mu_{ic} = -\gamma = -\frac{1}{2}$, then

$$\mathbf{v}^{(k+n)} = \overline{\mathbf{v}}^{(k)} - \frac{1}{2}\left(\overline{\mathbf{v}}^{(k)} - \mathbf{v}^{(k)}\right). \tag{4.45}$$

By plugging the definition of the centroid 4.41 into the previous equality, we find the recurrence formula

$$2n\mathbf{v}^{(k+n)} - \mathbf{v}^{(k+1)} - \ldots - \mathbf{v}^{(k+n-1)} - n\mathbf{v}^{(k)} = 0. \tag{4.46}$$

The associated characteristic equation is

$$2n\mu^n - \mu^{n-1} - \ldots - \mu - n = 0. \tag{4.47}$$

Reflection

If the reflection step is repeatedly performed with $\mu_r = \rho = 1$, then

$$\mathbf{v}^{(k+n)} = \overline{\mathbf{v}}^{(k)} + \left(\overline{\mathbf{v}}^{(k)} - \mathbf{v}^{(k)}\right). \tag{4.48}$$

By plugging the definition of the centroid 4.41 into the previous equality, we find the recurrence formula

$$n\mathbf{v}^{(k+n)} - 2\mathbf{v}^{(k+1)} - \ldots - 2\mathbf{v}^{(k+n-1)} + n\mathbf{v}^{(k)} = 0. \tag{4.49}$$

The associated characteristic equation is

$$n\mu^n - 2\mu^{n-1} - \ldots - 2\mu + n = 0. \tag{4.50}$$

The recurrence equations 4.44, 4.47 and 4.50 are linear. Their general solutions are of the form

$$\mathbf{v}^{(k)} = \mu_1^k \mathbf{a}_1 + \ldots + \mu_n^k \mathbf{a}_n, \tag{4.51}$$

where $\{\mu_i\}_{i=1,n}$ are the roots of the characteristic equations and $\{\mathbf{a}_i\}_{i=1,n} \in \mathbb{C}^n$ are independent vectors such that $\mathbf{v}^{(k)} \in \mathbb{R}^n$ for all $k \geq 0$.

The analysis by Han and Neumann [13] gives a deep understanding of the convergence rate for this particular situation. For $n = 1$, they show that the convergence rate is $\frac{1}{2}$. For $n = 2$, the convergence rate is $\frac{\sqrt{2}}{2} \approx 0.7$ with a particular choice for the initial simplex. For $n \geq 3$, Han and Neumann [13] perform a numerical analysis of the roots.

In the following Scilab script, we compute the roots of these 3 characteristic equations.

```
//
// computeroots1 --
//   Compute the roots of the characteristic equations of
//   usual Nelder-Mead method.
//
function computeroots1 ( n )
  // Polynomial for outside contraction :
  // n - 3x - ... - 3x^(n-1) + 2n x^(n) = 0
  mprintf("Polynomial for outside contraction :\n");
  coeffs = zeros(1,n+1);
  coeffs(1) = n
  coeffs(2:n) = -3
  coeffs(n+1) = 2 * n
  p=poly(coeffs,"x","coeff")
  disp(p)
  mprintf("Roots :\n");
  r = roots(p)
  for i=1:n
    mprintf("Root #%d/%d |%s|=%f\n", i, length(r),string(r(i)),abs(r(i)))
  end
  // Polynomial for inside contraction :
  // - n - x - ... - x^(n-1) + 2n x^(n)= 0
  mprintf("Polynomial for inside contraction :\n");
  coeffs = zeros(1,n+1);
  coeffs(1) = -n
  coeffs(2:n) = -1
  coeffs(n+1) = 2 * n
  p=poly(coeffs,"x","coeff")
  disp(p)
  mprintf("Roots :\n");
  r = roots(p)
  for i=1:n
    mprintf("Root #%d/%d |%s|=%f\n", i, length(r),string(r(i)),abs(r(i)))
  end
  // Polynomial for reflection :
  // n - 2x - ... - 2x^(n-1) + n x^(n) = 0
  mprintf("Polynomial for reflection :\n");
  coeffs = zeros(1,n+1);
  coeffs(1) = n
  coeffs(2:n) = -2
  coeffs(n+1) = n
  p=poly(coeffs,"x","coeff")
  disp(p)
  r = roots(p)
  mprintf("Roots :\n");
  for i=1:n
    mprintf("Root #%d/%d |%s|=%f\n", i, length(r),string(r(i)),abs(r(i)))
  end
endfunction
```

If we execute the previous script with $n = 10$, the following output is produced.

```
-->computeroots1 ( 10 )
Polynomial for outside contraction :

                2    3    4    5    6    7    8     9    10
   10 - 3x - 3x - 3x - 3x - 3x - 3x - 3x - 3x - 3x + 20x
Roots :
Root #1/10 |0.5822700+%i*0.7362568|=0.938676
```

```
Root #2/10 |0.5822700-%i*0.7362568|=0.938676
Root #3/10 |-0.5439060+%i*0.7651230|=0.938747
Root #4/10 |-0.5439060-%i*0.7651230|=0.938747
Root #5/10 |0.9093766+%i*0.0471756|=0.910599
Root #6/10 |0.9093766-%i*0.0471756|=0.910599
Root #7/10 |0.0191306+%i*0.9385387|=0.938734
Root #8/10 |0.0191306-%i*0.9385387|=0.938734
Root #9/10 |-0.8918713+%i*0.2929516|=0.938752
Root #10/10 |-0.8918713-%i*0.2929516|=0.938752
Polynomial for inside contraction :

              2   3   4   5   6   7   8    9    10
  - 10 - x - x - x - x - x - x - x - x - x + 20x
Roots :
Root #1/10 |0.7461586+%i*0.5514088|=0.927795
Root #2/10 |0.7461586-%i*0.5514088|=0.927795
Root #3/10 |-0.2879931+%i*0.8802612|=0.926175
Root #4/10 |-0.2879931-%i*0.8802612|=0.926175
Root #5/10 |-0.9260704|=0.926070
Root #6/10 |0.9933286|=0.993329
Root #7/10 |0.2829249+%i*0.8821821|=0.926440
Root #8/10 |0.2829249-%i*0.8821821|=0.926440
Root #9/10 |-0.7497195+%i*0.5436596|=0.926091
Root #10/10 |-0.7497195-%i*0.5436596|=0.926091
Polynomial for reflection :

                2    3    4    5    6    7    8     9    10
    10 - 2x - 2x - 2x - 2x - 2x - 2x - 2x - 2x - 2x + 10x
Roots :
Root #1/10 |0.6172695+%i*0.7867517|=1.000000
Root #2/10 |0.6172695-%i*0.7867517|=1.000000
Root #3/10 |-0.5801834+%i*0.8144859|=1.000000
Root #4/10 |-0.5801834-%i*0.8144859|=1.000000
Root #5/10 |0.9946011+%i*0.1037722|=1.000000
Root #6/10 |0.9946011-%i*0.1037722|=1.000000
Root #7/10 |0.0184670+%i*0.9998295|=1.000000
Root #8/10 |0.0184670-%i*0.9998295|=1.000000
Root #9/10 |-0.9501543+%i*0.3117800|=1.000000
Root #10/10 |-0.9501543-%i*0.3117800|=1.000000
```

The following Scilab script allows to compute the minimum and the maximum of the modulus of the roots. The "e" option of the "roots" command has been used to force the use of the

eigenvalues of the companion matrix as the computational method. The default algorithm, based on the Jenkins-Traub Rpoly method is generating a convergence error and cannot be used in this case.

```
function [rminoc , rmaxoc , rminic , rmaxic] = computeroots1_abstract ( n )
  // Polynomial for outside contraction :
  // n - 3x - ... - 3x^(n-1) + 2n x^(n) = 0
  coeffs = zeros(1,n+1);
  coeffs(1) = n
  coeffs(2:n) = -3
  coeffs(n+1) = 2 * n
  p=poly(coeffs,"x","coeff")
  r = roots(p , "e")
  rminoc = min(abs(r))
  rmaxoc = max(abs(r))
  // Polynomial for inside contraction :
  // - n - x - ... - x^(n-1) + 2n x^(n)= 0
  coeffs = zeros(1,n+1);
  coeffs(1) = -n
  coeffs(2:n) = -1
  coeffs(n+1) = 2 * n
  p=poly(coeffs,"x","coeff")
  r = roots(p , "e")
  rminic = min(abs(r))
  rmaxic = max(abs(r))
  mprintf("%d & %f & %f & %f & %f \\\\\n", n, rminoc, rmaxoc, rminic, rmaxic)
endfunction

function drawfigure1 ( nbmax )
  rminoctable = zeros(1,nbmax)
  rmaxoctable = zeros(1,nbmax)
  rminictable = zeros(1,nbmax)
  rmaxictable = zeros(1,nbmax)
  for n = 1 : nbmax
    [rminoc , rmaxoc , rminic , rmaxic] = computeroots1_abstract ( n )
    rminoctable ( n ) = rminoc
    rmaxoctable ( n ) = rmaxoc
    rminictable ( n ) = rminic
    rmaxictable ( n ) = rmaxic
  end
  plot2d ( 1:nbmax , [ rminoctable' , rmaxoctable' , rminictable' , rmaxictable' ] )
  f = gcf();
  f.children.title.text = "Nelder-Mead characteristic equation roots";
  f.children.x_label.text = "Number of variables (n)";
  f.children.y_label.text = "Roots of the characteristic equation";
  captions(f.children.children.children,["R-max-IC","R-min-IC","R-max-OC","R-min-OC"]);
  f.children.children(1).legend_location="in_lower_right";
  for i = 1:4
  mypoly = f.children.children(2).children(i);
  mypoly.foreground=i;
  mypoly.line_style=i;
  end
  xs2png(0,"neldermead-roots.png");
endfunction
```

For the reflection characteristic equation, the roots all have a unity modulus. The minimum and maximum roots of the inside contraction ("ic" in the table) and outside contraction ("oc" in the table) steps are presented in table 4.13. These roots are presented graphically in figure 4.14. We see that the roots start from 0.5 when $n = 1$ and converge rapidly toward 1 when $n \to \infty$.

4.4.2 With variable parameters

In this section, we analyze the roots of the characteristic equation with *variable* inside and outside contraction coefficients.

Outside contraction

If the outside contraction step is repeatedly performed with variable $\mu_{oc} \in [0, \mu_r[$, then

$$\mathbf{v}^{(k+n)} = \overline{\mathbf{v}}^{(k)} + \mu_{oc}\left(\overline{\mathbf{v}}^{(k)} - \mathbf{v}^{(k)}\right) \tag{4.52}$$

n	$\min_{i=1,n}\mu_i^{oc}$	$\max_{i=1,n}\mu_i^{oc}$	$\min_{i=1,n}\mu_i^{ic}$	$\max_{i=1,n}\mu_i^{ic}$
1	0.500000	0.500000	0.500000	0.500000
2	0.707107	0.707107	0.593070	0.843070
3	0.776392	0.829484	0.734210	0.927534
4	0.817185	0.865296	0.802877	0.958740
5	0.844788	0.888347	0.845192	0.973459
6	0.864910	0.904300	0.872620	0.981522
7	0.880302	0.916187	0.892043	0.986406
8	0.892487	0.925383	0.906346	0.989584
9	0.902388	0.932736	0.917365	0.991766
10	0.910599	0.938752	0.926070	0.993329
11	0.917524	0.943771	0.933138	0.994485
12	0.923446	0.948022	0.938975	0.995366
13	0.917250	0.951672	0.943883	0.996051
14	0.912414	0.954840	0.948062	0.996595
15	0.912203	0.962451	0.951666	0.997034
16	0.913435	0.968356	0.954803	0.997393
17	0.915298	0.972835	0.957559	0.997691
18	0.917450	0.976361	0.959999	0.997940
19	0.919720	0.979207	0.962175	0.998151
20	0.922013	0.981547	0.964127	0.998331
21	0.924279	0.983500	0.965888	0.998487
22	0.926487	0.985150	0.967484	0.998621
23	0.928621	0.986559	0.968938	0.998738
24	0.930674	0.987773	0.970268	0.998841
25	0.932640	0.988826	0.971488	0.998932
26	0.934520	0.989747	0.972613	0.999013
27	0.936316	0.990557	0.973652	0.999085
28	0.938030	0.991274	0.974616	0.999149
29	0.939666	0.991911	0.975511	0.999207
30	0.941226	0.992480	0.976346	0.999259
31	0.942715	0.992991	0.977126	0.999306
32	0.944137	0.993451	0.977856	0.999348
33	0.945495	0.993867	0.978540	0.999387
34	0.946793	0.994244	0.979184	0.999423
35	0.948034	0.994587	0.979791	0.999455
36	0.949222	0.994900	0.980363	0.999485
37	0.950359	0.995187	0.980903	0.999513
38	0.951449	0.995450	0.981415	0.999538
39	0.952494	0.995692	0.981900	0.999561
40	0.953496	0.995915	0.982360	0.999583
45	0.957952	0.996807	0.984350	0.999671
50	0.961645	0.997435	0.985937	0.999733
55	0.964752	0.997894	0.987232	0.999779
60	0.967399	0.998240	0.988308	0.999815
65	0.969679	0.998507	0.989217	0.999842
70	0.971665	0.998718	0.989995	0.999864
75	0.973407	0.998887	0.990669	0.999881
80	0.974949	0.999024	0.991257	0.999896
85	0.976323	0.999138	0.991776	0.999908
90	0.977555	0.999233	0.992236	0.999918
95	0.978665	0.999313	0.992648	0.999926
100	0.979671	0.999381	0.993018	0.999933

Fig. 4.13 : Roots of the characteristic equations of the Nelder-Mead method with standard coefficients. (Some results are not displayed to make the table fit the page).

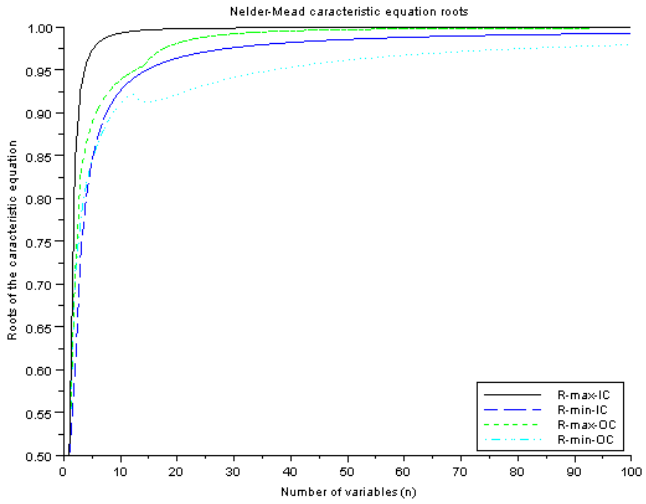

Fig. 4.14 : Modulus of the roots of the characteristic equations of the Nelder-Mead method with standard coefficients – R-max-IC is the maximum of the modulus of the root of the Inside Contraction steps

$$= (1+\mu_{oc})\overline{\mathbf{v}}^{(k)} - \mu_{oc}\mathbf{v}^{(k)} \tag{4.53}$$

By plugging the definition of the centroid into the previous equality, we find the recurrence formula

$$n\mathbf{v}^{(k+n)} - (1+\mu_{oc})\mathbf{v}^{(k+1)} - \ldots - (1+\mu_{oc})\mathbf{v}^{(k+n-1)} + n\mu_{oc}\mathbf{v}^{(k)} = 0 \tag{4.54}$$

The associated characteristic equation is

$$n\mu^n - (1+\mu_{oc})\mu^{n-1} - \ldots - (1+\mu_{oc})\mu + n\mu_{oc} = 0. \tag{4.55}$$

Inside contraction

We suppose that the inside contraction step is repeatedly performed with $-1 < \mu_{ic} < 0$. The characteristic equation is the same as 4.55, but it is here studied in the range $\mu_{ic} \in]-1, 0[$.

To study the convergence of the method, we simply have to study the roots of equation 4.55, where the range $]-1, 0[$ corresponds to the inside contraction (with $-1/2$ as the standard value) and where the range $]0, \mu_r[$ corresponds to the outside contraction (with $1/2$ as the standard value).

In the following Scilab script, we compute the minimum and maximum root of the characteristic equation, with n fixed.

```
//
// rootsvariable --
//   Compute roots of the characteristic equation
//   of Nelder-Mead with variable coefficient mu.
// Polynomial for outside/inside contraction :
// n mu - (1+mu)x - ... - (1+mu)x^(n-1) + n x^(n) = 0
//
function [rmin , rmax] = rootsvariable ( n , mu )
```

```
  coeffs = zeros(1,n+1);
  coeffs(1) = n * mu
  coeffs(2:n) = -(1+mu)
  coeffs(n+1) = n
  p=poly(coeffs,"x","coeff")
  r = roots(p , "e")
  rmin = min(abs(r))
  rmax = max(abs(r))
  mprintf("%f & %f & %f\\\\\n", mu, rmin, rmax)
endfunction

function drawfigure_variable ( n , nmumax )
  rmintable = zeros(1,nmumax)
  rmaxtable = zeros(1,nmumax)
  mutable = linspace ( -1 , 1 , nmumax )
  for index = 1 : nmumax
    mu = mutable ( index )
    [rmin , rmax ] = rootsvariable ( n , mu )
    rmintable ( index ) = rmin
    rmaxtable ( index ) = rmax
  end
  plot2d ( mutable , [ rmintable' , rmaxtable' ] )
  f = gcf();
  pause
  f.children.title.text = "Nelder-Mead characteristic equation roots";
  f.children.x_label.text = "Contraction coefficient";
  f.children.y_label.text = "Roots of the characteristic equation";
  captions(f.children.children.children,["R-max","R-min"]);
  f.children.children(1).legend_location="in_lower_right";
  for i = 1:2
  mypoly = f.children.children(2).children(i);
  mypoly.foreground=i;
  mypoly.line_style=i;
  end
  xs2png(0,"neldermead-roots-variable.png");
endfunction
```

The figure 4.15 presents the minimum and maximum modulus of the roots of the characteristic equation with $n = 10$. The result is that when μ_{oc} is close to 0, the minimum root has a modulus close to 0. The maximum root remains close to 1, whatever the value of the contraction coefficient. This result would mean that either modifying the contraction coefficient has no effect (because the maximum modulus of the roots is close to 1) or diminishing the contraction coefficient should improve the convergence speed (because the minimum modulus of the roots gets closer to 0). This is the expected result because the more the contraction coefficient is close to 0, the more the new vertex is close to 0, which is, in our particular situation, the global minimizer. No general conclusion can be drawn from this single experiment.

