Appl. Num. Anal. Comp. Math. **1**, No. 2, 524 – 534 (2004) / **DOI** 10.1002/anac.200410015

Efficient Implementation of the Nelder–Mead Search Algorithm

Saša Singer[*1] and **Sanja Singer**[**2]

[1] Department of Mathematics, University of Zagreb, P.O. Box 335, 10002 Zagreb, Croatia.

[2] Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, I. Lučića 5, 10000 Zagreb, Croatia.

Received 30 October 2004, revised 30 November 2004, accepted 5 December 2004
Published online 20 December 2004

Key words optimization, direct search methods, Nelder–Mead algorithm, complexity, termination test.
AMS 90C56, 65Y20, 65K10, 90C90

The Nelder–Mead or simplex search algorithm is one of the best known algorithms for unconstrained optimization of non–smooth functions. Even though the basic algorithm is quite simple, it is implemented in many different ways. Apart from some minor computational details, the main difference between various implementations lies in the selection of convergence (or termination) tests, which are used to break the iteration process.

A fairly simple efficiency analysis of each iteration step reveals a potential computational bottleneck in the domain convergence test. To be efficient, such a test has to be sublinear in the number of vertices of the working simplex. We have tested some of the most common implementations of the Nelder–Mead algorithm, and none of them is efficient in this sense.

Therefore, we propose a simple and efficient domain convergence test and discuss some of its properties. This test is based on tracking the volume of the working simplex throughout the iterations. Similar termination tests can also be applied in some other simplex–based direct search methods.

1 Introduction

The classical unconstrained optimization problem is to locate a point of minimum x^* of a given (nonlinear) function $f : \mathbb{R}^n \to \mathbb{R}$. If f is non–smooth or even discontinuous at some points in $\mathbb{R}^n$, the optimization method should use only the function values of f, since the derivatives of f may not exist at a particular point. Methods of this type are usually called Direct Search Methods (DSM).

The Nelder–Mead search (NMS) or simplex search [6] is one of the best known and most widely used methods in this class. Originally published in 1965, the method is nowadays a standard member of all major libraries and has many different implementations. Apart from some minor computational details in the basic algorithm, the main difference between various implementations lies in the selection of convergence (or termination) tests, which are used to break the iteration process.

Despite its popularity, there is a widespread belief that NMS becomes inefficient as n increases, even for moderate space dimensions (like $n \geq 10$). This belief is mainly based on extensive numerical evidence, but is hard to substantiate mathematically, due to the lack of convergence theory. Rigorous analysis of the Nelder–Mead simplex method seems to be a very hard problem. So far, there are convergence results only for low dimensions (1 and 2) by Lagarias et al. in a very readable paper [3], and there is a counterexample by McKinnon [5]. Unfortunately, these convergence results are valid only for strictly convex functions.

Almost nothing is known about behaviour of the Nelder–Mead method for discontinuous functions, which occur quite frequently in experimental mathematics. And that is where we experienced the inefficiency of the Nelder–Mead algorithm. While trying to develop some new pivoting strategies for structured problems in numerical linear algebra, we have compared several DSMs on a series of problems with discontinuous functions f

* Corresponding author: e-mail: singer@math.hr, Phone: +385 1 4605 745, Fax: +385 1 4680 335
** e-mail: ssinger@math.hr, Phone: +385 1 6168 215, Fax: +385 1 6156 940

(see [10]). Contrary to all our expectations, NMS turned out to be (by far) the slowest method, so we decided to determine the cause, and do something about it, if possible.

Computational efficiency analysis of a single NMS iteration in [11] shows that efficient implementation of the domain convergence test is crucial for overall efficiency. Moreover, most common implementations are either inefficient for discontinuous functions, or unable to handle such functions at all. To circumvent this problem, here we propose a simple and efficient domain convergence test and discuss some implementation details.

We begin by a brief description of the NMS algorithm, followed by a summary of efficiency results from [11]. This section also shows how to efficiently implement the simplex transformation part of the NMS. Then we describe the relative volume termination test. Finally, we present some numerical examples (based on [10]) which illustrate improvements in efficiency of the NMS.

2 The Nelder–Mead search algorithm

A simplex $S \subset \mathbb{R}^n$ is defined as the convex hull of $n+1$ points or vertices $x_0, \ldots, x_n \in \mathbb{R}^n$ and will be denoted by $S = S(x_0, \ldots, x_n)$. Simplex based direct search algorithms (including the NMS) perform certain transformations of the working simplex S, based on function values $f_j := f(x_j)$, for $j = 0, \ldots, n$, and return a point $x_{\rm final} \in \mathbb{R}^n$, which is the computed approximation for x^*. The general algorithm is

Algorithm Simplex DSM

INIT: construct an initial working simplex $S_{\rm init}$;
repeat the following steps: { next iteration }
  TERM: calculate termination test information;
  if the termination test is not satisfied **then**
    TRANSF: transform the working simplex;
until the termination test is satisfied;
$x_{\rm final}$:= the best vertex of the current simplex S;

where the best vertex is, obviously, the one with the smallest function value.

