Proceedings of the 1. Conference on
Applied Mathematics and Computation
Dubrovnik, Croatia, September 13–18, 1999
pp. 185–196

Complexity Analysis of Nelder–Mead Search Iterations*

Sanja Singer† and Saša Singer‡

Abstract. We investigate potential computational bottlenecks in the Nelder–Mead search (NMS) algorithm. Many implementations of the algorithm exist, differing mainly in the selection of convergence or termination criteria and some minor computational details. Therefore, a fairly general complexity analysis of a single NMS iteration step is presented, which leads to a simple efficiency measure. Efficiency analysis of the most common NMS implementations clearly identifies a computational bottleneck in domain convergence criterion. This seriously affects the performance of the method, at least for discontinuous functions.

AMS subject classification: 65K10, 65Y20, 90C56

Key words: optimization, direct search methods, Nelder–Mead simplex algorithm, complexity, termination criteria

1. Introduction

The classical unconstrained optimization problem is to locate a point of minimum (or maximum) x^* of a given function $f : \mathbb{R}^n \to \mathbb{R}$. There is a wide variety of methods at our disposal for the solution of this problem, depending on how much information about f is available.

In some applications, regardless of the particular choice of f, we can expect in advance that

- f is continuous, but not smooth, or
- f is not even continuous,

at least at some points in $\mathbb{R}^n$. This information imposes a fundamental restriction. To find x^*, the optimization method should use only the function values of f, without using or estimating the derivatives of f, as they may not exist at a particular point. Methods of this type are usually called Direct Search Methods (DSM).

*This work was supported by the Grant No. 037011 from the Ministry of Science and Technology of the Republic of Croatia.

†Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, I. Lučića 5, 10000 Zagreb, Croatia, e–mail: ssinger@math.hr

‡Department of Mathematics, University of Zagreb, Bijenička cesta 30, 10000 Zagreb, Croatia, e–mail: singer@math.hr

Nelder–Mead search (NMS) or simplex search is one of the best known and most widely used methods in this class. Originally published in 1965. [5], the method is nowadays a standard member of all major libraries and has many different implementations.

There is a widespread belief, based mainly on extensive numerical evidence, that NMS becomes inefficient as n increases, even for moderate space dimensions (like $n \geq 10$). This believed inefficiency of the NMS is hard to substantiate, at least mathematically, due to the lack of convergence theory. But, in a recent work [10] we have compared several DSMs on series of problems with discontinuous functions f. Contrary to all our expectations, NMS turned out to be (by far) the slowest method!

This result clearly indicated that there must be a computational bottleneck in our implementation of the NMS. Furthermore, the incriminated block of code that degrades the NMS, is either not present in the other methods, or it does not affect their performance. So we started to search for the bottleneck.

To present the results, we first give a brief description of the NMS algorithm in the next section, followed by the general complexity analysis of a single iteration. As a result, if everything else is implemented efficiently, we can identify the bottleneck in the domain convergence (or termination) criterion. And for discontinuous functions such a criterion becomes a necessity.

Efficiency analysis of the most common NMS implementations reveals that none of them is immune for discontinuous functions and some of them are unable to handle such functions at all. Finally, we illustrate the efficiency by a few typical examples.

2. Nelder–Mead search algorithm

The algorithm will be presented in "minimize" form, as in all standard implementations. We begin by specifying the "endpoints" of the algorithm. Any DSM minimizer requires the following input data

$f : \mathbb{R}^n \to \mathbb{R}$ — function to be minimized,

and returns

$x_{\text{final}} \in \mathbb{R}^n$ — computed approximation for x^*.

In practice, we can always specify an additional input

$x_{\text{init}} \in \mathbb{R}^n$ — initial (starting) point,

to be used as an initial approximation for x^*. This makes it easy to restart the algorithm by using the previously computed x_{final} as an input for the next run.

A simplex $S \subset \mathbb{R}^n$ is determined by $n+1$ points or vertices $x_0, \ldots, x_n \in \mathbb{R}^n$ and will be denoted as $S = S(x_0, \ldots, x_n)$. Recall that S is defined as the convex hull of the vertices $x_0, \ldots, x_n$. Simplex based DSM algorithms (including NMS) perform certain transformations of the working simplex S, based on the function values $f_j := f(x_j)$, for $j = 0, \ldots, n$. The general algorithm is

Algorithm Simplex DSM

INIT: construct an initial working simplex S_{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;
$x_{\text{final}} :=$ 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 $S_{\text{init}} = S(x_0, \ldots, x_n)$ around (or near) the initial point x_{init}, and computes the function values at all the vertices

$$f_j := f(x_j), \quad j = 0, \ldots, n.$$

The most frequent choice is $x_0 = x_{\text{init}}$ to allow proper restarts. Usually, S_{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_{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. There are many implementations of the NMS and, surprisingly, almost as many implementations (or variants) of TRANSF. The one presented here is based on [9]. It 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)
$$c := \frac{1}{n} \sum_{\substack{j=0 \\ j \neq h}}^{n} x_j.$$
3. Compute the new working simplex S from the old one. First, try to replace the worst point x_h with a better point x_{new}, by using reflection, expansion or contraction. If this fails, shrink the simplex towards the best point x_l.