4.5 Numerical experiments

In this section, we present some numerical experiments with the Nelder-Mead algorithm. The two first numerical experiments involve simple quadratic functions. These experiments allows to see the difference between Spendley's et al. algorithm and the Nelder-Mead algorithm. We then present several experiments taken from the bibliography. The O'Neill experiments [31] are performed in order to check that our algorithm is a correct implementation. We then present several numerical experiments where the Nelder-Mead does not converge properly. We analyze the Mc Kinnon counter example from [21]. We show the behavior of the Nelder-Mead simplex method for a family of examples which cause the method to converge to a non stationnary point. We analyze the counter examples presented by Han in his Phd thesis [12]. In these experiments,

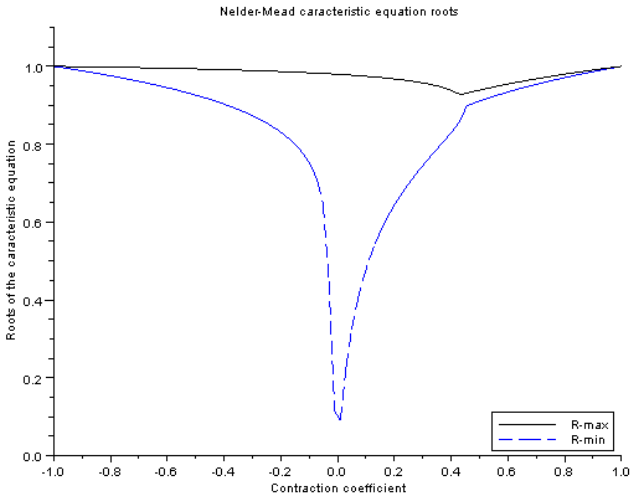

Fig. 4.15 : Modulus of the roots of the characteristic equations of the Nelder-Mead method with variable contraction coefficient and $n = 10$ – R-max is the maximum of the modulus of the root of the characteristic equation

the Nelder-Mead algorithm degenerates by applying repeatedly the inside contraction step. We also reproduce numerical experiments extracted from Torczon's Phd Thesis [46], where Virginia Torczon presents the multi-directional direct search algorithm.

4.5.1 Quadratic function

The function we try to minimize is the following quadratic in 2 dimensions

$$f(x_1, x_2) = x_1^2 + x_2^2 - x_1 x_2. \tag{4.56}$$

The stopping criteria is based on the relative size of the simplex with respect to the size of the initial simplex

$$\sigma_+(S) < tol \times \sigma_+(S_0), \tag{4.57}$$

where the tolerance is set to $tol = 10^{-8}$.

The initial simplex is a regular simplex with unit length.

The following Scilab script allows to perform the optimization.

```
function [ y , index ] = quadratic ( x , index )
  y = x(1)^2 + x(2)^2 - x(1) * x(2);
endfunction
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-function",quadratic);
nm = neldermead_configure(nm,"-x0",[2.0 2.0]');
nm = neldermead_configure(nm,"-maxiter",100);
nm = neldermead_configure(nm,"-maxfunevals",300);
nm = neldermead_configure(nm,"-tolxmethod",%f);
```

```
nm = neldermead_configure(nm,"-tolsimplexizerelative",1.e-8);
nm = neldermead_configure(nm,"-simplex0method","spendley");
nm = neldermead_configure(nm,"-method","variable");
nm = neldermead_search(nm);
neldermead_display(nm);
nm = neldermead_destroy(nm);
```

The numerical results are presented in table 4.16.

Iterations	65
Function Evaluations	130
x_0	$(2.0, 2.0)$
Relative tolerance on simplex size	10^{-8}
Exact $x^\star$	$(0., 0.)$
Computed $x^\star$	$(-2.519D-09, 7.332D-10)$
Computed $f(x^\star)$	$8.728930e-018$

Fig. 4.16 : Numerical experiment with Nelder-Mead method on the quadratic function $f(x_1, x_2) = x_1^2 + x_2^2 - x_1 x_2$

The various simplices generated during the iterations are presented in figure 4.17.

The figure 4.18 presents the history of the oriented length of the simplex. The length is updated at each iteration, which generates a continuous evolution of the length, compared to the step-by-step evolution of the simplex with the Spendley et al. algorithm.

The convergence is quite fast in this case, since less than 70 iterations allow to get a function value lower than 10^{-15}, as shown in figure 4.19.

Badly scaled quadratic function

The function we try to minimize is the following quadratic in 2 dimensions

$$f(x_1, x_2) = a x_1^2 + x_2^2, \tag{4.58}$$

where $a > 0$ is a chosen scaling parameter. The more a is large, the more difficult the problem is to solve with the simplex algorithm.

We set the maximum number of function evaluations to 400. The initial simplex is a regular simplex with unit length. The stopping criteria is based on the relative size of the simplex with respect to the size of the initial simplex

$$\sigma_+(S) < tol \times \sigma_+(S_0), \tag{4.59}$$

where the tolerance is set to $tol = 10^{-8}$.

The following Scilab script allows to perform the optimization.

```
a = 100.0;
function [ y , index ] = quadratic ( x , index )
  y = a * x(1)^2 + x(2)^2;
endfunction
```

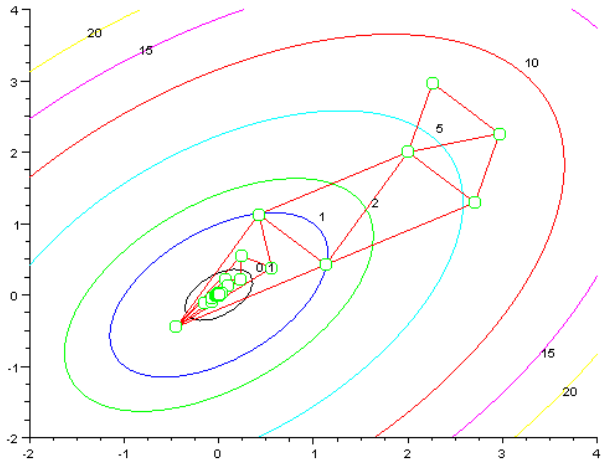

Fig. 4.17 : Nelder-Mead numerical experiment – history of simplex

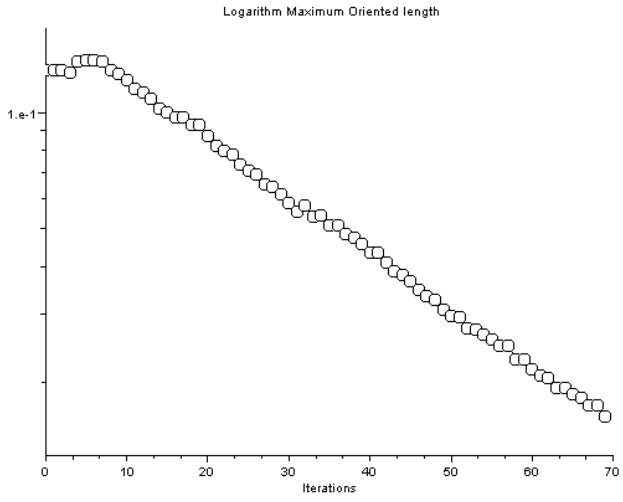

Fig. 4.18 : Nelder-Mead numerical experiment – History of logarithm of length of simplex

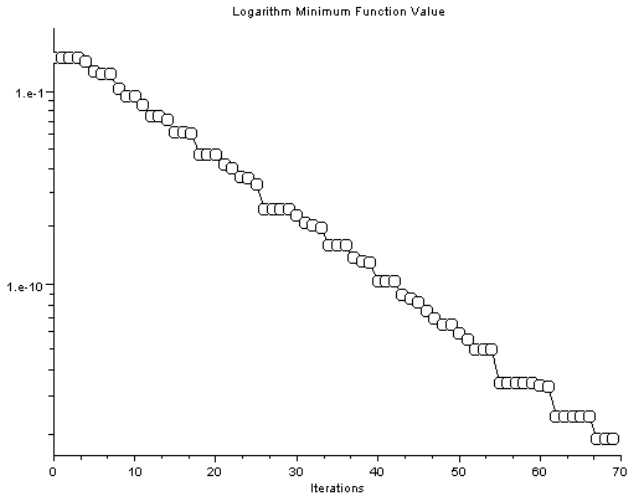

Fig. 4.19 : Nelder-Mead numerical experiment – History of logarithm of function

```
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-function",quadratic);
nm = neldermead_configure(nm,"-x0",[10.0  10.0]');
nm = neldermead_configure(nm,"-maxiter",400);
nm = neldermead_configure(nm,"-maxfunevals",400);
nm = neldermead_configure(nm,"-tolxmethod",%f);
nm = neldermead_configure(nm,"-tolsimplexizerelative",1.e-8);
nm = neldermead_configure(nm,"-simplex0method","spendley");
nm = neldermead_configure(nm,"-method","variable");
nm = neldermead_search(nm);
neldermead_display(nm);
nm = neldermead_destroy(nm);
```

The numerical results are presented in table 4.20, where the experiment is presented for $a = 100$. We can check that the number of function evaluation (161 function evaluations) is much lower than the number for the fixed shape Spendley et al. method (400 function evaluations) and that the function value at optimum is very accurate ($f(x^\star) \approx 10^{-17}$ compared to Spendley's et al. $f(x^\star) \approx 0.08$).

In figure 4.21, we analyze the behavior of the method with respect to scaling. We check that the method behaves very smoothly, with a very small number of additional function evaluations when the scaling deteriorates. This shows how much the Nelder-Mead algorithms improves over Spendley's et al. method.

4.5.2 Sensitivity to dimension

In this section, we try to reproduce the result presented by Han and Neumann [13], which shows that the convergence rate of the Nelder-Mead algorithms rapidly deteriorates when the number

	Nelder-Mead	Spendley et al.
Iterations	82	340
Function Evaluations	164	Max=400
a	100.0	100.0
x_0	$(10.0, 10.0)$	$(10.0, 10.0)$
Initial simplex	regular	regular
Initial simplex length	1.0	1.0
Relative tolerance on simplex size	10^{-8}	10^{-8}
Exact $x^\star$	$(0., 0.)$	$(0., 0.)$
Computed $x^\star$	$(-2.D-10-1.D-09)$	$(0.001, 0.2)$
Computed $f(x^\star)$	$1.D-017$	0.08

Fig. 4.20 : Numerical experiment with Nelder-Mead method on a badly scaled quadratic function. The variable shape Nelder-Mead algorithm improves the accuracy of the result compared to the fixed shaped Spendley et al. method.

a	Function Evaluations	Computed $f(x^\star)$	Computed $x^\star$
1.0	147	$1.856133e-017$	$(1.920D-09, -3.857D-09)$
10.0	156	$6.299459e-017$	$(2.482D-09, 1.188D-09)$
100.0	164	$1.140383e-017$	$(-2.859D-10, -1.797D-09)$
1000.0	173	$2.189830e-018$	$(-2.356D-12, 1.478D-09)$
10000.0	189	$1.128684e-017$	$(2.409D-11, -2.341D-09)$

Fig. 4.21 : Numerical experiment with Nelder-Mead method on a badly scaled quadratic function

of variables increases. The function we try to minimize is the following quadratic in n-dimensions

$$f(\mathbf{x}) = \sum_{i=1,n} x_i^2. \tag{4.60}$$

The initial simplex is given to the solver. The first vertex is the origin ; this vertex is never updated during the iterations. The other vertices are based on uniform random numbers in the interval $[-1, 1]$. The vertices $i = 2, n+1$ are computed from

$$\mathbf{v}_i^{(0)} = 2rand(n, 1) - 1, \tag{4.61}$$

as prescribed by [13]. In Scilab, the *rand* function returns a matrix of uniform random numbers in the interval $[0, 1)$.

The stopping criteria is based on the absolute size of the simplex, i.e. the simulation is stopped when

$$\sigma_+(S) < tol, \tag{4.62}$$

where the tolerance is set to $tol = 10^{-8}$.

We perform the experiment for $n = 1, \ldots, 19$. For each experiment, we compute the convergence rate from

$$\rho(S_0, n) = \left(\frac{\sigma(S_k)}{\sigma(S_0)} \right)^{1/k}, \tag{4.63}$$

where k is the number of iterations.

The following Scilab script allows to perform the optimization.

```
function [ f , index ] = quadracticn ( x , index )
  f = sum(x.^2);
endfunction
//
// solvepb --
//   Find the solution for the given number of dimensions
//
function [nbfevals , niter , rho] = solvepb ( n )
  rand("seed",0)
  nm = neldermead_new ();
  nm = neldermead_configure(nm,"-numberofvariables",n);
  nm = neldermead_configure(nm,"-function",quadracticn);
  nm = neldermead_configure(nm,"-x0",zeros(n,1));
  nm = neldermead_configure(nm,"-maxiter",2000);
  nm = neldermead_configure(nm,"-maxfunevals",2000);
  nm = neldermead_configure(nm,"-tolxmethod",%f);
  nm = neldermead_configure(nm,"-tolsimplexizerelative",0.0);
  nm = neldermead_configure(nm,"-tolsimplexizeabsolute",1.e-8);
  nm = neldermead_configure(nm,"-simplex0method","given");
  coords (1,1:n) = zeros(1,n);
  for i = 2:n+1
    coords (i,1:n) = 2.0 * rand(1,n) - 1.0;
  end
  nm = neldermead_configure(nm,"-coords0",coords);
  nm = neldermead_configure(nm,"-method","variable");
  nm = neldermead_search(nm);
  si0 = neldermead_get ( nm , "-simplex0" );
  sigma0 = optimsimplex_size ( si0 , "sigmaplus" );
  siopt = neldermead_get ( nm , "-simplexopt" );
  sigmaopt = optimsimplex_size ( siopt , "sigmaplus" );
  niter = neldermead_get ( nm , "-iterations" );
  rho = (sigmaopt/sigma0)^(1.0/niter);
  nbfevals = neldermead_get ( nm , "-funevals" );
  mprintf ( "%d_%d_%d_%f\n", n , nbfevals , niter , rho );
  nm = neldermead_destroy(nm);
endfunction
// Perform the 20 experiments
for n = 1:20
  [nbfevals niter rho] = solvepb ( n );
  array_rho(n) = rho;
  array_nbfevals(n) = nbfevals;
  array_niter(n) = niter;
end
```

The figure 4.22 presents the results of this experiment. The rate of convergence, as measured by $\rho(S_0, n)$ converges rapidly toward 1.

We check that the number of function evaluations increases approximately linearly with the dimension of the problem in figure 4.23. A rough rule of thumb is that, for $n = 1, 19$, the number of function evaluations is equal to $100n$.

The figure 4.24 presents the rate of convergence depending on the number of variables. The figure shows that the rate of convergence rapidly gets close to 1 when the number of variables increases. That shows that the rate of convergence is slower and slower as the number of variables increases, as explained by Han & Neumann.

n	Function evaluations	Iterations	$\rho(S_0, n)$
1	56	27	0.513002
2	113	55	0.712168
3	224	139	0.874043
4	300	187	0.904293
5	388	249	0.927305
6	484	314	0.941782
7	583	383	0.951880
8	657	430	0.956872
9	716	462	0.959721
10	853	565	0.966588
11	910	596	0.968266
12	1033	685	0.972288
13	1025	653	0.970857
14	1216	806	0.976268
15	1303	864	0.977778
16	1399	929	0.979316
17	1440	943	0.979596
18	1730	1193	0.983774
19	1695	1131	0.982881
20	1775	1185	0.983603

Fig. 4.22 : Numerical experiment with Nelder-Mead method on a generalized quadratic function

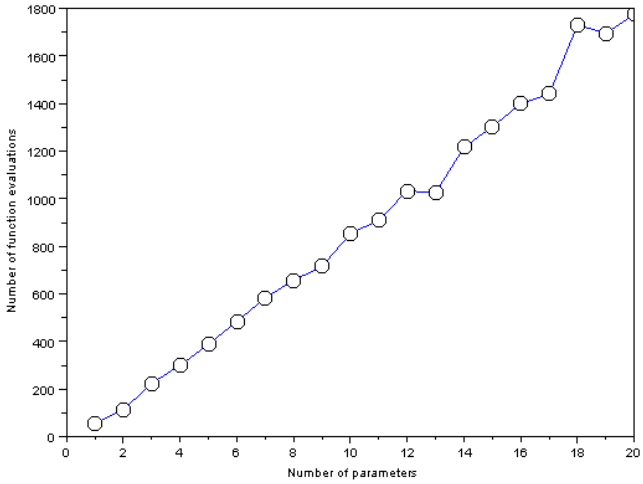

Fig. 4.23 : Nelder-Mead numerical experiment – Number of function evaluations depending on the number of variables

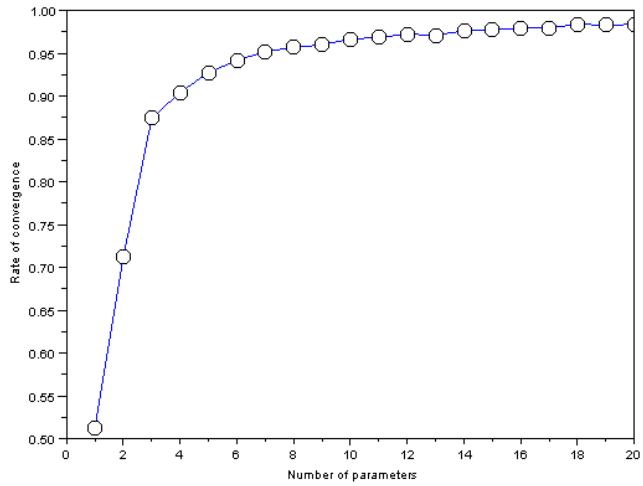

Fig. 4.24 : Nelder-Mead numerical experiment – Rate of convergence depending on the number of variables

4.5.3 O'Neill test cases

In this section, we present the results by O'Neill, who implemented a fortran 77 version of the Nelder-Mead algorithm [31].