Algorithm INIT constructs the initial simplex $S_{\rm init} = S(x_0, \ldots, x_n)$ around (or near) the initial point $x_{\rm init}$ which is usually given as an input, and computes the function values f_j, $j = 0, \ldots, n$, at all the vertices. The most frequent choice is $x_0 = x_{\rm init}$ to allow proper restarts. Usually, $S_{\rm init}$ is chosen to be right–angled at x_0, based on coordinate axes, or

$$x_j := x_0 + h_j e_j, \quad j = 1, \ldots, n,$$

with stepsizes h_j in directions of unit vectors e_j in $\mathbb{R}^n$. In some implementations, $S_{\rm init}$ can be a regular simplex, where all edges have the same length.

The inner loop algorithm TRANSF determines the type of the simplex based DSM. In all implementations of the NMS, TRANSF consists of the following 3 steps.

1. Determine indices h, s, l of the worst, second–worst and the best point, respectively

$$f_h = \max_j f_j, \quad f_s = \max_{j \neq h} f_j, \quad f_l = \min_{j \neq h} f_j.$$

2. Calculate the centroid c of the best side (this is the one opposite to the worst point x_h)

$$c := \frac{1}{n} \sum_{\substack{j=0 \\ j \neq h}}^{n} x_j. \tag{1}$$

3. Compute the new working simplex S from the old one, denoted by S'. First, try to replace the worst point x_h with a better point $x_{\rm new}$, by using reflection, expansion or contraction with respect to the best side. If this fails, shrink the simplex towards the best point x_l.

Simplex transformations $S' \to S$ in the NMS are controlled by four parameters: α for reflection, β for contraction, γ for expansion and δ for shrinkage (or massive contraction). They should satisfy the following constraints

$$\alpha > 0, \quad 0 < \beta < 1, \quad \gamma > 1, \quad \gamma > \alpha, \quad 0 < \delta < 1.$$

The standard values, used in most implementations, are

$$\alpha = 1, \quad \beta = \frac{1}{2}, \quad \gamma = 2, \quad \delta = \frac{1}{2}.$$

A slightly different choice has been suggested in [7]. There is also an alternate notation for these four parameters: ρ, γ, χ, σ, respectively, which is used in [3] and Matlab implementation fminsearch.m of the NMS [4].

Step 3 of TRANSF can be implemented in several different ways. Regarding efficiency, these differences are minor, so everything that will be subsequently said applies to all variants of the basic algorithm. The following algorithm for step 3 is based on [9].

```
  { Try to REFLECT the simplex }
x_r := c + α(c − x_h); f_r := f(x_r);
if f_r < f_s then { Accept REFLECT }
    x_h := x_r; f_h := f_r;
    if f_r < f_l then { Try to EXPAND }
        x_e := c + γ(x_r − c); f_e := f(x_e);
        if f_e < f_l then { Accept EXPAND }
            x_h := x_e; f_h := f_e;
else { We have f_r ≥ f_s. REFLECT if it helps, and try to CONTRACT }
    if f_r < f_h then
        x_c := c + β(x_r − c) { Outside contraction }
    else
        x_c := c + β(x_h − c); { Inside contraction }
    f_c := f(x_c);
    if f_c < min{f_r, f_h} then { Accept CONTRACT, inside or outside }
        x_h := x_c; f_h := f_c;
    else { SHRINK the simplex towards the best point }
        for j := 0 to n do
            if j ≠ l then
                x_j := x_l + δ(x_j − x_l); f_j := f(x_j);
```

Note that we accept expansion of the working simplex as soon as $f_r < f_l$ and $f_e < f_l$, regardless of the relationship between f_r and f_e. It may happen that $f_r < f_e$, so x_r would be a better new point than x_e, and we still include x_e in the new simplex. This is the so called "greedy expansion". We want to keep the simplex as large as possible, to avoid premature termination of the iterations, which is sometimes useful for non-smooth functions. On the other hand, the "greedy minimization" approach, used in [3], includes the better of the two points x_r, x_e in the new simplex, and the simplex is expanded only if $f_e < f_r < f_l$. It is hard to say which approach is better, as we have encountered examples where each one "beats" the other.

Any algorithm must terminate in a finite number of steps or iterations. For simplicity, assume that the algorithm TERM computes the logical (boolean) value *term* which becomes true when it is time to stop the iterations. Quite generally, *term* is composed of three different parts

$$term := term\_x \textbf{ or } term\_f \textbf{ or } fail;$$

where

- *term_x* is a "domain convergence or termination test", which becomes true when the working simplex S is sufficiently small in some sense (some or all vertices x_j are close enough),

- *term_f* is a "function value convergence test", which becomes true when (some or all) function values f_j are close enough in some sense,
- *fail* is a "no convergence in time" test.

There are many ways to define *term_x* and *term_f* tests and some examples will be given in the following sections. But, regardless of the exact definition of *term*, two simple facts should be observed.

The *fail* test must be present in any numerical algorithm. A method may be convergent in theory, but fail to do so in practice, due to many reasons, such as inexact computation.

Without a *term_x* test, the algorithm will obviously not work for discontinuous functions. But, such a test is necessary for continuous functions, as well, if we want to find a reasonable approximation for x^* in addition to the minimal function value. In such cases, *term_f* test is only a safeguard for "flat" functions.

3 Efficiency of a single iteration

To say anything about overall efficiency of the NMS, we would need some convergence theory to provide an estimate for the number of iterations required to satisfy any reasonable accuracy requirement in the termination test. Since no such theory exists, as yet, we confine ourselves to a more modest goal — efficiency analysis of a single NMS iteration. This has been done in [11] and reveals several potential computational bottlenecks. It also explains quite well why some implementations of the NMS (sometimes) run painfully slow.

For the sake of completeness, here we give a summary of results from [11], with emphasis on efficient implementation of various parts of TRANSF.

Let $T_{\mathrm{alg}}(\mathrm{input})$ denote the number of "flops" (machine floating point arithmetic operations) needed to execute the algorithm ALG for a given input.

In practice, the input function f is always given as an algorithm F which computes $f(x)$ for a given $x \in \mathbb{R}^n$. Therefore, it has its own complexity T_f which depends on the input point x. Of course, T_f implicitly depends on n — the flops required to compute $f(x)$ actually operate on coordinates $x(i)$, $i = 1, \ldots, n$, not on the whole x.

In order to obtain simple and useful complexity results, we have to assume that T_f is (more or less) **independent** of x, and (essentially) depends only on n, so that $T_f = T_f(n)$ holds for the complexity of F. This assumption is easily verified and certainly valid in many practical applications. In theory, it can also be viewed in the probabilistic sense, as the average complexity on x.

All this enables us to express complexity results in terms of n and $T_f(n)$. The standard asymptotic notation (o, O, Θ, Ω, ω) will be used just to hide the unnecessary details of the flops count.

Each complete iteration in the algorithm Simplex DSM consists of TERM and TRANSF, so the complexity $T_{\mathrm{iter}}(n)$ of a single iteration is

$$T_{\mathrm{iter}}(n) = T_{\mathrm{transf}}(n) + T_{\mathrm{term}}(n). \tag{2}$$

As TRANSF is the effective part of the inner loop, we want to spend as much time as possible in TRANSF doing useful work, without wasting too much time in TERM. This motivates the following definition of efficiency.

Definition 3.1 The efficiency of a single iteration of the Simplex DSM algorithm is

$$E_{\mathrm{iter}}(n) := \frac{T_{\mathrm{transf}}(n)}{T_{\mathrm{iter}}(n)} = 1 - \frac{T_{\mathrm{term}}(n)}{T_{\mathrm{iter}}(n)}. \tag{3}$$

The algorithm is efficient if $E_{\mathrm{iter}}(n) \approx 1$ holds for most of the iterations.

For the Nelder–Mead simplex transformation algorithm TRANSF we have the following complexity result [11].

Theorem 3.2 *Assume that every step of the NMS algorithm* TRANSF *is implemented as efficiently as possible. For all iterations, except the first one, the complexity of* TRANSF *is*

$$T_{\mathrm{transf}}(n) = \begin{cases} \Theta(n) + \Theta(T_f(n)), & \textit{without shrinkage in step } 3, \\ \Theta(n^2) + \Theta(nT_f(n)), & \textit{with shrinkage in step } 3. \end{cases} \tag{4}$$

P r o o f. The proof is constructive, as it shows how to implement all the steps of TRANSF efficiently, and (4) then follows easily.

We begin by considering the effects of the simplex transformation $S' \to S$ which performed in step 3. Without shrinkage, only one point in S', the worst one, is replaced by a new point. So, nonshrink steps are **fast** in the Nelder–Mead algorithm, which is one of the reasons why it is so popular in practice. To obtain an efficient implementation, we have to update all the objects (primarily c) by keeping all the information for unchanged points in $S' \to S$. To simplify the notation, the values of objects related to S' (or in the previous iteration) will be denoted by primes.