The O'Neill implementation of the Nelder-Mead algorithm has the following particularities

- the initial simplex is computed from the axes and a (single) length,
- the stopping rule is based on variance (not standard deviation) of function value,
- the expansion is greedy, i.e. the expansion point is accepted if it is better than the lower point,
- an automatic restart is performed if a factorial test shows that the computed optimum is greater than a local point computed with a relative epsilon equal to 1.e-3 and a step equal to the length of the initial simplex.

The following tests are presented by O'Neill :

- Rosenbrock's parabolic valley [39]

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{4.64}$$

with starting point $\mathbf{x}_0 = (x_1, x_2) = (-1.2, 1)^T$. The function value at initial guess is $f(\mathbf{x}_0) = 24.2$. The solution is $\mathbf{x}^\star = (1, 1)^T$ where the function value is $f(\mathbf{x}^\star) = 0$.

- Powell's quartic function [36]

$$f(x_1, x_2, x_3, x_4) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4 \tag{4.65}$$

with starting point $\mathbf{x}_0 = (x_1, x_2, x_3, x_4) = (3, -1, 0, 1)^T$. The function value at initial guess is $f(\mathbf{x}_0) = 215.$. The solution is $\mathbf{x}^\star = (0, 0, 0, 0)^T$ where the function value is $f(\mathbf{x}^\star) = 0.$.

- Fletcher and Powell's helical valley [8]

$$f(x_1, x_2, x_3) = 100\left(x_3 + 10\theta(x_1, x_2)\right)^2 + \left(\sqrt{x_1^2 + x_2^2} - 1\right)^2 + x_3^2 \tag{4.66}$$

where

$$2\pi\theta(x_1, x_2) = \begin{cases} \arctan(x_2, x_1), & \text{if } x_1 > 0 \\ \pi + \arctan(x_2, x_1), & \text{if } x_1 < 0 \end{cases} \tag{4.67}$$

with starting point $\mathbf{x}_0 = (x_1, x_2, x_3) = (-1, 0, 0)$. The function value at initial guess is $f(\mathbf{x}_0) = 2500$. The solution is $\mathbf{x}^\star = (1, 0, 0)^T$ where the function value is $f(\mathbf{x}^\star) = 0.$. Note that since $\arctan(0/0)$ is not defined neither the function f on the line $(0, 0, x_3)$. This line is excluded by assigning a very large value to the function.

- the sum of powers

$$f(x_1, \ldots, x_{10}) = \sum_{i=1,10} x_i^4 \tag{4.68}$$

with starting point $\mathbf{x}_0 = (x_1, \ldots, x_{10}) = (1, \ldots, 1)$. The function value at initial guess is $f(\mathbf{x}_0) = 10$. The solution is $\mathbf{x}^\star = (0, \ldots, 0)^T$ where the function value is $f(\mathbf{x}^\star) = 0$..

The parameters are set to (following O'Neill's notations)

- $REQMIN = 10^{-16}$, the absolute tolerance on the variance of the function values in the simplex,
- $STEP = 1.0$, the absolute side length of the initial simplex,
- $ICOUNT = 1000$, the maximum number of function evaluations.

The following Scilab script allows to define the objective functions.

```
// Rosenbrock's "banana" function
// initialguess [-1.2 1.0]
// xoptimum [1.0 1.0}
// foptimum 0.0
function [ y , index ] = rosenbrock ( x , index )
y = 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
endfunction
// Powell's quartic valley
// initialguess [3.0 -1.0 0.0 1.0]
// xoptimum [0.0 0.0 0.0 0.0]
// foptimum 0.0
function [ f , index ] = powellquartic ( x , index )
  f = (x(1)+10.0*x(2))^2 + 5.0 * (x(3)-x(4))^2 + (x(2)-2.0*x(3))^4 + 10.0 * (x(1) - x(4))^4
endfunction
// Fletcher and Powell helical valley
// initialguess [-1.0 0.0 0.0]
// xoptimum [1.0 0.0 0.0]
// foptimum 0.0
function [ f , index ] = fletcherpowellhelical ( x , index )
  rho = sqrt(x(1) * x(1) + x(2) * x(2))
  twopi = 2 * %pi
  if ( x(1)==0.0 ) then
    f = 1.e154
  else
    if ( x(1)>0 ) then
      theta = atan(x(2)/x(1)) / twopi
    elseif ( x(1)<0 ) then
      theta = (%pi + atan(x(2)/x(1))) / twopi
    end
    f =  100.0 * (x(3)-10.0*theta)^2 + (rho - 1.0)^2 + x(3)*x(3)
  end
endfunction
// Sum of powers
// initialguess ones(10,1)
// xoptimum zeros(10,1)
// foptimum 0.0
function [ f , index ] = sumpowers ( x , index )
  f = sum(x(1:10).^4);
endfunction
```

The following Scilab function solves an optimization problem, given the number of parameters, the cost function and the initial guess.

```
//
// solvepb --
//   Find the solution for the given problem.
// Arguments
//   n : number of variables
//   cfun : cost function
```

```
//   x0 : initial guess
//
function [nbfevals , niter , nbrestart , fopt , cputime ] = solvepb ( n , cfun , x0 )
  tic();
  nm = neldermead_new ();
  nm = neldermead_configure(nm,"-numberofvariables",n);
  nm = neldermead_configure(nm,"-function",cfun);
  nm = neldermead_configure(nm,"-x0",x0);
  nm = neldermead_configure(nm,"-maxiter",1000);
  nm = neldermead_configure(nm,"-maxfunevals",1000);
  nm = neldermead_configure(nm,"-tolxmethod",%f);
  nm = neldermead_configure(nm,"-tolsimplexizemethod",%f);
  // Turn ON the tolerance on variance
  nm = neldermead_configure(nm,"-tolvarianceflag",%t);
  nm = neldermead_configure(nm,"-tolabsolutevariance",1.e-16);
  nm = neldermead_configure(nm,"-tolrelativevariance",0.0);
  // Turn ON automatic restart
  nm = neldermead_configure(nm,"-restartflag",%t);
  nm = neldermead_configure(nm,"-restarteps",1.e-3);
  nm = neldermead_configure(nm,"-restartstep",1.0);
  // Turn ON greedy expansion
  nm = neldermead_configure(nm,"-greedy",%t);
  // Set initial simplex to axis-by-axis (this is already the default anyway)
  nm = neldermead_configure(nm,"-simplex0method","axes");
  nm = neldermead_configure(nm,"-simplex0length",1.0);
  nm = neldermead_configure(nm,"-method","variable");
  //nm = neldermead_configure(nm,"-verbose",1);
  //nm = neldermead_configure(nm,"-verbosetermination",1);
  //
  // Perform optimization
  //
  nm = neldermead_search(nm);
  //neldermead_display(nm);
  niter = neldermead_get ( nm , "-iterations" );
  nbfevals = neldermead_get ( nm , "-funevals" );
  fopt = neldermead_get ( nm , "-fopt" );
  xopt = neldermead_get ( nm , "-xopt" );
  nbrestart = neldermead_get ( nm , "-restartnb" );
  status = neldermead_get ( nm , "-status" );
  nm = neldermead_destroy(nm);
  cputime = toc();
  mprintf ( "==============================\n")
  mprintf ( "status = %s\n" , status )
  mprintf ( "xopt = [%s]\n" , strcat(string(xopt)," ") )
  mprintf ( "fopt = %e\n" , fopt )
  mprintf ( "niter = %d\n" , niter )
  mprintf ( "nbfevals = %d\n" , nbfevals )
  mprintf ( "nbrestart = %d\n" , nbrestart )
  mprintf ( "cputime = %f\n" , cputime )
  //mprintf ( "%d %d %e %d %f\n", nbfevals , nbrestart , fopt , niter , cputime );
endfunction
```

The following Scilab script solves the 4 cases.

```
// Solve Rosenbrock's
x0 = [-1.2 1.0].';
[nbfevals , niter , nbrestart , fopt , cputime ] = solvepb ( 2 , rosenbrock , x0 );

// Solve Powell's quartic valley
x0 = [3.0 -1.0 0.0 1.0].';
[nbfevals , niter , nbrestart , fopt , cputime ] = solvepb ( 4 , powellquartic , x0 );

// Solve Fletcher and Powell helical valley
x0 = [-1.0 0.0 0.0].';
[nbfevals , niter , nbrestart , fopt , cputime ] = solvepb ( 3 , fletcherpowellhelical , x0 );

// Solve Sum of powers
x0 = ones(10,1);
[nbfevals , niter , nbrestart , fopt , cputime ] = solvepb ( 10 , sumpowers , x0 );
```

The table 4.25 presents the results which were computed by O'Neill compared with Scilab's. For most experiments, the results are very close in terms of number of function evaluations. The problem #4 exhibits a different behavior than the results presented by O'Neill. For Scilab, the tolerance on variance of function values is reach after 3 restarts, whereas for O'Neill, the algorithm is restarted once and gives the result with 474 function evaluations. We did not find

any explanation for this behavior. A possible cause of difference may be the floating point system which are different and may generate different simplices in the algorithms. Although the CPU times cannot be compared (the article is dated 1972 !), let's mention that the numerical experiment were performed by O'Neill on a ICL 4-50 where the two problem 1 and 2 were solved in 3.34 seconds and the problems 3 and 4 were solved in 22.25 seconds.

Author	Problem	Function Evaluations	Number Of Restarts	Function Value	Iterations	CPU Time
O'Neill	1	148	0	3.19e-9	?	?
Scilab	1	155	0	1.158612e-007	80	0.625000
O'Neill	2	209	0	7.35e-8	?	?
Scilab	2	234	0	1.072588e-008	126	0.938000
O'Neill	3	250	0	5.29e-9	?	?
Scilab	3	263	0	4.560288e-008	137	1.037000
O'Neill	4	474	1	3.80e-7	?	?
Scilab	4	616	3	3.370756e-008	402	2.949000

Fig. 4.25 : Numerical experiment with Nelder-Mead method on O'Neill test cases - O'Neill results and Scilab's results

4.5.4 Mc Kinnon: convergence to a non stationnary point

In this section, we analyze the Mc Kinnon counter example from [21]. We show the behavior of the Nelder-Mead simplex method for a family of examples which cause the method to converge to a non stationnary point.

Consider a simplex in two dimensions with vertices at 0 (i.e. the origin), $\mathbf{v}^{(n+1)}$ and $\mathbf{v}^{(n)}$. Assume that

$$f(0) < f(\mathbf{v}^{(n+1)}) < f(\mathbf{v}^{(n)}). \tag{4.69}$$

The centroid of the simplex is $\overline{\mathbf{v}} = \mathbf{v}^{(n+1)}/2$, the midpoint of the line joining the best and second vertex. The reflected point is then computed as

$$\mathbf{r}^{(n)} = \overline{\mathbf{v}} + \rho(\overline{\mathbf{v}} - \mathbf{v}^{(n)}) = \mathbf{v}^{(n+1)} - \mathbf{v}^{(n)} \tag{4.70}$$

Assume that the reflection point $\mathbf{r}^{(n)}$ is rejected, i.e. that $f(\mathbf{v}^{(n)}) < f(\mathbf{r}^{(n)})$. In this case, the inside contraction step is taken and the point $\mathbf{v}^{(n+2)}$ is computed using the reflection factor $-\gamma = -1/2$ so that

$$\mathbf{v}^{(n+2)} = \overline{\mathbf{v}} - \gamma(\overline{\mathbf{v}} - \mathbf{v}^{(n)}) = \frac{1}{4}\mathbf{v}^{(n+1)} - \frac{1}{2}\mathbf{v}^{(n)} \tag{4.71}$$

Assume then that the inside contraction point is accepted, i.e. $f(\mathbf{v}^{(n+2)}) < f(\mathbf{v}^{(n+1)})$. If this sequence of steps repeats, the simplices are subject to the following linear recurrence formula

$$4\mathbf{v}^{(n+2)} - \mathbf{v}^{(n+1)} + 2\mathbf{v}^{(n)} = 0 \tag{4.72}$$

Their general solutions are of the form

$$\mathbf{v}^{(n)} = \lambda_1^k a_1 + \lambda_2^k a_2 \tag{4.73}$$

where $\lambda_{ii=1,2}$ are the roots of the characteristic equation and $a_{ii=1,2} \in \mathbb{R}^n$. The characteristic equation is

$$4\lambda^2 - \lambda + 2\lambda = 0 \tag{4.74}$$

and has the roots

$$\lambda_1 = \frac{1+\sqrt{33}}{8} \approx 0.84307, \qquad \lambda_2 = \frac{1-\sqrt{33}}{8} \approx -0.59307 \tag{4.75}$$

After Mc Kinnon has presented the computation of the roots of the characteristic equation, he presents a special initial simplex for which the simplices degenerates because of repeated failure by inside contraction (RFIC in his article). Consider the initial simplex with vertices $\mathbf{v}^{(0)} = (1, 1)$ and $\mathbf{v}^{(1)} = (\lambda_1, \lambda_2)$ and 0. If follows that the particular solution for these initial conditions is $\mathbf{v}^{(n)} = (\lambda_1^n, \lambda_2^n)$.

Consider the function $f(x_1, x_2)$ given by

$$\begin{aligned} f(x_1, x_2) &= \theta\phi|x_1|^\tau + x_2 + x_2^2, \qquad x_1 \leq 0, & (4.76)\\ &= \theta x_1^\tau + x_2 + x_2^2, \qquad x_1 \geq 0. & (4.77) \end{aligned}$$

where θ and ϕ are positive constants. Note that $(0, -1)$ is a descent direction from the origin $(0, 0)$ and that f is stricly convex provided $\tau > 1$. f has continuous first derivatives if $\tau > 1$, continuous second derivatives if $\tau > 2$ and continuous third derivatives if $\tau > 3$.

Mc Kinnon computed the conditions on θ, ϕ and τ so that the function values are ordered as expected, i.e. so that the reflection step is rejected and the inside contraction is accepted. Examples of values which makes these equations hold are as follows : for $\tau = 1$, $\theta = 15$ and $\phi = 10$, for $\tau = 2$, $\theta = 6$ and $\phi = 60$ and for $\tau = 3$, $\theta = 6$ and $\phi = 400$.

We consider here the more regular case $\tau = 3$, $\theta = 6$ and $\phi = 400$, i.e. the function is defined by

$$f(x_1, x_2) = \begin{cases} -2400x_1^3 + x_2 + x_2^2, & \text{if } x_1 \leq 0, \\ 6x_1^3 + x_2 + x_2^2, & \text{if } x_1 \geq 0. \end{cases} \tag{4.78}$$

The solution is $\mathbf{x}^\star = (0, -0.5)^T$.