Let $T_k(n)$ denote the complexity of step k in TRANSF, for $k = 1, 2, 3$. Then

$$T_{\mathrm{transf}}(n) = T_1(n) + T_2(n) + T_3(n). \tag{5}$$

Efficient implementation of step 1 requires $O(n)$ (or, roughly, at most $3n$) comparisons to find the new indices h, s, l, even without updating. Therefore,

$$T_1(n) = O(n). \tag{6}$$

In some implementations (notably those for Matlab), the vertices of S are sorted to obtain a nondecreasing order of f_j in **each** iteration. This is certainly inefficient, as it gives $T_1(n) = O(n^2)$, or $T_1(n) = O(n \log n)$, at best.

On the other hand, if we want to apply the tie-breaking rules from [3] in case of equal function values, it is necessary to keep the vertices of S ordered and labeled $x_0, x_1, \ldots, x_n$, such that $f_0 \le f_1 \le \cdots \le f_n$ holds in each iteration. Then sorting is required only to obtain the initial ordering in S_{init}, or when S is a result of the shrink transformation, since only one point from S' (the best one in S') remains in S. In this case, $T_1 = O(n \log n)$ or $T_1(n) = O(n^2)$, depending on the sorting algorithm used.

If S has been obtained from S' by a nonshrink transformation in the previous step 3, we can do much better. Only one point has changed and the ordering can be updated in linear time (at most n comparisons) by one step of straight insertion sort. This again gives (6).

At first glance it seems that $\Theta(n^2)$ flops are required to compute the centroid c from (1) in step 2. This is true only for the initial centroid c_{init}, in the first iteration. Later on, the centroids can be efficiently updated, according to the simplex transformation which produced S in the previous iteration. It is easy to see that

$$c = \begin{cases} c' + \dfrac{1}{n}(x_{h'} - x_h), & \text{without shrinkage in } S' \to S, \\ x_{l'} + \delta(c' - x_{l'}) + \dfrac{1}{n}(x_{h'} - x_h), & \text{with shrinkage in } S' \to S. \end{cases} \tag{7}$$

Note that all points refer to the current simplex S. Only the indices h' and l' have to be saved, but not the coordinates of the corresponding points in S', so almost no additional storage is required. As it can happen that $h = h'$, from (7) it is obvious that

$$T_2(n) = O(n), \tag{8}$$

except for the first iteration, when $T_2(n) = \Theta(n^2)$. It should be noted that centroid updates are not really necessary after a shrink transformation (which itself takes much more time), but they still pay off if there is a significant number of shrinkages throughout the iterations. On the other hand, all other centroid updates are essential for efficiency, as will be illustrated in the last section.

Most of the work in step 3 is spent to compute a certain number of new points, including the function value for each point. Each new point x requires $\Theta(n)$ flops to compute and $T_f(n)$ flops for $f(x)$, or the complexity per point is $\Theta(n) + T_f(n)$. At least one new point x_r is always computed. Without shrinkage, at most one additional point x_e or x_c is computed. Finally, if shrinkage is used, $n + 2$ points are computed. Thus

$$T_3(n) = \begin{cases} \Theta(n) + \Theta(T_f(n)), & \text{without shrinkage}, \\ \Theta(n^2) + \Theta(nT_f(n)), & \text{with shrinkage}. \end{cases} \tag{9}$$

Substitution of (6), (8) and (9) in (5) completes the proof. □

It is reasonable to assume that $f(x)$ depends, in general, on all n coordinates of x. When we compute $f(x)$ for a given x, each coordinate $x(i)$ is used at least once as an operand in a flop, and, since flops take at most two operands, at least $n/2$ flops are needed to compute $f(x)$. Thus, $T_f(n)$ is at least **linear** in n, or $T_f(n) = \Omega(n)$, so the second term dominates in (4).

Experience also shows that slow shrinkage transformations are quite rare in practice, so the first relation in (4)

$$T_{\mathrm{transf}}(n) = \Theta(T_f(n)) \tag{10}$$

can be used to judge the efficiency, as it is true for most of the iterations. From (2) and (3) it follows that

$$E_{\mathrm{iter}}(n) = 1 - \frac{T_{\mathrm{term}}(n)}{T_{\mathrm{term}}(n) + \Theta(T_f(n))},$$

so the efficiency of a single NMS iteration crucially depends on $T_{\mathrm{term}}(n)$, or how fast the termination test is with respect to the function evaluation. If $T_{\mathrm{term}}(n) = \omega(T_f(n))$ for a given function f, the termination test becomes a computational bottleneck, and the NMS algorithm is inefficient for that f.