The following Scilab script solves the optimization problem. We must use the "-simplex0method" option so that a user-defined initial simplex is used. Then the "-coords0" allows to define the coordinates of the initial simplex, where each row corresponds to a vertex of the simplex

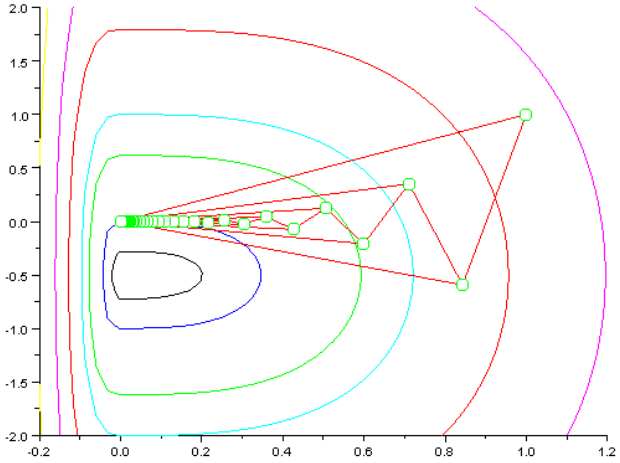

Fig. 4.26 : Nelder-Mead numerical experiment – Mc Kinnon example for convergence toward a non stationnary point

```
function [ f , index ] = mckinnon3 ( x , index )
  if ( length ( x ) ~= 2 )
    error ( 'Error: function expects a two dimensional input\n' );
  end
  tau = 3.0;
  theta = 6.0;
  phi = 400.0;
  if ( x(1) <= 0.0 )
    f = theta * phi * abs ( x(1) ).^tau + x(2) * ( 1.0 + x(2) );
  else
    f = theta       *       x(1).^tau   + x(2) * ( 1.0 + x(2) );
  end
endfunction
lambda1 = (1.0 + sqrt(33.0))/8.0;
lambda2 = (1.0 - sqrt(33.0))/8.0;
coords0 = [
1.0  1.0
0.0  0.0
lambda1 lambda2
];
x0 = [1.0  1.0]';
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-function",mckinnon3);
nm = neldermead_configure(nm,"-x0",x0);
nm = neldermead_configure(nm,"-maxiter",200);
nm = neldermead_configure(nm,"-maxfunevals",300);
nm = neldermead_configure(nm,"-tolfunrelative",10*%eps);
nm = neldermead_configure(nm,"-tolxrelative",10*%eps);
nm = neldermead_configure(nm,"-simplex0method","given");
nm = neldermead_configure(nm,"-coords0",coords0);
nm = neldermead_search(nm);
neldermead_display(nm);
nm = neldermead_destroy(nm);
```

The figure 4.26 shows the contour plot of this function and the first steps of the Nelder-Mead method. The global minimum is located at $(0, -1/2)$. Notice that the simplex degenerates to the point $(0, 0)$, which is a non stationnary point.

The figure 4.27 presents the first steps of the algorithm in this numerical experiment. Because of the particular shape of the contours of the function, the reflected point is always worse that

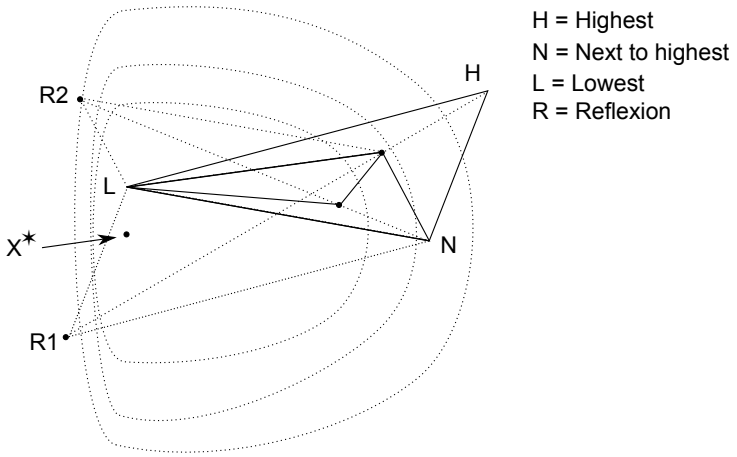

Fig. 4.27 : Nelder-Mead numerical experiment – Detail of the first steps. The simplex converges to a non stationnary point, after repeated inside contractions.

the worst vertex $\mathbf{x}_{n+1}$. This leads to the inside contraction step. The vertices constructed by Mc Kinnon are so that the situation loops without end.

4.5.5 Kelley: oriented restart

Kelley analyzed Mc Kinnon counter example in [18]. He analyzed the evolution of the simplex gradient and found that its norm begins to grow when the simplex start to degenerate. Therefore, Kelley suggest to detect the stagnation of the algorithm by using a termination criteria which is based on a sufficient decrease condition. Once that the stagnation is detected and the algorithm is stopped, restarting the algorithm with a non-degenerated simplex allows to converge toward the global minimum. Kelley advocates the use of the oriented restart, where the new simplex is so that it maximizes the chances of producing a good descent direction at the next iteration.

The following Scilab script solves the optimization problem. We must use the "-simplex0method" option so that a user-defined initial simplex is used. Then the "-coords0" allows to define the coordinates of the initial simplex, where each row corresponds to a vertex of the simplex.

We also use the "-kelleystagnationflag" option, which turns on the termination criteria associated with Kelley's stagnation detection method. Once that the algorithm is stopped, we want to automatically restart the algorithm. This is why we turn on the "-restartflag" option, which enables to perform automatically 3 restarts. After an optimization process, the automatic restart algorithm needs to know if the algorithm must restart or not. By default, the algorithm uses a factorial test, due to O'Neill. This is why we configure the "-restartdetection" to the "kelley" option, which uses Kelley's termination condition as a criteria to determine if a restart must be performed.

```
function [ f , index ] = mckinnon3 ( x , index )
```

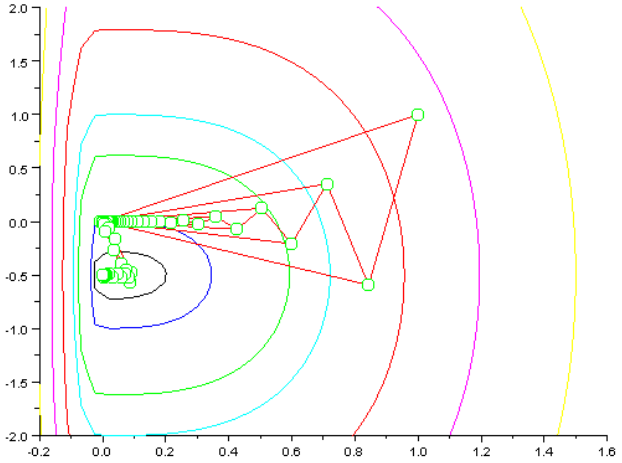

Fig. 4.28 : Nelder-Mead numerical experiment – Mc Kinnon example with Kelley's stagnation detection.

```
  if ( length ( x ) ~= 2 )
    error ( 'Error: function expects a two dimensional input\n' );
  end
  tau = 3.0;
  theta = 6.0;
  phi = 400.0;
  if ( x(1) <= 0.0 )
    f = theta * phi * abs ( x(1) ).^tau + x(2) * ( 1.0 + x(2) );
  else
    f = theta       *       x(1).^tau   + x(2) * ( 1.0 + x(2) );
  end
endfunction
lambda1 = (1.0 + sqrt(33.0))/8.0;
lambda2 = (1.0 - sqrt(33.0))/8.0;
coords0 = [
1.0 1.0
0.0 0.0
lambda1 lambda2
];
x0 = [1.0 1.0]';
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-function",mckinnon3);
nm = neldermead_configure(nm,"-x0",x0);
nm = neldermead_configure(nm,"-maxiter",200);
nm = neldermead_configure(nm,"-maxfunevals",300);
nm = neldermead_configure(nm,"-tolsimplexizerelative",1.e-6);
nm = neldermead_configure(nm,"-simplex0method","given");
nm = neldermead_configure(nm,"-coords0",coords0);
nm = neldermead_configure(nm,"-kelleystagnationflag",%t);
nm = neldermead_configure(nm,"-restartflag",%t);
nm = neldermead_configure(nm,"-restartdetection","kelley");
nm = neldermead_search(nm);
neldermead_display(nm);
nm = neldermead_destroy(nm);
```

The figure 4.28 presents the first steps of the algorithm in this numerical experiment. We see that the algorithm converges now toward the minimum $\mathbf{x}^\star = (0, -0.5)^T$.

4.5.6 Han counter examples

In his Phd thesis [12], Han presents two counter examples in which the Nelder-Mead algorithm degenerates by applying repeatedly the inside contraction step.

First counter example

The first counter example is based on the function

$$f(x_1, x_2) = x_1^2 + x_2(x_2 + 2)(x_2 - 0.5)(x_2 - 2) \tag{4.79}$$

This function is nonconvex, bounded below and has bounded level sets. The initial simplex is chosen as $S_0 = [(0., -1), (0, 1), (1, 0)]$. Han proves that the Nelder-Mead algorithm generates a sequence of simplices $S_k = [(0., -1), (0, 1), (\frac{1}{2^k}, 0)]$.

```
function [ f , index ] = han1 ( x , index )
  f = x(1)^2 + x(2) * (x(2) + 2.0) * (x(2) - 0.5) * (x(2) - 2.0);
endfunction
coords0 = [
    0.   -1.
    0.    1.
    1.    0.
]
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-function",han1);
nm = neldermead_configure(nm,"-x0",[1.0  1.0]');
nm = neldermead_configure(nm,"-maxiter",50);
nm = neldermead_configure(nm,"-maxfunevals",300);
nm = neldermead_configure(nm,"-tolfunrelative",10*%eps);
nm = neldermead_configure(nm,"-tolxrelative",10*%eps);
nm = neldermead_configure(nm,"-simplex0method","given");
nm = neldermead_configure(nm,"-coords0",coords0);
nm = neldermead_search(nm);
neldermead_display(nm);
nm = neldermead_destroy(nm);
```

The figure 4.29 presents the isovalues and the simplices during the steps of the Nelder-Mead algorithm. Note that the limit simplex contains no minimizer of the function. The failure is caused by repeated inside contractions.

Second counter example

The second counter example is based on the function

$$f(x_1, x_2) = x_1^2 + \rho(x_2) \tag{4.80}$$

where ρ is a continuous convex function with bounded level sets defined by

$$\begin{cases} \rho(x_2) = 0, & \text{if} \quad |x_2| \leq 1, \\ \rho(x_2) \geq 0, & \text{if} \quad |x_2| > 1. \end{cases} \tag{4.81}$$

The example given by Han for such a ρ function is

$$\rho(x_2) = \begin{cases} 0, & \text{if} \quad |x_2| \leq 1, \\ x_2 - 1, & \text{if} \quad x_2 > 1, \\ -x_2 - 1, & \text{if} \quad x_2 < -1. \end{cases} \tag{4.82}$$

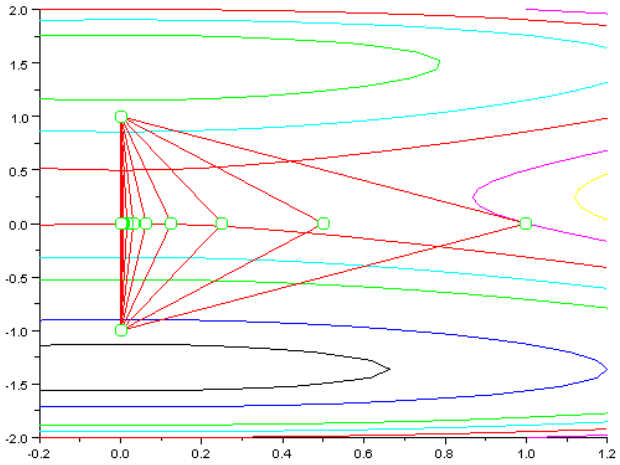

Fig. 4.29 : Nelder-Mead numerical experiment – Han example #1 for convergence toward a non stationnary point

The initial simplex is chosen as $S_0 = [(0., 1/2), (0, -1/2), (1, 0)]$. Han prooves that the Nelder-Mead algorithm generates a sequence of simplices $S_k = [(0., 1/2), (0, -1/2), (\frac{1}{2^k}, 0)]$.

```
function [ f , index ] = han2 ( x , index )
  if abs(x(2)) <= 1.0 then
    rho = 0.0
  elseif x(2) > 1.0 then
    rho = x(2) - 1
  else
    rho = -x(2) - 1
  end
  f = x(1)^2 + rho;
endfunction
coords0 = [
    0.     0.5
    0.    -0.5
    1.     0.
]
nm = neldermead_new ();
nm = neldermead_configure(nm,"-numberofvariables",2);
nm = neldermead_configure(nm,"-function",han2);
nm = neldermead_configure(nm,"-x0",[1.0  1.0]');
nm = neldermead_configure(nm,"-maxiter",50);
nm = neldermead_configure(nm,"-maxfunevals",300);
nm = neldermead_configure(nm,"-tolfunrelative",10*%eps);
nm = neldermead_configure(nm,"-tolxrelative",10*%eps);
nm = neldermead_configure(nm,"-simplex0method","given");
nm = neldermead_configure(nm,"-coords0",coords0);
nm = neldermead_search(nm);
neldermead_display(nm);
nm = neldermead_destroy(nm);
```

The figure 4.30 presents the isovalues and the simplices during the steps of the Nelder-Mead algorithm. The failure is caused by repeated inside contractions.

These two examples of non convergence show that the Nelder-Mead method may unreliable. They also reveal that the Nelder-Mead method can generate simplices which collapse into a degenerate simplex, by applying repeated inside contractions.

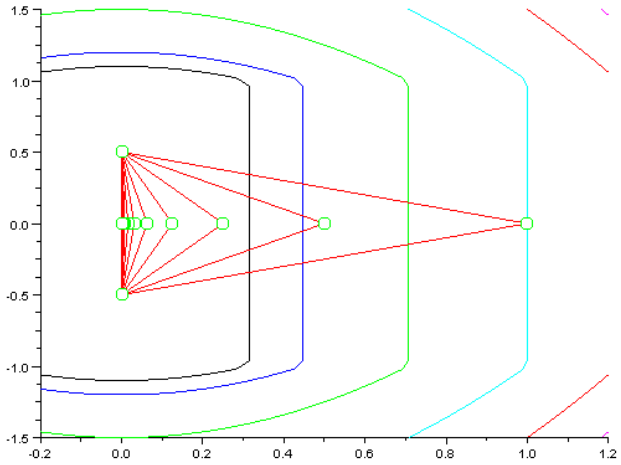

Fig. 4.30 : Nelder-Mead numerical experiment – Han example #2 for convergence toward a non stationnary point

4.5.7 Torczon's numerical experiments

In her Phd Thesis [46], Virginia Torczon presents the multi-directional direct search algorithm. In order to analyze the performances of her new algorithm, she presents some interesting numerical experiments with the Nelder-Mead algorithm. These numerical experiments are based on the collection of test problems [22], published in the ACM by Moré, Garbow and Hillstrom in 1981. These test problems are associated with varying number of variables. In her Phd, Torczon presents numerical experiments with n from 8 to 40. The stopping rule is based on the relative size of the simplex. The angle between the descent direction (given by the worst point and the centroid), and the gradient of the function is computed when the algorithm is stopped. Torczon shows that, when the tolerance on the relative simplex size is decreased, the angle converges toward 90°. This fact is observed even for moderate number of dimensions.

In this section, we try to reproduce Torczon numerical experiments.

All experiments are associated with the following sum of squares cost function

$$f(\mathbf{x}) = \sum_{i=1,m} f_i(\mathbf{x})^2, \tag{4.83}$$

where $m \geq 1$ is the number of functions f_i in the problem.

The stopping criteria is based on the relative size of the simplex and is the following

$$\frac{1}{\Delta} \max_{i=2,n+1} \|\mathbf{v}_i - \mathbf{v}_1\| \leq \epsilon, \tag{4.84}$$

where $\Delta = \max(1, \|\mathbf{v}_1\|)$. Decreasing the value of ϵ allows to get smaller simplex sizes.

The initial simplex is not specified by Virginia Torczon. In our numerical experiments, we choose an axis-by-axis simplex, with an initial length equal to 1.

Penalty #1

The first test function is the *Penalty #1* function :

$$f_i(\mathbf{x}) = 10^{-5/2}(x_i - 1), \qquad i = 1, n \tag{4.85}$$

$$f_{n+1} = -\frac{1}{4} + \sum_{j=1,n} x_j^2. \tag{4.86}$$

The initial guess is given by $\mathbf{x}_0 = ((\mathbf{x}_0)_1, (\mathbf{x}_0)_2, \ldots, (\mathbf{x}_0)_n)^T$ and $(\mathbf{x}_0)_j = j$ for $j = 1, n$.

The problem given by Moré, Garbow and Hillstrom in [22, 23] is associated with the size $n = 4$. The value of the cost function at the initial guess $\mathbf{x}_0 = (1, 2, 3, 4)^T$ is $f(\mathbf{x}_0) = 885.063$. The value of the function at the optimum is given in [22, 23] as $f(\mathbf{x}^\star) = 2.24997d - 5$.

Virginia Torzcon present the results of this numerical experiment with the Penalty #1 test case and $n = 8$. For this particular case, the initial function value is $f(\mathbf{x}_0) = 4.151406.10^4$.

In the following Scilab script, we define the *penalty1* function. We define the function *penalty1_der* which allows to compute the numerical derivative. The use of a global variable is not

```
function [ y , index , n ] = penalty1 ( x , index , n )
  y = 0.0
  for i = 1 : n
    fi = (x(i) - 1) * sqrt(1.e-5)
    y = y + fi^2
  end
  fi = -1/4 + norm(x)^2
  y = y + fi^2
endfunction

function y = penalty1_der ( x , n )
  [ y , index ] = penalty1 ( x , 1 , n )
endfunction
```

The following Scilab function defines the termination criteria, as defined in 4.84.

```
function [ this , terminate , status ] = mystoppingrule ( this , simplex )
  global _DATA_
  v1 = optimsimplex_getx ( simplex , 1 )
  delta = max ( 1.0 , norm(v1) )
  maxnorms = 0.0
  n = neldermead_cget ( this , "-numberofvariables" )
  for i = 2 : n
    vi = optimsimplex_getx ( simplex , i )
    ni = norm ( vi - v1 )
    maxnorms = max ( maxnorms , ni )
  end
  epsilon = _DATA_.epsilon
  if ( maxnorms / delta < epsilon ) then
    terminate = %t
    status = "torczon"
  else
    terminate = %f
  end
endfunction
```

The following *solvepb* function takes as input the dimension of the problem n, the cost function, the initial guess and the tolerance. It uses the *neldermead* component and configures it so that the algorithm uses the specific termination function defined previously.

```
function [nbfevals , niter , fopt , cputime ] = solvepb ( n , cfun , x0 , tolerance )
  tic();
  global _DATA_;
  _DATA_ = tlist ( [
    "T_TORCZON"
    "epsilon"
  ]);
  _DATA_.epsilon = tolerance;

  nm = neldermead_new ();
  nm = neldermead_configure(nm,"-numberofvariables",n);
  nm = neldermead_configure(nm,"-function",cfun);
  nm = neldermead_configure(nm,"-costfargument",n);
  nm = neldermead_configure(nm,"-x0",x0);
  nm = neldermead_configure(nm,"-maxiter",10000);
  nm = neldermead_configure(nm,"-maxfunevals",10000);
  nm = neldermead_configure(nm,"-tolxmethod",%f);
  nm = neldermead_configure(nm,"-tolsimplexizemethod",%f);
  // Turn ON my own termination criteria
  nm = neldermead_configure(nm,"-myterminate",mystoppingrule);
  nm = neldermead_configure(nm,"-myterminateflag",%t);
  //
  // Perform optimization
  //
  nm = neldermead_search(nm);
  niter = neldermead_get ( nm , "-iterations" );
  nbfevals = neldermead_get ( nm , "-funevals" );
  fopt = neldermead_get ( nm , "-fopt" );
  xopt = neldermead_get ( nm , "-xopt" );
  status = neldermead_get ( nm , "-status" );
  nm = neldermead_destroy(nm);
  cputime = toc();
  // Compute angle between gradient and simplex direction
  sopt = neldermead_get ( nm , "-simplexopt" )
  xhigh = optimsimplex_getx ( sopt , n + 1 )
  xbar = optimsimplex_xbar ( sopt )
  d = xbar - xhigh;
  g = derivative ( list ( penalty1_der , n ) , xopt , order=4 );
  cost = -g*d.' / norm(g) / norm(d)
  theta =acosd(cost)
  // Compute condition of matrix of directions
  D = optimsimplex_dirmat ( sopt )
  k = cond ( D )
  // Display result
  mprintf ( "==============================\n")
  mprintf ( "status = %s\n" , status )
  mprintf ( "Tolerance=%e\n" , tolerance )
  mprintf ( "xopt = [%s]\n" , strcat(string(xopt)," ") )
  mprintf ( "fopt = %e\n" , fopt )
  mprintf ( "niter = %d\n" , niter )
  mprintf ( "nbfevals = %d\n" , nbfevals )
  mprintf ( "theta = %25.15f (deg)\n" , theta )
  mprintf ( "cputime = %f (s)\n" , cputime )
  mprintf ( "cond(D) = %e (s)\n" , k )
endfunction
```

We are now able to make a loop, and get the optimum function value for various values of the tolerance use in the termination criteria.

```
x0 = [1 2 3 4 5 6 7 8].';
for tol = [1.e-1 1.e-2 1.e-3 1.e-4 1.e-5 1.e-6 1.e-7]
  [nbfevals , niter , fopt , cputime ] = solvepb ( 8 , penalty1 , x0 , tol );
end
```

The figure 4.31 presents the results of these experiments. As Virginia Torczon, we get an increasing number of function evaluations, with very little progress with respect to the function value. We also get a search direction which becomes increasingly perpendicular to the gradient.

The number of function evaluations is not the same between Torczon's and Scilab so that we can conclude that the algorithm may be different variants of the Nelder-Mead algorithm or uses a different initial simplex. We were not able to explain why the number of function evaluations is so different.

Author	Step Tolerance	$f(\mathbf{v}_1^\star)$	Function Evaluations	Angle (°)
Torzcon	1.e-1	7.0355e-5	1605	89.396677792198
Scilab	1.e-1	9.567114e-5	314	101.297069897149110
Torzcon	1.e-2	6.2912e-5	3360	89.935373548613
Scilab	1.e-2	8.247686e-5	501	88.936037514983468
Torzcon	1.e-3	6.2912e-5	3600	89.994626919197
Scilab	1.e-3	7.485404e-5	1874	90.134605846897529
Torzcon	1.e-4	6.2912e-5	3670	89.999288284747
Scilab	1.e-4	7.481546e-5	2137	90.000107262503008
Torzcon	1.e-5	6.2912e-5	3750	89.999931862232
Scilab	1.e-5	7.481546e-5	2193	90.000366248870506
Torzcon	1.e-6	6.2912e-5	3872	89.999995767877
Scilab	1.e-6	7.427204e-5	4792	90.000006745652769
Torzcon	1.e-7	6.2912e-5	3919	89.999999335010
Scilab	1.e-7	7.427204e-5	4851	89.999996903432063

Fig. 4.31 : Numerical experiment with Nelder-Mead method on penalty #1 test case - Torczon results and Scilab's results

The figure 4.32 presents the condition number of the matrix of simplex direction. When this condition number is high, the simplex is distorted. The numerical experiment shows that the condition number is fastly increasing. This corresponds to the fact that the simplex is increasingly distorted and might explains why the algorithm fails to make any progress.

4.6 Conclusion

The main advantage of the Nelder-Mead algorithm over Spendley et al. algorithm is that the shape of the simplex is dynamically updated. That allows to get a reasonably fast convergence rate on badly scaled quadratics, or more generally when the cost function is made of a sharp valley. Still, the behavior of the algorithm when the dimension of the problem increases is disappointing: the more there are variables, the more the algorithm is slow. In general, it is expected that the number of function evaluations is roughly equal to $100n$, where n is the number of parameters. When the algorithm comes close to the optimum, the simplex becomes more and more distorted, so that less and less progress is made with respect to the value of the cost function. This can measured by the fact that the direction of search becomes more and more perpendicular to the gradient of the cost function. It can also be measure by an increasing value of the condition number of the matrix of simplex directions. Therefore, the user should not require a high accuracy from the algorithm. Nevertheless, in most cases, the Nelder-Mead algorithms provides a good *improvement*

Tolerance	$cond(D)$
1.e-1	1.573141e+001
1.e-2	4.243385e+002
1.e-3	7.375247e+008
1.e-4	1.456121e+009
1.e-5	2.128402e+009
1.e-6	2.323514e+011
1.e-7	3.193495e+010

Fig. 4.32 : Numerical experiment with Nelder-Mead method on penalty #1 test case - Condition number of the matrix of simplex directions

of the solution. In some situations, the simplex can become so distorted that it converges toward a non-stationnary point. In this case, restarting the algorithm with a new nondegenerate simplex allows to converge toward the optimum.

Chapter 5

The *fminsearch* function

In this chapter, we analyze the implementation of the *fminsearch* which is provided in Scilab. In the first part, we describe the specific choices of this implementation with respect to the Nelder-Mead algorithm. In the second part, we present some numerical experiments which allows to check that the feature is behaving as expected, by comparison to Matlab's *fminsearch*.

5.1 *fminsearch*'s algorithm

In this section, we analyse the specific choices used in *fminsearch*'s algorithm. We detail what specific variant of the Nelder-Mead algorithm is performed, what initial simplex is used, the default number of iterations and the termination criteria.

5.1.1 The algorithm

The algorithm used is the Nelder-Mead algorithm. This corresponds to the "variable" value of the "-method" option of the *neldermead*. The "non greedy" version is used, that is, the expansion point is accepted only if it improves over the reflection point.

5.1.2 The initial simplex

The fminsearch algorithm uses a special initial simplex, which is an heuristic depending on the initial guess. The strategy chosen by fminsearch corresponds to the -simplex0method flag of the neldermead component, with the "pfeffer" method. It is associated with the - simplex0deltausual = 0.05 and -simplex0deltazero = 0.0075 parameters. Pfeffer's method is an heuristic which is presented in "Global Optimization Of Lennard-Jones Atomic Clusters" by Ellen Fan [7]. It is due to L. Pfeffer at Stanford. See in the help of optimsimplex for more details.

5.1.3 The number of iterations

In this section, we present the default values for the number of iterations in fminsearch.

The options input argument is an optionnal data structure which can contain the options.MaxIter field. It stores the maximum number of iterations. The default value is 200n, where n is the number of variables. The factor 200 has not been chosen by chance, but is the result of experiments performed against quadratic functions with increasing space dimension.

This result is presented in "Effect of dimensionality on the nelder-mead simplex method" by Lixing Han and Michael Neumann [13]. This paper is based on Lixing Han's PhD, "Algorithms in Unconstrained Optimization" [12]. The study is based on numerical experiment with a quadratic function where the number of terms depends on the dimension of the space (i.e. the number of variables). Their study shows that the number of iterations required to reach the tolerance criteria is roughly 100n. Most iterations are based on inside contractions. Since each step of the Nelder-Mead algorithm only require one or two function evaluations, the number of required function evaluations in this experiment is also roughly 100n.

5.1.4 The termination criteria

The algorithm used by *fminsearch* uses a particular termination criteria, based both on the absolute size of the simplex and the difference of the function values in the simplex. This termination criteria corresponds to the "-tolssizedeltafvmethod" termination criteria of the *neldermead* component.

The size of the simplex is computed with the $\sigma - +$ method, which corresponds to the "sigmaplus" method of the *optimsimplex* component. The tolerance associated with this criteria is given by the "TolX" parameter of the *options* data structure. Its default value is 1.e-4.

The function value difference is the difference between the highest and the lowest function value in the simplex. The tolerance associated with this criteria is given by the "TolFun" parameter of the *options* data structure. Its default value is 1.e-4.

5.2 Numerical experiments

In this section, we analyse the behaviour of Scilab's *fminsearch* function, by comparison of Matlab's *fminsearch*. We especially analyse the results of the optimization, so that we can check that the algorithm is indeed behaving the same way, even if the implementation is completely different.

We consider the unconstrained optimization problem [39]

$$\min f(\mathbf{x}) \tag{5.1}$$

where $\mathbf{x} \in \mathbb{R}^2$ and the objective function f is defined by

$$f(\mathbf{x}) = 100 * (x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{5.2}$$

The initial guess is

$$\mathbf{x}^0 = (-1.2, 1.)^T, \tag{5.3}$$

where the function value is

$$f(\mathbf{x}^0) = 24.2. \tag{5.4}$$

The global solution of this problem is

$$\mathbf{x}^\star = (1, 1.)^T \tag{5.5}$$

where the function value is

$$f(\mathbf{x}^\star) = 0. \tag{5.6}$$

5.2.1 Algorithm and numerical precision

In this section, we are concerned by the comparison of the behavior of the two algorithms. We are going to check that the algorithms produces the same intermediate and final results. We also analyze the numerical precision of the results, by detailing the number of significant digits.

To make a more living presentation of this topic, we will include small scripts which allow to produce the output that we are going to analyze. Because of the similarity of the languages, in order to avoid confusion, we will specify, for each script, the language we use by a small comment. Scripts and outputs written in Matlab's language will begin with

```
% Matlab
% ...
```

while script written in Scilab's language will begin with

```
// Scilab
// ...
```

The following Matlab script allows to see the behaviour of Matlab's *fminsearch* function on Rosenbrock's test case.

```
% Matlab
format long
banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
[x,fval,exitflag,output] = fminsearch(banana,[-1.2, 1])
output.message
```

When this script is launched in Matlab, the following output is produced.

```
>> % Matlab
>> format long
>> banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
>> [x,fval] = fminsearch(banana,[-1.2, 1])
>> [x,fval,exitflag,output] = fminsearch(banana,[-1.2, 1])
x =
   1.000022021783570   1.000042219751772
fval =
     8.177661197416674e-10
exitflag =
     1
output =
    iterations: 85
     funcCount: 159
     algorithm: 'Nelder-Mead simplex direct search'
       message: [1x194 char]
>> output.message
ans =
Optimization terminated:
the current x satisfies the termination criteria using
OPTIONS.TolX of 1.000000e-04
and F(X) satisfies the convergence criteria using
OPTIONS.TolFun of 1.000000e-04
```

The following Scilab script allows to solve the problem with Scilab's *fminsearch*.

```
// Scilab
format(25)
function y = banana (x)
  y = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
endfunction
[x , fval , exitflag , output] = fminsearch ( banana , [-1.2 1] )
output.message
```

The output associated with this Scilab script is the following.

```
-->// Scilab
-->format(25)
-->function y = banana (x)
-->  y = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
-->endfunction
-->[x , fval , exitflag , output] = fminsearch ( banana , [-1.2 1] )
 output  =
   algorithm: "Nelder-Mead simplex direct search"
   funcCount: 159
   iterations: 85
   message: [3x1 string]
 exitflag  =
    1.
 fval  =
    0.00000000008177661099387
 x  =
    1.000022021783556702 7009    1.000042219751771 0998227
-->output.message
 ans  =

!Optimization terminated:                                                    !
!                                                                            !
!the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004  !
!                                                                            !
!and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-004     !
```

Because the two softwares do not use the same formatting rules to produce their outputs, we must perform additionnal checking in order to check our results.

The following Scilab script displays the results with 16 significant digits.

```
// Scilab
// Print the result with 15 significant digits
mprintf ( "%.15e" , fval );
mprintf ( "%.15e %.15e" , x(1) , x(2) );
```

The previous script produces the following output.

```
-->// Scilab
-->mprintf ( "%.15e" , fval );
8.177661099387146e-010
-->mprintf ( "%.15e %.15e" , x(1) , x(2) );
1.000022021783557e+000 1.000042219751771e+000
```

These results are reproduced verbatim in the table 5.1.

We must compute the common number of significant digits in order to check the consistency of the results. The following Scilab script computes the relative error between Scilab and Matlab results.

```
// Scilab
// Compare the result
xmb = [1.000022021783570   1.000042219751772 ];
err = norm(x - xmb) / norm(xmb);
mprintf ( "Relative Error on x : %e\n", err );
fmb = 8.177661197416674e-10;
err = abs(fval - fmb) / abs(fmb);
mprintf ( "Relative Error on f : %e\n", err );
```

The previous script produces the following output.

```
// Scilab
Relative Error on x : 9.441163e-015
Relative Error on f : 1.198748e-008
```

Matlab Iterations Scilab Iterations	85 85	
Matlab Function Evaluations Scilab Function Evaluations	159 159	
Matlab $\mathbf{x}^\star$ Scilab $\mathbf{x}^\star$	1.000022021783570 1.000022021783557e+000	1.000042219751772 1.000042219751771e+000
Matlab $f(\mathbf{x}^\star)$ Scilab $f(\mathbf{x}^\star)$	8.177661197416674e-10 8.177661099387146e-010	

Fig. 5.1 : Numerical experiment with Rosenbrock's function – Comparison of results produced by Matlab and Scilab.

We must take into account for the floating point implementations of both Matlab and Scilab. In both these numerical softwares, double precision floating point numbers are used, i.e. the relative precision is both these softwares is $\epsilon \approx 10^{-16}$. That implies that there are approximately 16 significant digits. Therefore, the relative error on x, which is equivalent to 15 significant digits, is acceptable.

Therefore, the result is as close as possible to the result produced by Matlab. More specifically :

- the optimum x is the same up to 15 significant digits,
- the function value at optimum is the same up to 8 significant digits,
- the number of iterations is the same,
- the number of function evaluations is the same,
- the exit flag is the same,
- the content of the output is the same (but the string is not display the same way).

The output of the two functions is the same. We must now check that the algorithms performs the same way, that is, produces the same intermediate steps.