To avoid this situation for all functions f, the complexity of TERM must satisfy

$$T_{\mathrm{term}}(n) = o(n), \tag{11}$$

In other words, the termination test has to be **sublinear** in n. Note that a linear termination test $T_{\mathrm{term}}(n) = \Theta(n)$ is certainly efficient for all functions f such that $T_f(n) = \omega(n)$. But, if $T_f(n) = \Theta(n)$ with a small hidden constant in Θ, then $E_{\mathrm{iter}}(n) = O(1)$ and the overall efficiency may be low.

We see that (11) is a necessary condition for efficiency. But, if everything else is implemented efficiently, it is also sufficient — it guarantees that we have an implementation of the Nelder–Mead algorithm which is efficient for all functions f.

It should be pointed out that this type of inefficiency is quite specific for the Nelder–Mead method. When we look back, it comes as a consequence of (10). The same argument does not apply to many other DSM algorithms which have more than a constant number of function evaluations per iteration (usually n). It is just because the NMS iterations are so fast, that the termination test becomes a problem. Of course, the most interesting question is whether it occurs in practice.

3.1 Efficiency of termination tests

To estimate $T_{\mathrm{term}}(n)$, we have to analyze all three parts of the termination test.

The *fail* test just checks the number of iterations or function evaluations against the prescribed maximum allowed value. Its complexity is $\Theta(1)$, and it is always efficient.

There are essentially two different types of *term_f* tests in practice. The first type uses a constant number of function values (usually 2 or 3) to compute the test, and the complexity is $\Theta(1)$, which is again efficient. A typical example, taken from amoeba in [8], is

$$term\_f := 2 \cdot \frac{|f_h - f_l|}{|f_h| + |f_l|} \le tol\_f, \tag{12}$$

where tol_f is some prescribed relative tolerance. We have found this test to be quite effective in practice.

On the other hand, fminsearch.m in Matlab [4] uses

$$term\_f := f_h - f_l \le tol\_f, \tag{13}$$

where tol_f is now a given absolute tolerance.

The second type of tests uses all $n + 1$ function values, so the complexity is $\Theta(n)$, which may be inefficient. Note that (13) in fminsearch.m is actually computed as

$$term\_f := \max_{j \neq l} |f_j - f_l| \le tol\_f,$$

so it is really of the second type.

The *term_x* test is the real bottleneck in practice. All implementations of the Nelder–Mead method that we have checked can be divided into three groups with respect to the *term_x* test.

1. This test requires $\Theta(n^2)$ flops, which is slow (sometimes very slow). All tests based on the diameter of the working simplex fall into this group. For example, fminsearch.m in Matlab [4] uses

$$term\_x := \max_{j \neq l} \|x_j - x_l\|_\infty \leq tol\_x,$$

where tol_x is a given absolute tolerance, while Higham's nmsmax.m [1, 2] uses

$$term\_x := \frac{\max_{j \neq l} \|x_j - x_l\|_1}{\max\{1, \|x_l\|_1\}} \leq tol\_x, \tag{14}$$

where tol_x is now a given relative tolerance.

2. This test requires $\Theta(n)$ flops, which may be inefficient when $T_f(n) = \Theta(n)$. We have found only one example of such a test — by Rowan in simplx.f from [9]:

$$term\_x := \|x_h - x_l\|_2 \leq tol\_x \cdot \|x_h^{(0)} - x_l^{(0)}\|_2, \tag{15}$$

where $x_l^{(0)}$, $x_h^{(0)}$ denote the best and the worst point in the initial simplex S_{init}, and tol_x is again a given relative tolerance.

3. Such a test is not present at all. This is, obviously, efficient, but works only for (at least) continuous functions f. For example, one of the most popular implementations amoeba from [8] belongs to this group.

We can conclude that none of these implementations is efficient for all functions f.

4 Efficient domain convergence test

In the view of all we have said so far, it is quite surprising that an efficient domain convergence test, which satisfies (11), is not so hard to construct. In fact, it is almost obvious, from geometrical point of view, and performs even better than required by (11).

The NMS algorithm performs a series of simplex transformations and it is easy to see that simplex volumes behave in a simple way under these transformations. The volume of the simplex $S = S(x_0, \ldots, x_n)$ is defined as

$$V(S) := \frac{1}{n!} \cdot \sqrt{\Gamma(x_1 - x_0, \ldots, x_n - x_0)},$$

where Γ denotes the Gramm determinant.