The following Matlab script allows to get deeper information by printing a message at each iteration with the "Display" option.

```
% Matlab
opt = optimset('Display','iter');
[x,fval,exitflag,output] = fminsearch(banana,[-1.2, 1] , opt );
```

The previous script produces the following output.

```
% Matlab
 Iteration   Func-count     min f(x)         Procedure
     0           1            24.2
     1           3            20.05         initial simplex
     2           5          5.1618          expand
     3           7          4.4978          reflect
```

4	9	4.4978	contract outside
5	11	4.38136	contract inside
6	13	4.24527	contract inside
7	15	4.21762	reflect
8	17	4.21129	contract inside
9	19	4.13556	expand
10	21	4.13556	contract inside
11	23	4.01273	expand
12	25	3.93738	expand
13	27	3.60261	expand
14	28	3.60261	reflect
15	30	3.46622	reflect
16	32	3.21605	expand
17	34	3.16491	reflect
18	36	2.70687	expand
19	37	2.70687	reflect
20	39	2.00218	expand
21	41	2.00218	contract inside
22	43	2.00218	contract inside
23	45	1.81543	expand
24	47	1.73481	contract outside
25	49	1.31697	expand
26	50	1.31697	reflect
27	51	1.31697	reflect
28	53	1.1595	reflect
29	55	1.07674	contract inside
30	57	0.883492	reflect
31	59	0.883492	contract inside
32	61	0.669165	expand
33	63	0.669165	contract inside
34	64	0.669165	reflect
35	66	0.536729	reflect
36	68	0.536729	contract inside
37	70	0.423294	expand
38	72	0.423294	contract outside
39	74	0.398527	reflect
40	76	0.31447	expand
41	77	0.31447	reflect
42	79	0.190317	expand
43	81	0.190317	contract inside
44	82	0.190317	reflect
45	84	0.13696	reflect
46	86	0.13696	contract outside
47	88	0.113128	contract outside
48	90	0.11053	contract inside
49	92	0.10234	reflect
50	94	0.101184	contract inside
51	96	0.0794969	expand
52	97	0.0794969	reflect
53	98	0.0794969	reflect
54	100	0.0569294	expand
55	102	0.0569294	contract inside
56	104	0.0344855	expand
57	106	0.0179534	expand
58	108	0.0169469	contract outside
59	110	0.00401463	reflect
60	112	0.00401463	contract inside
61	113	0.00401463	reflect
62	115	0.000369954	reflect
63	117	0.000369954	contract inside
64	118	0.000369954	reflect
65	120	0.000369954	contract inside
66	122	5.90111e−005	contract outside
67	124	3.36682e−005	contract inside
68	126	3.36682e−005	contract outside
69	128	1.89159e−005	contract outside
70	130	8.46083e−006	contract inside
71	132	2.88255e−006	contract inside
72	133	2.88255e−006	reflect
73	135	7.48997e−007	contract inside
74	137	7.48997e−007	contract inside
75	139	6.20365e−007	contract inside
76	141	2.16919e−007	contract outside
77	143	1.00244e−007	contract inside
78	145	5.23487e−008	contract inside
79	147	5.03503e−008	contract inside
80	149	2.0043e−008	contract inside
81	151	1.12293e−009	contract inside
82	153	1.12293e−009	contract outside
83	155	1.12293e−009	contract inside

```
    84          157      1.10755e-009          contract outside
    85          159      8.17766e-010          contract inside

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-004
```

The following Scilab script set the "Display" option to "iter" and run the *fminsearch* function.

```
// Scilab
opt = optimset ( "Display" , "iter" );
[x , fval , exitflag , output] = fminsearch ( banana , [-1.2 1] , opt );
```

```
// Scilab
 Iteration   Func-count     min f(x)          Procedure
     0           3            24.2
     1           3           20.05          initial simplex
     2           5         5.161796          expand
     3           7         4.497796          reflect
     4           9         4.497796          contract outside
     5          11        4.3813601          contract inside
     6          13        4.2452728          contract inside
     7          15        4.2176247          reflect
     8          17        4.2112906          contract inside
     9          19        4.1355598          expand
    10          21        4.1355598          contract inside
    11          23        4.0127268          expand
    12          25        3.9373812          expand
    13          27         3.602606          expand
    14          28         3.602606          reflect
    15          30        3.4662211          reflect
    16          32        3.2160547          expand
    17          34        3.1649126          reflect
    18          36        2.7068692          expand
    19          37        2.7068692          reflect
    20          39        2.0021824          expand
    21          41        2.0021824          contract inside
    22          43        2.0021824          contract inside
    23          45        1.8154337          expand
    24          47        1.7348144          contract outside
    25          49        1.3169723          expand
    26          50        1.3169723          reflect
    27          51        1.3169723          reflect
    28          53        1.1595038          reflect
    29          55        1.0767387          contract inside
    30          57        0.8834921          reflect
    31          59        0.8834921          contract inside
    32          61        0.6691654          expand
    33          63        0.6691654          contract inside
    34          64        0.6691654          reflect
    35          66        0.5367289          reflect
    36          68        0.5367289          contract inside
    37          70        0.4232940          expand
    38          72        0.4232940          contract outside
    39          74        0.3985272          reflect
    40          76        0.3144704          expand
    41          77        0.3144704          reflect
    42          79        0.1903167          expand
    43          81        0.1903167          contract inside
    44          82        0.1903167          reflect
    45          84        0.1369602          reflect
    46          86        0.1369602          contract outside
    47          88        0.1131281          contract outside
    48          90        0.1105304          contract inside
    49          92        0.1023402          reflect
    50          94        0.1011837          contract inside
    51          96        0.0794969          expand
    52          97        0.0794969          reflect
    53          98        0.0794969          reflect
    54         100        0.0569294          expand
    55         102        0.0569294          contract inside
    56         104        0.0344855          expand
    57         106        0.0179534          expand
    58         108        0.0169469          contract outside
    59         110        0.0040146          reflect
    60         112        0.0040146          contract inside
    61         113        0.0040146          reflect
    62         115        0.0003700          reflect
```

```
    63          117         0.0003700         contract inside
    64          118         0.0003700         reflect
    65          120         0.0003700         contract inside
    66          122         0.0000590         contract outside
    67          124         0.0000337         contract inside
    68          126         0.0000337         contract outside
    69          128         0.0000189         contract outside
    70          130         0.0000085         contract inside
    71          132         0.0000029         contract inside
    72          133         0.0000029         reflect
    73          135         0.0000007         contract inside
    74          137         0.0000007         contract inside
    75          139         0.0000006         contract inside
    76          141         0.0000002         contract outside
    77          143         0.0000001         contract inside
    78          145         5.235D-08         contract inside
    79          147         5.035D-08         contract inside
    80          149         2.004D-08         contract inside
    81          151         1.123D-09         contract inside
    82          153         1.123D-09         contract outside
    83          155         1.123D-09         contract inside
    84          157         1.108D-09         contract outside
    85          159         8.178D-10         contract inside

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-004
```

We check that the two softwares produces indeed the same intermediate results in terms of iteration, function evaluations, function values and type of steps. The only difference is the iteration #0, which is associated with function evaluation #1 in Matlab and with function evaluation #3 in Scilab. This is because Scilab calls back the output function once the initial simplex is computed, which requires 3 function evaluations.

5.2.2 Output and plot functions

In this section, we check that the output and plot features of the *fminsearch* function are the same. We also check that the fields and the content of the *optimValues* data structure and the *state* variable are the same in both languages.

The following output function plots in the current graphic window the value of the current parameter **x**. It also unloads the content of the *optimValues* data structure and prints a message in the console. To let Matlab load that script, save the content in a .m file, in a directory known by Matlab.

```
% Matlab
function stop = outfun(x, optimValues, state)
stop = false;
hold on;
plot(x(1),x(2),'.');
fc = optimValues.funccount;
fv = optimValues.fval;
it = optimValues.iteration;
pr = optimValues.procedure;
disp(sprintf ( '%d %e %d %s - %s\n' , fc , fv , it , pr , state ))
drawnow
```

The following Matlab script allows to perform the optimization so that the output function is called back at each iteration.

```
% Matlab
options = optimset('OutputFcn', @outfun);
[x fval] = fminsearch(banana, [-1.2, 1], options)
```

This produces the plot which is presented in figure 5.2.

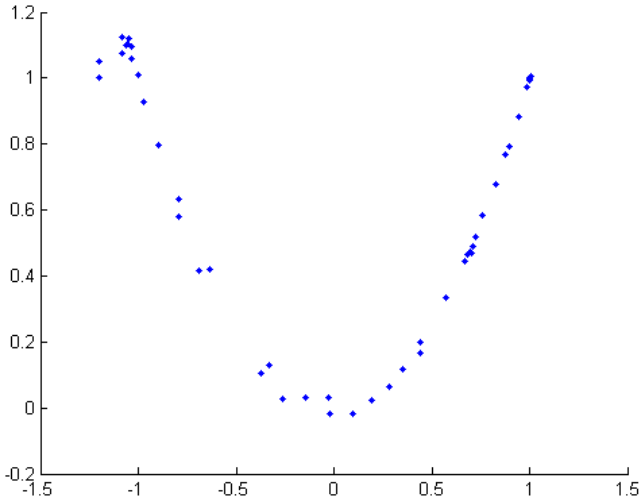

Fig. 5.2 : Plot produced by Matlab's fminsearch, with customized output function.

Matlab also prints the following messages in the console.

```
% Matlab
1 2.420000e+001 0 -- init
1 2.420000e+001 0 -- iter
3 2.005000e+001 1 -initial simplex- iter
5 5.161796e+000 2 -expand- iter
7 4.497796e+000 3 -reflect- iter
9 4.497796e+000 4 -contract outside- iter
11 4.381360e+000 5 -contract inside- iter
13 4.245273e+000 6 -contract inside- iter
[...]
149 2.004302e-008 80 -contract inside- iter
151 1.122930e-009 81 -contract inside- iter
153 1.122930e-009 82 -contract outside- iter
155 1.122930e-009 83 -contract inside- iter
157 1.107549e-009 84 -contract outside- iter
159 8.177661e-010 85 -contract inside- iter
159 8.177661e-010 85 -contract inside- done
```

The following Scilab script sets the "OutputFcn" option and then calls the *fminsearch* in order to perform the optimization.

```
// Scilab
function outfun ( x , optimValues , state )
  plot( x(1),x(2),'.');
  fc = optimValues.funccount;
  fv = optimValues.fval;
  it = optimValues.iteration;
  pr = optimValues.procedure;
  mprintf ( "%d %e %d -%s- %s\n" , fc , fv , it , pr , state )
endfunction
opt = optimset ( "OutputFcn" , outfun);
[x fval] = fminsearch ( banana , [-1.2 1] , opt );
```

The previous script produces the plot which is presented in figure 5.3.

Except for the size of the dots (which can be configured in both softwares), the graphics are exactly the same.

Scilab also prints the following messages in the console.

```
// Scilab
3 2.420000e+001 0 -- init
3 2.005000e+001 1 -initial simplex- iter
```

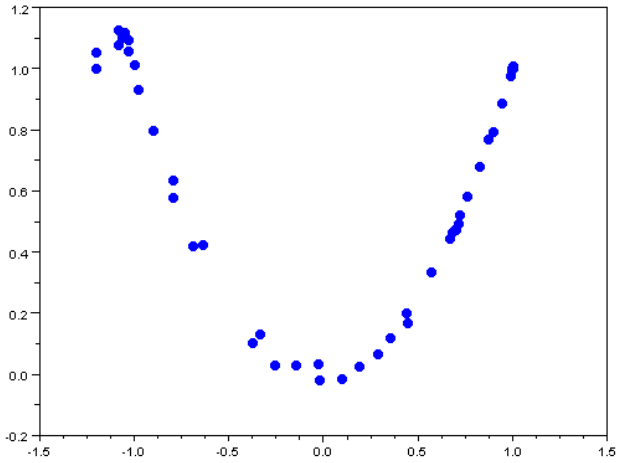

Fig. 5.3 : Plot produced by Scilab's fminsearch, with customized output function.

```
5  5.161796e+000  2 -expand- iter
7  4.497796e+000  3 -reflect- iter
9  4.497796e+000  4 -contract outside- iter
11  4.381360e+000  5 -contract inside- iter
13  4.245273e+000  6 -contract inside- iter
[...]
149  2.004302e-008  80 -contract inside- iter
151  1.122930e-009  81 -contract inside- iter
153  1.122930e-009  82 -contract outside- iter
155  1.122930e-009  83 -contract inside- iter
157  1.107549e-009  84 -contract outside- iter
159  8.177661e-010  85 -contract inside- iter
159  8.177661e-010  85 -- done
```

We see that the output produced by the two software are identical, expect for the two first lines and the last line. The lines #1 and #2 are different is because Scilab computes the function values of all the vertices before calling back the output function. The last line is different because Scilab considers that once the optimization is performed, the type of the step is an empty string. Instead, Matlab displays the type of the last performed step.

5.2.3 Predefined plot functions

Several pre-defined plot functions are provided with the *fminsearch* function. These functions are

- *optimplotfval*,
- *optimplotx*,
- *optimplotfunccount*.

In the following Matlab script, we use the *optimplotfval* pre-defined function.

```
% Matlab
options = optimset('PlotFcns',@optimplotfval);
[x fval] = fminsearch(banana , [-1.2, 1] , options)
```

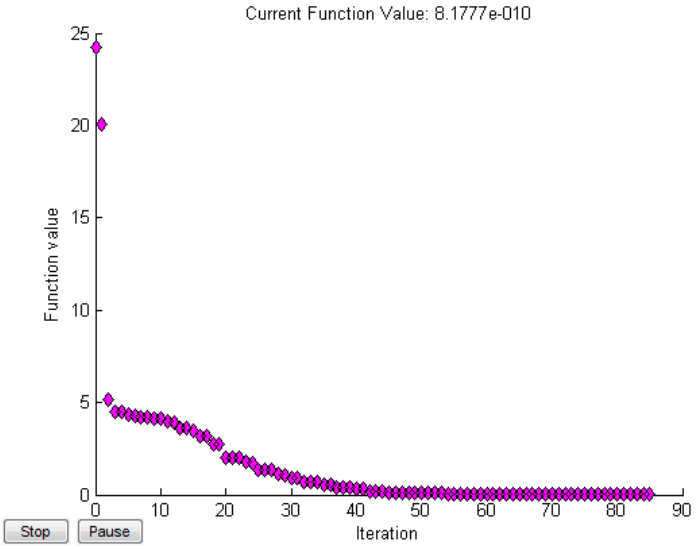

Fig. 5.4 : Plot produced by Matlab's fminsearch, with the *optimplotfval* function.

The previous script produces the plot which is presented in figure 5.4.

The following Scilab script uses the *optimplotfval* pre-defined function.

```
// Scilab
opt = optimset ( "OutputFcn" , optimplotfval );
[x fval] = fminsearch ( banana , [-1.2 1] , opt );
```

The previous script produces the plot which is presented in figure 5.5.

The comparison between the figures 5.4 and 5.5 shows that the two features produce very similar plots. Notice that Scilab's *fminsearch* does not provide the "Stop" and "Pause" buttons.

The figures 5.6 and 5.7 present the results of Scilab's *optimplotx* and *optimplotfunccount* functions.

5.3 Conclusion

The current version of Scilab's *fminsearch* provides the same algorithm as Matlab's *fminsearch*. The numerical precision is the same. The *optimset* and *optimget* functions allows to configure the optimization, as well as the output and plotting function. Pre-defined plotting function allows to get a fast and nice plot of the optimization.

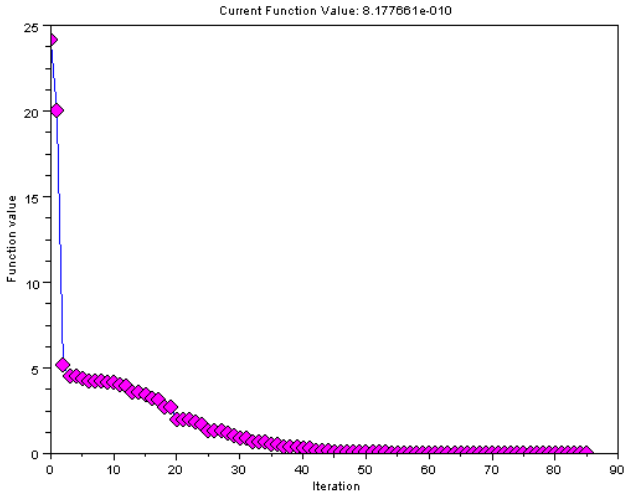

Fig. 5.5 : Plot produced by Scilab's fminsearch, with the *optimplotfval* function.

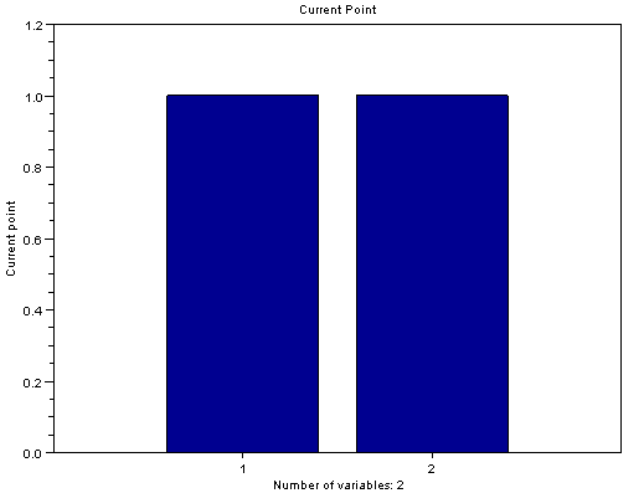

Fig. 5.6 : Plot produced by Scilab's fminsearch, with the *optimplotx* function.

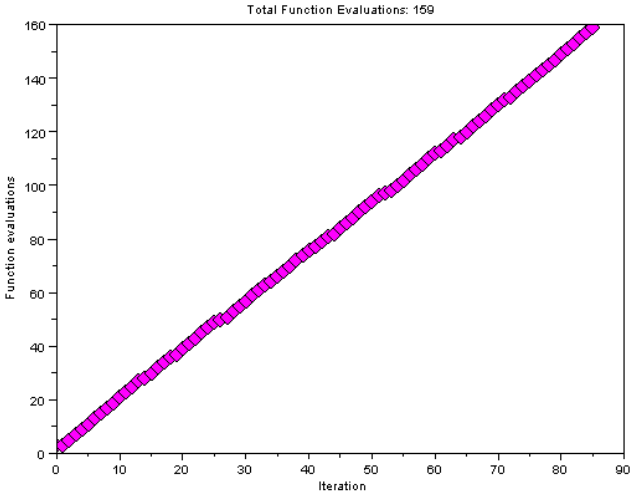

Fig. 5.7 : Plot produced by Scilab's fminsearch, with the *optimplotfunccount* function.

Chapter 6

Conclusion

That tool might be extended in future releases so that it provides the following features :

- Kelley restart based on simplex gradient [9],
- C-based implementation (a prototype is provided in appendix B),
- parallel implementation of the DIRECT algorithm,
- implementation of the Hook-Jeeves and Multidimensional Search methods [9]
- parallel implementation of the Nelder-Mead algorithm. See for example [21]. ?This paper generalizes the widely used Nelder and Mead (Comput J 7:308?313, 1965) simplex algorithm to parallel processors. Unlike most previous parallelization methods, which are based on parallelizing the tasks required to compute a specific objective function given a vector of parameters, our parallel simplex algorithm uses parallelization at the parameter level. Our parallel simplex algorithm assigns to each processor a separate vector of parameters corresponding to a point on a simplex. The processors then conduct the simplex search steps for an improved point, communicate the results, and a new simplex is formed. The advantage of this method is that our algorithm is generic and can be applied, without rewriting computer code, to any optimization problem which the non-parallel Nelder?Mead is applicable. The method is also easily scalable to any degree of parallelization up to the number of parameters. In a series of Monte Carlo experiments, we show that this parallel simplex method yields computational savings in some experiments up to three times the number of processors.?