Each of the 5 elementary simplex transformations in step 3 of the NMS can be written as $S := \text{transform}(S')$, where $\text{transform} \in \{\text{reflect}, \text{expand}, \text{inside contract}, \text{outside contract}, \text{shrink}\}$. It is easy to see that the following holds for the volume ratio

$$\frac{V(S)}{V(S')} = \begin{cases} \alpha, & \text{if transform} = \text{reflect}, \\ \beta, & \text{if transform} = \text{inside contract}, \\ \alpha \cdot \beta, & \text{if transform} = \text{outside contract}, \\ \alpha \cdot \gamma, & \text{if transform} = \text{expand}, \\ \delta^n, & \text{if transform} = \text{shrink}. \end{cases} \tag{16}$$

This means that the volume ratio can be updated in a **single** flop in each iteration, if we precompute the factors in (16). The obvious "relative volume" termination test is

$$term\_v := V(S) \leq tol\_v \cdot V(S_{\text{init}}), \tag{17}$$

where *tol_v* is the prescribed (relative) volume tolerance.

To avoid underflow and overflow problems in floating–point computation of (17) for higher values of n, we can rewrite it in a "linearized" form

$$term\_x := LV(S) \leq tol\_x \cdot LV(S_{\text{init}}), \tag{18}$$

where $LV(S) := \sqrt[n]{V(S)}$ denotes a "linearized volume" of S, and *tol_x* is the same prescribed relative tolerance as in the usual *term_x* tests.

Linearized volume ratio updates are trivial from (16), so (18) also requires a **single** flop per iteration, which is always efficient.

Finally, the initial linearized volume $LV(S_{\text{init}})$ can be easily computed for all standard choices of S_{init}. For example, if we use a right–angled simplex with vertices $x_j := x_0 + h_j e_j$, $j = 1, \ldots, n$, it is obvious that $LV(S_{\text{init}}) = \sqrt[n]{h_1 \cdots h_n}$.

Actually, there is no need at all to compute $LV(S_{\text{init}})$, as we are only interested in (linearized) volume ratios. We can easily set $LV(S_{\text{init}}) = 1$, regardless of how S_{init} is formed, and apply linearized volume ratio updates until (18) is satisfied.

Some theoretical justification for this test can be found in [3], where $V(S) \to 0$ is used in convergence proofs. Of course, in practice, we can encounter situations when $V(S) \to 0$, and S simply degenerates into a hyperplane, without being small (in terms of its diameter). But, this should not be interpreted as a disadvantage of this test. On the contrary, if this happens, it usually means that there is something wrong with our problem. Either, n is too high, and we have too much freedom (as in nonlinear least squares problems), or the method has simply stuck, and it is time to restart (or accept the results).

As we shall see in the following section, the linearized "relative volume" test performs quite well in practice. Similar termination tests can also be applied in some other simplex–based direct search methods.

5 Numerical experiments

The main goal of this section is to show what gains can be achieved by efficient implementation of various parts of the NMS algorithm.

Our model problem, as in [10], is to maximize the growth factor in Gaussian elimination for various pivoting strategies (also, see Chapter 26 in [2]).

Let $m \in \mathbb{N}$ and let $A \in \mathbb{R}^{m \times m}$ be a real matrix of order m. Consider the LU factorization or the Gaussian elimination of A with a prescribed pivoting strategy P. Numerical stability of this process can be expressed in terms of the pivotal growth factor [2]. The "worst" cases for P are described by maximal values of

$$f(A) = \text{growth factor } \rho_m(A) \text{ in the LU–P factorization of } A, \quad A \in \mathbb{R}^{m \times m}.$$

The growth factor $\rho_m(A)$ for a particular strategy is defined by

$$\rho_m(A) = \frac{\max_{i,j,k} |a_{ij}^{(k)}|}{\max_{i,j} |a_{ij}|},$$

where $a_{ij}^{(k)}$ are the intermediate elements generated during the elimination. Note that f is discontinuous at some points in $\mathbb{R}^n$, with $n = m^2$. For all reasonable pivoting strategies, T_f only slightly depends on A, and we have

$$T_f(m) = \Theta(m^3) \quad \text{or} \quad T_f(n) = \Theta(n^{3/2}).$$

For simplicity, we first take the pivoting strategy P to be the well known partial pivoting. It is known that the maximal growth factor is 2^{m-1}, but this is not very important for our experiment. We want to demonstrate how various implementations of step 1, step 2, and TERM affect the overall efficiency. To this aim, we test two different implementations of step 1

- S1 (slow), which uses sorting to find the indices h, s, l, with $T_1(n) = O(n^2) = O(m^4)$,
- F1 (fast), which finds h, s, l directly by comparisons, with $T_1(n) = O(n) = O(m^2)$.

No attempt is made to observe the tie-breaking rules from [3] in case of equal function values.

Likewise, to compute c in step 2, we have two choices: (1) and (7). The first one is slow, with $T_2(n) = \Theta(n^2) = \Theta(m^4)$, and the second is fast, $T_2(n) = O(n) = O(m^2)$.

Finally, in TERM we always use the function value test (12), and compare three domain convergence tests with drastically different complexities: Higham's (14), Rowan's (15), and the linearized "relative volume" test (18).

Since $T_f(m) = \Theta(m^3)$, a linear *term_x* test, like Rowan's, is quite fast, as it takes $\Theta(n) = \Theta(m^2)$ time. In fact, we do not expect to see much difference in performance between (15) and (18). On the other hand, a quadratic test, like Higham's, takes $\Theta(n^2) = \Theta(m^4)$ time, which is out of question, as the evaluation of f is slow enough by itself.

By combining some of the above choices, we obtain the following 5 algorithms with different implementations of the NMS:

A1 = S1 + Slow centroid (1) + Higham's *term_x* (14),
A2 = F1 + Slow centroid (1) + Higham's *term_x* (14),
A3 = F1 + Fast centroid (7) + Higham's *term_x* (14),
A4 = F1 + Fast centroid (7) + Rowan's *term_x* (15),
A5 = F1 + Fast centroid (7) + Volume *term_x* (18).

These algorithms are tested and timed on exactly the same set of jobs. For each value of $m = 2, \ldots, 8$, we use m random starting matrices and maximize the pivotal growth ρ_m. Each of these m jobs consists of 5 runs (start and 4 restarts), with a maximum of 4000 function evaluations per run. Finally, all termination tests use the same tolerances $tol\_x = 10^{-8}$, $tol\_f = 10^{-10}$, since all computations are performed in IEEE extended precision with unit roundoff $u = 5.42 \cdot 10^{-20}$.

To compare efficiencies, for each value of m, we compute two simple efficiency measures — the average time per iteration, denoted by $e_{\mathrm{iter}}(m)$, and the average time per function evaluation, denoted by $e_{\mathrm{feval}}(m)$. In both cases, the average is taken over all runs and all jobs for that m. Our timing includes the whole iteration loop, i.e., both TERM and TRANSF, together with the initial centroid c_{init} calculation, which is slow. The computed values of $e_{\mathrm{iter}}(m)$ and $e_{\mathrm{feval}}(m)$ for all 5 algorithms are given in Table 1.

Table 1 Average times (in 10^{-6} s) per iteration and per function evaluation for various algorithms.

	A1		A2		A3		A4		A5	
m	e_{iter}	e_{feval}	e_{iter}	e_{feval}	e_{iter}	e_{feval}	e_{iter}	e_{feval}	e_{iter}	e_{feval}
2	162	73	146	79	138	74	124	67	114	61
3	363	212	358	218	312	190	230	139	212	125
4	827	506	823	513	668	423	393	247	382	218
5	1684	1121	1669	1130	1297	875	603	406	552	364
6	3163	2231	3132	2193	2350	1646	885	618	798	555
7	5535	4031	5460	3972	3978	2894	1228	893	1103	801
8	9112	6568	8986	6336	6427	4532	1717	1213	1545	1089

These results clearly show how much is gained by efficient implementation of various steps in the NMS.

We must say that all the values in Table 1 are intentionally much higher than they should really be in practice. They were obtained by running an experimental code, with quite a lot of book-keeping, on a rather slow computer (Pentium II, 333 MHz), especially chosen to get more accurate timings.

To get some feeling of the amount of computation involved in this test, Table 2 shows the numbers of function evaluations and iterations (both total and by type of transformation) performed by each algorithm to produce the results in the last row $m = 8$ (or $n = 64$) of Table 1. The results for A3 and A5 are equal by coincidence.

Table 2 Number of function evaluations and iterations for $m = 8$ in various algorithms.

number of	A1	A2	A3	A4	A5
function evaluations	157407	156107	156091	155359	156091
iterations	112201	110073	110057	109759	110057
reflections	81985	80541	80525	80938	80525
expansions	1410	1489	1489	1504	1489
contractions	28609	27817	27817	27087	27817
shrinkages	197	226	226	230	226

Because of high running times, we have used only m random starts, which is far too small to find high pivotal growth factors. The maximal values found in this experiment are not worth mentioning, but it should be noted that all algorithms ended with similar values.

The first experiment was designed just to compare efficiencies of algorithms A1–A5. In the following one, we are going to use only A5, but in a more serious attempt to find high growth factors for partial, complete, and rook pivoting strategies (see [2]). This time, we use $2m^3$ random starting matrices, allow 10 runs (9 restarts) for each job, with a stronger requirement $tol\_x = 10^{-12}$ in (18). All other parameters are the same as in the first experiment. Actually, the source code is the same as before.