Chapter 7

Acknowledgments

I would like to thank Vincent Couvert, the team manager for Scilab releases, for his support during the development of this software. I would like to thank Serge Steer, INRIA researcher, for his comments and the discussions on this subject. Professor Han, Associate Professor of Mathematics in the University of Michigan-Flint University was kind enough to send me a copy of his Phd and I would like to thank him for that. My colleagues Allan Cornet and Yann Collette helped me in many steps in the long process from the initial idea to the final release of the tool and I would like to thank them for their their time.

Appendix A

Nelder-Mead bibliography

In this section, we present a brief overview of selected papers, sorted in chronological order, which deal with the Nelder-Mead algorithm

A.1 Spendley, Hext, Himsworth, 1962

"Sequential Application of Simplex Designs in Optimisation and Evolutionary Operation", Spendley W., Hext G. R. and Himsworth F. R., American Statistical Association and American Society for Quality, 1962

This article [43] presents an algorithm for unconstrained optimization in which a simplex is used. The simplex has a fixed, regular (i.e. all lengths are equal), shape and is made of n+1 vertices (where n is the number of parameters to optimize). The algorithm is based on the reflection of the simplex with respect to the centroid of better vertices. One can add a shrink step so that the simplex size can converge to zero. Because the simplex shape cannot change, the convergence rate may be very slow if the eigenvalues of the hessian matrix have very different magnitude.

A.2 Nelder, Mead, 1965

"A Simplex Method for Function Minimization", Nelder J. A. and Mead R., The Computer Journal, 1965

This article [29] presents the Nelder-Mead unconstrained optimization algorithm. It is based on a simplex made of n+1 vertices and is a modification of the Spendley's et al algorithm. It includes features which enables the simplex to adapt to the local landscape of the cost function. The additional steps are expansion, inside contraction and outside contraction. The stopping criterion is based on the standard deviation of the function value on the simplex.

The convergence of the algorithm is better than Spendley's et al. The method is compared against Powell's free-derivative method (1964) with comparable behavior. The algorithm is

"greedy" in the sense that the expansion point is kept if it improves the best function value in the current simplex. Most Nelder-Mead variants which have been analyzed after are keeping the expansion point only if it improves over the reflection point.

A.3 Box, 1965

"A New Method of Constrained Optimization and a Comparison With Other Methods", M. J. Box, The Computer Journal 1965 8(1):42-52, 1965, British Computer Society

In this paper [4], Box presents a modification of the NM algorithm which takes into accounts for bound constraints and non-linear constraints. This variant is called the Complex method. The method expects that the initial guess satisfies the nonlinear constraints. The nonlinear constraints are supposed to define a convex set. The algorithm ensures that the simplex evolves in the feasible space.

The method to take into account for the bound constraints is based on projection of the parameters inside the bounded domain. If some nonlinear constraint is not satisfied, the trial point is moved halfway toward the centroid of the remaining points (which are all satisfying the nonlinear constraints).

The simplex may collapse into a subspace if a projection occurs. To circumvent this problem, k>=n+1 vertices are used instead of the original n+1 vertices. A typical value of k is k=2n. The initial simplex is computed with a random number generator, which takes into account for the bounds on the parameters. To take into account for the nonlinear constraints, each vertex of the initial simplex is moved halfway toward the centroid of the points satisfying the constraints (in which the initial guess already is).

A.4 Guin, 1968

"Discussion and correspondence: modification of the complex method of constrained optimization", J. A. Guin, The Computer Journal, 1968

In this article [11], Guin suggest 3 rules to improve the practical convergence properties of Box's complex method. These suggestions include the use of the next-to-worst point when the worst point does not produce an improvement of the function value. The second suggestion is to project the points strictly into the bounds, instead of projecting inside the bounds. The third suggestion is related to the failure of the method when the centroid is no feasible. In that case, Guin suggest to restrict the optimization in the subspace defined by the best vertex and the centroid.

A.5 O'Neill, 1971

"Algorithm AS47 - Function minimization using a simplex procedure", R. O'Neill, 1971, Applied Statistics

In this paper [31], R. O'Neill presents a fortran 77 implementation of the Nelder-Mead algorithm. The initial simplex is computed axis-by-axis, given the initial guess and a vector of step lengths. A factorial test is used to check if the computed optimum point is a local minimum.

A.6 Parkinson and Hutchinson, 1972

In [33], "An investigation into the efficiency of variants on the simplex method", Parkinson and Hutchinson explored several ways of improvement. First, they investigate the sensitivity of the algorithm to the initial simplex. Two parameters were investigated, i.e. the initial length and the orientation of the simplex. An automatic setting for the orientation, though very desirable, is not easy to design. Parkinson and Hutchinson tried to automatically compute the scale of the initial simplex by two methods, based on a "line search" and on a local "steepest descent". Their second investigation adds a new step to the algorithm, the unlimited expansion. After a sucessful expansion, the algorithm tries to produce an expansion point by taking the largest possible number of expansion steps. After an unlimited expansion steps is performed, the simplex is translated, so that excessive modification of the scale and shape is avoided. Combined and tested against low dimension problems, the modified algorithm, named PHS, provides typical gains of 20function evaluations.

A.7 Richardson and Kuester, 1973

"Algorithm 454: the complex method for constrained optimization", Richardson Joel A. and Kuester J. L., Commun. ACM, 1973

In this paper [38], Richardson and Kuester shows a fortran 77 implementation of Box's complex optimization method. The paper clarifies several specific points from Box's original paper while remaining very close to it. Three test problems are presented with the specific algoritmic settings (such as the number of vertices for example) and number of iterations.

A.8 Shere, 1973

"Remark on algorithm 454 : The complex method for constrained optimization", Shere Kenneth D., Commun. ACM, 1974

In this article [41], Shere presents two counterexamples where the algorithm 454, implemented by Richardson and Kuester produces an infinite loop. "This happens whenever the corrected point, the centroid of the remaining complex points, and every point on the line segment joining these

two points all have functional values lower than the functional values at each of the remaining complex points.

A.9 Routh, Swartz, Denton, 1977

"Performance of the Super-Modified Simplex", M.W. Routh, P.A. Swartz, M.B. Denton, Analytical Chemistry, 1977

In this article [40], Routh, Swartz and Denton present a variant of the Nelder-Mead algorithm, which is called the Modified Simplex Method (SMS) in their paper. The algorithm is modified in the following way. After determination of the worst response (W), the responses at the centroid (C) and reflected (R) vertices are measured and a second-order polynomial curve is fitted to the responses at W, C and R. Furthermore, the curve is extrapolated beyond W and R by a percentage of the W-R vector resulting in two types of curve shapes. In the concave down case, a maximum occurs within the interval. Assuming a maximization process, evaluation of the derivative of the curve reveals the location of the predicted optimum whose response is subsequently evaluated, the new vertex is located at that position, and the optimization process is continued. In the concave up case, a response maximum does not occur within the interval so the extended interval boundary producing the highest predicted response is chosen as the new vertex location, its response is determined, and the optimization is continued. If the response at the predicted extended interval boundary location does not prove to be greater than the response at R, the vertex R may instead be retained as the new vertex and the process continued. The slope at the extended interval boundary may additionally be evaluated dictating the magnitude of the expansion coefficient, i.e. the greater the slope (indicating rapid approach to the optimum location), the smaller the required expansion coefficient and, conversely, the smaller the slope (indicating remoteness from the optimum location), the larger the required expansion coefficient.

Some additional safeguard procedure must be used in order to prevent the collapse of the simplex.

A.10 Van Der Wiel, 1980

"Improvement of the Super-Modified Simplex Optimization Procedure", P.F.A., Van Der Wiel Analytica Chimica Acta, 1980

In this article [47], Van Der Wiel tries to improve the SMS method by Routh et al.. His modifications are based on a Gaussian fit, weighted reflection point and estimation of response at the reflection point. Van Der Wiel presents a simplified pseudo-code for one algorithm The method is tested in 5 cases, where the cost function is depending on the exponential function.

A.11 Walters, Parker, Morgan and Deming, 1991

"Sequential Simplex Optimization for Quality and Productivity in Research, Development, and Manufacturing", F. S. Walters, L. R. Parker, Jr., S. L. Morgan, and S. N. Deming, 1991

In this book [48], Walters, Parker, Morgan and Deming give a broad view on the simplex methods in chemistry. The Spendley et al. and Nelder-Mead algorithms are particularily deeply analyzed, with many experiments analyzed in great detail. Template tables are given, so that an engineer can manually perform the optimization and make the necessary calculations. Practical advices are given, which allow to make a better use of the algorithms.

In chapter 5, "Comments on Fixed-size and Variable-size Simplexes", comparing the path of the two algorithms allows to check that a real optimum has been found. When the authors analyze the graph produced by the response depending on the number of iteration, the general behavior of the fixed-size algorithm is made of four steps. Gains in response are initially rapid, but the rate of return decreases as the simplex probes to find the ridge and then moves along the shallower ridge to find the optimum. The behavior from different starting locations is also analyzed. Varying the size of the initial simplex is also analyzed for the fixed-size simplex algorithm. The many iterations which are produced when a tiny initial simplex is used with the fixed-size simplex is emphasized.

The chapter 6, "General Considerations", warns that the user may setup an degenerate initial simplex, leading to a false convergence of the algorithm. Various other initial simplices are analyzed. Modifications in the algorithm to take into account for bounds contraints are presented. The behavior of the fixed-size and variable-size simplex algorithms is analyzed when the simplex converges. The "k+1" rule, introduced by Spendley et al. to take into account for noise in the cost function is presented.

The chapter 7, "Additional Concerns and Topics" deals with advanced questions regarding these algorithms. The variable size simplex algorithm is analyzed in the situation of a ridge. Partially oscillatory collapse of the Nelder-Mead algorithm is presented. The same behavior is presented in the case of a saddle point. This clearly shows that practionners were aware of the convergence problem of this algorithm well before Mc Kinnon presented a simple counter example (in 1998). The "Massive Contraction" step of Nelder and Mead is presented as a solution for this oscillatory behavior. The authors present a method, due to Ernst, which allows to keep the volume of the simplex, instead of shrinking it. This method is based on a translation of the simplex. This modification requires $n + 1$ function evaluations. A more efficient method, due to King, is based on reflection with respect to the next-to-worst vertex. This modification was first suggested by Spendley et al. in their fixed-size simplex algorithm.

In the same chapter, the authors present the behavior of the algorithms in the case of multiple optima. They also present briefly other types of simplex algorithms.

A complete bibliography (from 1962 to 1990) on simplex-based optimization is given in the end of the book.

A.12 Subrahmanyam, 1989

"An extension of the simplex method to constrained nonlinear optimization", M. B. Subrahmanyam, Journal of Optimization Theory and Applications, 1989

In this article [44], the simplex algorithm of Nelder and Mead is extended to handle nonlinear optimization problems with constraints. To prevent the simplex from collapsing into a subspace near the constraints, a delayed reflection is introduced for those points moving into the infeasible region. Numerical experience indicates that the proposed algorithm yields good results in the presence of both inequality and equality constraints, even when the constraint region is narrow.

If a vertex becomes infeasible, we do not increase the value at this vertex until the next iteration is completed. Thus, the next iteration is accomplished using the actual value of the function at the infeasible point. At the end of the iteration, in case the previous vertex is not the worst vertex, it is assigned a high value, so that it then becomes a candidate for reflection during the next iteration.

The paper presents numerical experiments which are associated with thousands of calls to the cost function. This may be related with the chosen reflection factor equal to 0.95, which probably cause a large number of reflections until the simplex can finally satisfy the constraints.

A.13 Numerical Recipes in C, 1992

"Numerical Recipes in C, Second Edition", W. H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, 1992

In this book [37], an ANSI C implementation of the Nelder-Mead algorithm is given. The initial simplex is based on the axis. The termination criterion is based on the relative difference of the function value of the best and worst vertices in the simplex.

A.14 Lagarias, Reeds, Wright, Wright, 1998

"Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions", Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright and Paul E. Wright, SIAM Journal on Optimization, 1998

This paper [19] presents convergence properties of the Nelder-Mead algorithm applied to stricly convex functions in dimensions 1 and 2. Proofs are given to a minimizer in dimension 1, and various limited convergence results for dimension 2.

A.15 Mc Kinnon, 1998

"Convergence of the Nelder–Mead Simplex Method to a Nonstationary Point", SIAM J. on Optimization, K. I. M. McKinnon, 1998

In this article [21], Mc Kinnon analyzes the behavior of the Nelder-Mead simplex method for a family of examples which cause the method to converge to a nonstationnary point. All the examples use continuous functions of two variables. The family of functions contains strictly convex functions with up to three continuous derivatives. In all the examples, the method repeatedly applies the inside contraction step with the best vertex remaining fixed. The simplices tend to a straight line which is orthogonal to the steepest descent direction. It is shown that this behavior cannot occur for functions with more than three continuous derivatives.

A.16 Kelley, 1999

"Detection and Remediation of Stagnation in the Nelder–Mead Algorithm Using a Sufficient Decrease Condition", SIAM J. on Optimization, Kelley, C. T., 1999

In this article [17], Kelley presents a test for sufficient decrease which, if passed for the entire iteration, will guarantee convergence of the Nelder-Mead iteration to a stationary point if the objective function is smooth. Failure of this condition is an indicator of potential stagnation. As a remedy, Kelley propose to restart the algorithm with an oriented simplex, smaller than the previously optimum simplex, but with a better shape and which approximates the steepest descent step from the current best point. The method is experimented against Mc Kinnon test function and allow to converge to the optimum, where the original Nelder -Mead algorithm was converging to a non-stationary point. Although the oriented simplex works well in practice, other strategies may be chosen with similar results, such as a simplex based on axis, a regular simplex (like Spendley's) or a simplex based on the variable magnitude (like Pfeffer's suggestion in Matlab's fminsearch). The paper also shows one convergence theorem which prove that if the sufficient decrease condition is satisfied and if the product of the condition of the simplex by the simplex size converge to zero, therefore, with additional assumptions on the cost function and the sequence of simplices, any accumulation point of the simplices is a critical point of f.

The same ideas are presented in the book [18].

A.17 Han, 2000

In his Phd thesis [12], Lixing Han analyzes the properties of the Nelder-Mead algorithm. Han present two examples in which the Nelder-Mead simplex method does not converge to a single point. The first example is a nonconvex function with bounded level sets and it exhibits similar nonconvergence properties with the Mc Kinnon counterexample $f(\xi_1, \xi_2) = \xi_1^2 - \xi_2(\xi_2 - 2)$. The second example is a convex function with bounded level sets, for which the Nelder-Mead simplices converge to a degenerate simplex, but not to a single point. These nonconvergent examples support the observations by some practitionners that in the Nelder-Mead simplices may collapse into a degenerate simplex and therefore support the use of a restart strategy. Han also investigates the effect of the dimensionality of the Nelder-Mead method. It is shown that the Nelder-Mead

simplex method becomes less efficient as the dimension increases. Specifically, Han consider the quadratic function $\xi_1^2 + \ldots + \xi_1^n$ and shows that the Nelder-Mead method becomes less efficient as the dimension increases. The considered example offers insight into understanding the effect of dimensionnality on the Nelder-Mead method. Given all the known failures and inefficiencies of the Nelder-Mead method, a very interesting question is why it is so popular in practice. Han present numerical results of the Nelder-Mead method on the standard collection of Moré-Garbow-Hillstrom with dimensions $n \leq 6$. Han compare the Nelder-Mead method with a finite difference BFGS method and a finite difference steepest descent method. The numerical results show that the Nelder-Mead method is much more efficient than the finite difference steepest descent method for the problems he tested with dimensions $n \leq 6$. It is also often comparable with the finite difference BFGS method, which is believed to be the best derivative-free method. Some of these results are reproduced in [13] by Han and Neumann, "Effect of dimensionality on the Nelder-ĂŞMead simplex method" and in [14], "On the roots of certain polynomials arising from the analysis of the Nelder-Mead simplex method".

A.18 Nazareth, Tseng, 2001

"Gilding the Lily: A Variant of the Nelder-Mead Algorithm Based on Golden-Section Search" Computational Optimization and Applications, 2001, Larry Nazareth and Paul Tseng

The article [28] propose a variant of the Nelder-Mead algorithm derived from a reinterpretation of univariate golden-section direct search. In the univariate case, convergence of the variant can be analyzed analogously to golden-section search.

The idea is based on a particular choice of the reflection, expansion, inside and outside contraction parameters, based on the golden ratio. This variant of the Nelder-Mead algorithm is called Nelder-Mead-Golden- Ratio, or NM-GS. In one dimension, the authors exploit the connection with golden-search method and allows to prove a convergence theorem on unimodal univariate functions. This is marked contrast to the approach taken by Lagarias et al. where considerable effort is expended to show convergence of the original NM algorithm on strictly convex univariate functions. With the NM-GS variant, one obtain convergence in the univariate case (using a relatively simple proof) on the broader class of unimodal functions.

In the multivariate case, the authors modify the variant by replacing strict descent with fortified descent and maintaining the interior angles of the simplex bounded away from zero. Convergence of the modified v ariant can be analyzed by applying results for a fortified- descent simplicial search method. Some numerical experience with the variant is reported.

A.19 Perry, Perry, 2001

"A New Method For Numerical Constrained Optimization" by Ronald N. Perry, Ronald N. Perry, March 2001

In this report [34], we propose a new method for constraint handling that can be applied to established optimization algorithms and which significantly improves their ability to traverse through constrained space. To make the presentation concrete, we apply the new constraint method to the Nelder and Mead polytope algorithm. The resulting technique, called SPIDER, has shown great initial promise for solving difficult (e.g., nonlinear, nondifferentiable, noisy) constrained problems.

In the new method, constraints are partitioned into multiple levels. A constrained performance, independent of the objective function, is defined for each level. A set of rules, based on these partitioned performances, specify the ordering and movement of vertices as they straddle constraint boundaries; these rules [...] have been shown to significantly aid motion along constraints toward an optimum. Note that the new approach uses not penalty function and thus does not warp the performance surface, thereby avoiding the possible ill-conditioning of the objective function typical in penalty methods.

No numerical experiment is presented.

A.20 Andersson, 2001

"Multiobjective Optimization in Engineering Design - Application to fluid Power Systems" Johan Andersson, 2001

This PhD thesis [2] gives a brief overview of the Complex method by Box in section 5.1.

A.21 Peters, Bolte, Marschner, Nüssen and Laur, 2002

In [35], "Enhanced Optimization Algorithms for the Development of Microsystems", the authors combine radial basis function interpolation methods with the complex algorithm by Box. Interpolation with radial basis functions is a linear approach in which the model function f is generated via the weighted sum of the basis functions $\Phi_i(r)$. The parameter r describes the distance of the current point from the center x_i of the ith basis function. It is calculated via the euclidean norm. It is named ComplInt strategy. The name stands for Complex in combination with interpolation. The Complex strategy due to Box is very well suited for the combination with radial basis function interpolation for it belongs to the polyhedron strategies. The authors presents a test performed on a pratical application, which leaded them to the following comment : "The best result achieved with the ComplInt strategy is not only around 10% better than the best result of the Complex strategy due to Box, the ComplInt also converges much faster than the Complex does: while the Complex strategy needs an average of 7506, the ComplInt only calls for an average of 2728 quality function evaluations."

A.22 Han, Neumann, 2006

"Effect of dimensionality on the Nelder-Mead simplex method", L. Han and M. Neumann (2006),

In this article [13], the effect of dimensionality on the Nelder-Mead algorithm is investigated. It is shown that by using the quadratic function $f(x) = x^T * x$, the Nelder-Mead simplex method deteriorates as the dimension increases. More precisely, in dimension 1, with the quadratic function $f(x) = x^2$ and a particular choice of the initial simplex, applies inside contraction step repeatedly and the convergence rate (as the ratio between the length of the simplex at two consecutive steps) is 1/2. In dimension 2, with a particular initial simplex, the NM algorithm applies outside contraction step repeatedly and the convergence rate is $\sqrt{(2)}/2$.

For n>=3, a numerical experiment is performed on the quadratic function with the fminsearch algorithm from Matlab. It is shown that the original NM algorithm has a convergence rate which is converging towards 1 when n increases. For n=32, the rate of convergence is 0.9912.

A.23 Singer, Nelder, 2008

http://www.scholarpedia.org/article/Nelder-Mead_algorithm Singer and Nelder

This article is a complete review of the Nelder-Mead algorithm. Restarting the algorithm is adviced when a premature termination occurs.

Appendix B

Implementations of the Nelder-Mead algorithm

In the following sections, we analyze the various implementations of the Nelder-Mead algorithm. We analyze the Matlab implementation provided by the *fminsearch* command. We analyze the matlab algorithm provided by C.T. Kelley and the Scilab port by Y. Collette. We present the Numerical Recipes implementations. We analyze the O'Neill fortran 77 implementation "AS47". The Burkardt implementation is also covered. The implementation provided in the NAG collection is detailed. The Nelder-Mead algorithm from the Gnu Scientific Library is analyzed.

B.1 Matlab : fminsearch

The Matlab command fminsearch implements the Nelder-Mead algorithm [20]. It provides features such as

- maximum number of function evaluations,
- maximum number of iterations,
- termination tolerance on the function value,
- termination tolerance on x,
- output command to display the progress of the algorithm.

B.2 Kelley and the Nelder-Mead algorithm

C.T. Kelley has written a book [18] on optimization method and devotes a complete chapter to direct search algorithms, especially the Nelder-Mead algorithm. Kelley provides in [16] the Matlab implementation of the Nelder-Mead algorithm. That implementation uses the restart

strategy that Kelley has published in [17] and which improves the possible stagnation of the algorithm on non local optimization points. No tests are provided.

The following is extracted from the README provided with these algorithms.

```
These files are current as of December 9, 1998.

-----------------

MATLAB/FORTRAN software for Iterative Methods for Optimization

by C. T. Kelley

These M-files are implementations of the algorithms from the book
"Iterative Methods for Optimization", to be published by SIAM,
by C. T. Kelley. The book, which describes the algorithms, is available
from SIAM (service@siam.org). These files can be modified for non-commercial
purposes provided that the authors:

C. T. Kelley for all MATLAB codes,
P. Gilmore and T. D. Choi for iffco.f
J. M. Gablonsky for DIRECT

are acknowledged and clear comment lines are inserted
that the code has been changed. The authors assume no no responsibility
for any errors that may exist in these routines.

Questions, comments, and bug reports should be sent to

Professor C. T. Kelley
Department of Mathematics, Box 8205
North Carolina State University
Raleigh, NC 27695-8205

(919) 515-7163
(919) 515-3798 (FAX)

Tim_Kelley@ncsu.edu
```

From Scilab's point of view, that ?licence? is a problem since it prevents the use of the source for commercial purposes.

B.3 Nelder-Mead Scilab Toolbox : Lolimot

The Lolimot project by Yann Collette provide two Scilab-based Nelder- Mead implementations [5]. The first implementation is a Scilab port of the Kelley script. The licence problem is therefore not solved by this script. The second implementation [6] implements the restart strategy by Kelley. No tests are provided.

B.4 Numerical Recipes

The Numerical Recipes [37] provides the C source code of an implementation of the Nelder-Mead algorithm. Of course, this is a copyrighted material which cannot be included in Scilab.

B.5 NASHLIB : A19

Nashlib is a collection of Fortran subprograms from "Compact Numerical Methods for Computers; Linear Algebra and Function Minimisation, " by J.C. Nash. The subprograms are written without many of the extra features usually associated with commercial mathematical software, such as extensive error checking, and are most useful for those applications where small program size is particularly important. The license is public domain.

Nahslib includes one implementation of the Nelder-Mead algorithm [26], [27]. It is written in fortran 77. The coding style is "goto"-based and may not be easy to maintain.

B.6 O'Neill implementations

The paper [31] by R. O'Neil in the journal of Applied Statistics presents a fortran 77 implementation of the Nelder-Mead algorithm. The source code itself is available in [30]. Many of the following implementations are based on this primary source code. We were not able to get the paper [31] itself.

On his website, John Burkardt gives a fortran 77 source code of the Nelder-Mead algorithm [32]. The following are the comments in the header of the source code.

```
c  Discussion:
c
c    This routine seeks the minimum value of a user-specified function.
c
c     Simplex function minimisation procedure due to Nelder+Mead(1965),
c     as implemented by O'Neill(1971, Appl.Statist. 20, 338-45), with
c     subsequent comments by Chambers+Ertel(1974, 23, 250-1), Benyon(1976,
```

```
c     25, 97) and Hill(1978, 27, 380-2)
c
c    The function to be minimized must be defined by a function of
c    the form
c
c      function fn ( x, f )
c      double precision fn
c      double precision x(*)
c
c    and the name of this subroutine must be declared EXTERNAL in the
c    calling routine and passed as the argument FN.
c
c    This routine does not include a termination test using the
c    fitting of a quadratic surface.
c
c  Modified:
c
c    27 February 2008
c
c  Author:
c
c    FORTRAN77 version by R ONeill
c    Modifications by John Burkardt
```

The "Bayesian Survival Analysis" book by Joseph G. Ibrahim, Ming-Hui Chen, and Debajyoti Sinha provides in [1] a fortran 77 implementation of the Nelder-Mead algorithm. The following is the header of the source code.

```
c     Simplex function minimisation procedure due to Nelder+Mead(1965),
c     as implemented by O'Neill(1971, Appl.Statist. 20, 338-45), with
c     subsequent comments by Chambers+Ertel(1974, 23, 250-1), Benyon(1976,
c     25, 97) and Hill(1978, 27, 380-2)
```

The O'Neill implementation uses a restart procedure which is based on a local axis by axis search for the optimality of the computed optimum.

B.7 Burkardt implementations

John Burkardt gives several implementations of the Nelder-Mead algorithm

- in fortran 77 [32]
- in Matlab by Jeff Borggaard [3].

B.8 NAG Fortran implementation

The NAG Fortran library provides the E04CCF/E04CCA routines [24] which implements the simplex optimization method. E04CCA is a version of E04CCF that has additional parameters in order to make it safe for use in multithreaded applications. As mentioned in the documentation, "The method tends to be slow, but it is robust and therefore very useful for functions that are subject to inaccuracies.". The termination criteria is based on the standard deviation of the function values of the simplex.

The specification of the cost function for E04CCA is:

```
SUBROUTINE FUNCT ( N, XC, FC, IUSER, RUSER)
```

where IUSER and RUSER and integer and double precision array, which allow the user to supply information to the cost function. An output routine, called MONIT is called once every iteration in E04CCF/E04CCA. It can be used to print out the current values of any selection of its parameters but must not be used to change the values of the parameters.

B.9 GSL implementation

The Gnu Scientific Library provides two Nelder-Mead implementations. The authors are Tuomo Keskitalo, Ivo Alxneit and Brian Gough. The size of the simplex is the root mean square sum of length of vectors from simplex center to corner points. The termination criteria is based on the size of the simplex.

The C implementation of the minimization algorithm is original. The communication is direct, in the sense that the specific optimization algorithm calls back the cost function. A specific optimization implementation provides four functions : "alloc", "free", "iterate" and "set". A generic optimizer is created by connecting it to a specific optimizer. The user must write the loop over the iterations, making successive calls to the generic "iterate" function, which, in turns, calls the specific "iterate" associated with the specific optimization algorithm.

The cost function can be provided as three function pointers

- the cost function f,
- the gradient g,
- both the cost function and the gradient.

Some additional parameters can be passed to these functions.

Bibliography

[1] optim1.f. http://www.stat.uconn.edu/~mhchen/survbook/example51/optim1.f.

[2] Johan Andersson and Linköpings Universitet. Multiobjective optimization in engineering design: Application to fluid power systems. Technical report, Department of Mechanical Engineering, Linköping University, 2001. https://polopoly.liu.se/content/1/c6/10/99/74/phdthesis.pdf.

[3] Jeff Borggaard. nelder_mead. January 2009. http://people.sc.fsu.edu/~burkardt/m_src/nelder_mead/nelder_mead.m.

[4] M. J. Box. A new method of constrained optimization and a comparison with other methods. *The Computer Journal*, 8(1):42–52, 1965.

[5] Yann Collette. Lolimot. http://sourceforge.net/projects/lolimot/.

[6] Yann Collette. Lolimot - optim_nelder_mead.sci. http://lolimot.cvs.sourceforge.net/viewvc/lolimot/scilab/optim/macros/optim_nelder_mead.sci?revision=1.1.1.1&view=markup.

[7] Ellen Fan. Global optimization of lennard-jones atomic clusters. Technical report, McMaster University, February 2002.

[8] R. Fletcher and M. J. D. Powell. A Rapidly Convergent Descent Method for Minimization. *The Computer Journal*, 6(2):163–168, 1963.

[9] P. E. Gill, W. Murray, and M. H. Wright. *Practical optimization*. Academic Press, London, 1981.

[10] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[11] J. A. Guin. Discussion and correspondence: modification of the complex method of constrained optimization. *The Computer Journal*, 10(4):416–417, February 1968.

[12] Lixing Han. *Algorithms in Unconstrained Optimization*. Ph.D., The University of Connecticut, 2000.

[13] Lixing Han and Michael Neumann. Effect of dimensionality on the nelder-mead simplex method. *Optimization Methods and Software*, 21(1):1–16, 2006.

[14] Lixing Han, Michael Neumann, and Jianhong Xu. On the roots of certain polynomials arising from the analysis of the nelder-mead simplex method. *Linear Algebra and its Applications*, 363:109–124, 2003.

[15] Stephen J. Wright Jorge Nocedal. *Numerical Optimization*. Springer, 1999.

[16] C. T. Kelley. *Iterative Methods for Optimization: Matlab Codes*. North Carolina State University.

[17] C. T. Kelley. Detection and remediation of stagnation in the nelder–mead algorithm using a sufficient decrease condition. *SIAM J. on Optimization*, 10(1):43–55, 1999.

[18] C. T. Kelley. *Iterative Methods for Optimization*, volume 19. SIAM Frontiers in Applied Mathematics, 1999.

[19] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the nelder–mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1998.

[20] The Mathworks. Matlab ? fminsearch. http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/ref/fminsearch.html.

[21] K. I. M. McKinnon. Convergence of the nelder–mead simplex method to a nonstationary point. *SIAM J. on Optimization*, 9(1):148–158, 1998.

[22] J. J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Algorithm 566: Fortran subroutines for testing unconstrained optimization software [c5], [e4]. *ACM Trans. Math. Softw.*, 7(1):136–140, 1981.

[23] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Testing unconstrained optimization software. *ACM Trans. Math. Softw.*, 7(1):17–41, 1981.

[24] NAG. Nag fortran library routine document : E04ccf/e04cca. http://www.nag.co.uk/numeric/Fl/manual/xhtml/E04/e04ccf.xml.

[25] J. C. Nash. *Compact numerical methods for computers : linear algebra and function minimisation*. Hilger, Bristol, 1979.

[26] J.C. Nash. Gams - a19a20 - description. February 1980. http://gams.nist.gov/serve.cgi/Module/NASHLIB/A19A20/11238/.

[27] J.C. Nash. Gams - a19a20 - source code. February 1980. http://gams.nist.gov/serve.cgi/ModuleComponent/11238/Source/ITL/A19A20.

[28] Larry Nazareth and Paul Tseng. Gilding the lily: A variant of the nelder-mead algorithm based on golden-section search. *Comput. Optim. Appl.*, 22(1):133–144, 2002.

[29] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[30] R. O'Neill. Algorithm as47 - fortran 77 source code. 1971. http://lib.stat.cmu.edu/apstat/47.

[31] R. O'Neill. Algorithm AS47 - Function minimization using a simplex procedure. *Applied Statistics*, 20(3):338–346, 1971.

[32] R. O'Neill and John Burkardt. Gams - a19a20 - source code. 2008. http://people.sc.fsu.edu/~burkardt/f77_src/asa047/asa047.f.

[33] Parkinson and Hutchinson. An investigation into the efficiency of variants on the simplex method. *F. A. Lootsma, editor, Numerical Methods for Non-linear Optimization*, pages 115–135, 1972.

[34] Ronald N. Perry and Ronald N. Perry. A new method for numerical constrained optimization, 2001.

[35] D. Peters, H. Bolte, C. Marschner, O. Nüssen, and R. Laur. Enhanced optimization algorithms for the development of microsystems. *Analog Integr. Circuits Signal Process.*, 32(1):47–54, 2002.

[36] M. J. D. Powell. An Iterative Method for Finding Stationary Values of a Function of Several Variables. *The Computer Journal*, 5(2):147–151, 1962.

[37] W. H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, Second Edition*. 1992.

[38] Joel A. Richardson and J. L. Kuester. Algorithm 454: the complex method for constrained optimization. *Commun. ACM*, 16(8):487–489, 1973.

[39] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, March 1960.

[40] M. W. Routh, P.A. Swartz, and M.B. Denton. Performance of the super modified simplex. *Analytical Chemistry*, 49(9):1422–1428, 1977.

[41] Kenneth D. Shere. Remark on algorithm 454 : The complex method for constrained optimization. *Commun. ACM*, 17(8):471, 1974.

[42] A. Singer and J. Nelder. Nelder-mead algorithm. *Scholarpedia*, 4(7):2928, 2009.

[43] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. *Technometrics*, 4(4):441–461, 1962.

[44] M. B. Subrahmanyam. An extension of the simplex method to constrained nonlinear optimization. *J. Optim. Theory Appl.*, 62(2):311–319, 1989.

[45] Virginia Torczon and Michael W. Trosset. From evolutionary operation to parallel direct search: Pattern search algorithms for numerical optimization. *Computing Science and Statistics*, 29:396–401, 1998.

[46] Virginia Joanne Torczon. Multi-directional search: A direct search algorithm for parallel machines. Technical report, Rice University, 1989.

[47] P.F.A. Van Der Wiel. Improvement of the super modified simplex optimisation procedure. *Analytica Chemica Acta*, 122:421?–433, 1980.

[48] F. S. Walters, L. R. Parker, S. L. Morgan, and S. N. Deming. *Sequential Simplex Optimization for Quality and Productivity in Research, Development, and Manufacturing.* Chemometrics series. CRC Press, Boca Raton, FL, 1991.

Index