The whole run takes about 4 hours on a much faster machine (Pentium 4, $3\,\mathrm{GHz}$), and the obtained maximal growth factors are given in Table 3.

Table 3 Obtained growth factors by algorithm A5 for partial, complete and rook pivoting.

	Pivoting strategy		
m	partial	complete	rook
2	1.99999999991650406	1.99999999989327416	1.99999999992127092
3	3.99999999984057538	1.99999999992693958	2.99997357960583527
4	7.99611697434938424	3.97730448158971522	4.45411806209411044
5	15.1876649879346456	3.99978652257938052	5.80481033824281461
6	24.3141518961117570	3.99985831323532101	6.59020480078501313
7	39.2122482678208174	4.16061043683397093	6.76462611299813698
8	59.7370345076401456	4.24812396323047320	7.57048152020520245
9	71.9623989093906266	4.29664555613955791	8.88565039452449410
10	106.394726804636864	4.35430403736946545	9.18275254080742746

This is still not a systematic search for maximal (or high) growth factors, as we have used only purely random starting matrices for each order m. No attempt has been made to use "bad" matrices for smaller m's as starts for higher m's. Actually, this experiment only shows that the probability of finding a really "bad" matrix (with high growth factor) is much smaller than $1/m$, $1/m^2$, or even $1/m^3$, as m grows. This fact is well known, and has already been demonstrated by similar experiments in [12], which, like [2], is an excellent reference on the subject of pivoting.

The results of these experiments are not very significant by themselves. We may say that the results for rook pivoting in Table 3 are better lower bounds then those in Problem 9.18 in [2]. But this is only a byproduct of these tests, showing that something can be gained by more extensive experimenting, which is possible by using more efficient tools.

6 Conclusion

The relative "linearized volume" test proposed in this work makes the NMS run much faster, and its performance is comparable to the standard domain convergence termination tests, regarding the obtained optimal function values. This may not be so important for simple and easy problems, but for some difficult and time consuming problems which occur quite frequently in experimental mathematics, the advantages may be enormous.

For years, the Nelder–Mead method has been an invaluable tool in numerical experiments, and making it more efficient simply gives us the opportunity to tackle more complicated and larger problems.

Acknowledgements This work was supported by grant 0037114 by Ministry of Science, Education and Sports, Croatia.

References

[1] N. J. Higham, The Test Matrix Toolbox for Matlab (version 3.0), Numerical Analysis Report 276, Manchester Centre for Computational Mathematics, Manchester, England (1995).
[2] N. J. Higham, Accuracy and Stability of Numerical Algorithms, second ed. (SIAM, Philadelphia, 2002).
[3] J. C. Lagarias, J. A. Reeds, M. H. Wright and P. E. Wright, Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions, SIAM. J. Optim. **9**, 112–147 (1998).
[4] The MathWorks, Inc., MATLAB Language Reference Manual, Version 7 (Natick, Massachusetts, 2004).
[5] K. I. M. McKinnon, Convergence of the Nelder–Mead Simplex Method to a Nonstationary Point, SIAM. J. Optim. **9**, 148–158 (1998).
[6] J. A. Nelder and R. Mead, A simplex method for function minimization, Comput. J. **7**, 308–313 (1965).
[7] J. M. Parkinson and D. Hutchinson, An investigation into the efficiency of variants on the simplex method, in: Numerical Methods for Nonlinear Optimization, edited by F. A. Lootsma (Academic Press, New York, 1972), pp. 115–135.
[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes in FORTRAN: The Art of Scientific Computing, second ed. (Cambridge University Press, Cambridge, New York, 1992).
[9] T. H. Rowan, Functional Stability Analysis of Numerical Algorithms, Ph.D. thesis, University of Texas, Austin, (1990).
[10] S. Singer and S. Singer, Some applications of direct search methods, in: Proceedings of the 7^{th} International Conference on Operational Research — KOI'98, Rovinj, Croatia, 1998, edited by I. Aganović, T. Hunjak, and R. Scitovski (Croatian Operational Research Society, Osijek, 1999), pp. 169–176.
[11] S. Singer and S. Singer, Complexity Analysis of Nelder–Mead Search Iterations, in: Proceedings of the 1. Conference on Applied Mathematics and Computation, Dubrovnik, Croatia, 1999, edited by M. Rogina, V. Hari, N. Limić and Z. Tutek (PMF–Matematički odjel, Zagreb, 2001), pp. 185–196.
[12] L. N. Trefethen and D. Bau III, Numerical Linear Algebra (SIAM, Philadelphia, 1997).