Simplex transformations in the NMS are controlled by 4 parameters (or coefficients): α for reflection, β for contraction, γ for expansion and δ for shrinkage (or massive contraction). They should satisfy the following constraints

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

The following algorithm then shows the details of step 3.

```
  { 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) else x_c := c + β(x_h − c);
    f_c := f(x_c);
    if f_c < min{f_r, f_h} then { Accept CONTRACT }
        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);
```

The simplex transformation coefficients have the following standard values

$$\alpha = 1, \quad \beta = 0.5, \quad \gamma = 2, \quad \delta = 0.5$$

and they are used in most implementations. A slightly different choice has been suggested in [7].

Algorithm Simplex DSM 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 finish — stop the iterations. Quite generally, *term* is composed of 3 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. Even if there is a convergence theory, the method may fail to converge 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, if we want to find a reasonable approximation for x^*, a *term_x* test is necessary for continuous functions, as well. In such cases, *term_f* test is only a safeguard for "flat" functions.

3. Complexity analysis of a single iteration

A truly general complexity analysis of the NMS algorithm is impossible at the present time. There is no convergence theory to support such an analysis by providing an estimate for the number of iterations required to satisfy any reasonable accuracy constraint given in the termination test.

Therefore, we limit ourselves to the complexity analysis of a single NMS iteration. Besides, this is enough to identify the bottleneck we are searching for.

Definition 1. *The complexity of algorithm* ALG *is the number of "flops" — machine floating point arithmetic operations needed to execute the algorithm for a given input. This number will be denoted as* $T_{\mathrm{alg}}(\mathrm{input})$.

This definition of complexity is the best suited one for numerical algorithms, where most of the time is spent in floating point calculations. Before we can use it, we have to clarify what will be counted as a flop. The answer is simple — everything that takes at most constant time to compute, for all representable floating point numbers. This includes the four standard arithmetic operations, as well as a number of functions (such as $\sqrt{}$, exp, log). A comparison of floating point numbers will also be counted as a flop, since it is performed via subtraction and comparison to zero.

By definition, all complexity results should be expressed in terms of input data, and for any DSM algorithm, this is f. But, in practice, 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 to be expressed in terms of the input point x. Thus, we can use $T_f = T_f(x)$, instead of f, to express complexity. Of course, T_f implicitly depends on n — the flops required to compute $f(x)$ operate on coordinates $x(i)$, $i = 1, \ldots, n$, not on the whole x.

To study the effects of the space dimension n on complexity, we have to make n the parameter of the problem. This effectively means that we have a sequence of functions $f_n : \mathbb{R}^n \to \mathbb{R}$, $n \in \mathbb{N}$, each of them to be minimized. In practice, we can simply assume that the algorithm F has an additional input parameter n, and computes the value $f_n(x)$. The complexity $T_f = T_f(x, n)$ now explicitly depends on n. The particular function minimized by DSM will continue to be denoted by f, to simplify the notation, as n will be obvious from the domain of f.

To simplify the presentation, the standard asymptotic notation (o, O, Θ, Ω, ω) will be used, but in a somewhat extended sense, customary in complexity analysis of algorithms. The flops count will be correct for all functions f, all points x and all space dimensions $n \in \mathbb{N}$. Asymptotic notation will be used just to hide the unnecessary details of this count, but the result will remain valid for all arguments, no restriction to sufficiently large arguments being imposed.

With this notation, our definition of complexity can be simply related to the usual time complexity. For sequential execution of numerical algorithms we have

$$\text{Execution time of } \mathsf{ALG}(\text{input}) = \Theta(T_{\text{alg}}(\text{input})).$$

In practice, these two quantities are often almost proportional.

Note that any DSM algorithm uses F many times to compute $f(x)$ for various points x throughout the iterations. In order to obtain simple and useful complexity results, we have to make one more important assumption — eliminate (or neglect) the dependence of $T_f(x, n)$ on x.

Assumption 1. *The complexity of computing the function value $f(x)$, for a given point x, is (more or less)* **independent** *of x, and (essentially) depends only on n.*

More precisely, we assume that there exist $c_1, c_2 \in \mathbb{R}$, with $0 < c_1 \leq 1 \leq c_2$, and a function $T_f : \mathbb{N} \to \mathbb{N}$, such that

$$c_1 T_f(n) \leq T_f(x, n) \leq c_2 T_f(n), \quad \forall x \in \mathbb{R}^n, \quad \forall n \in \mathbb{N}.$$

This assumption is valid in many practical applications and a few examples will be given later on. The same type of assumption can be stated in the probabilistic sense, as the average complexity on x. From now on, we assume that $T_f = T_f(n)$ holds for the complexity of the algorithm F, but it is easy to see that all the results remain valid in the general case $c_1 \neq c_2$.

As a consequence, our complexity results will be stated in terms of n and $T_f(n)$. Before proceeding, we have to ensure that n is not artificially large.

Definition 2. *Function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be sensibly defined if all n coordinates of x influence the function value $f(x)$.*

If this is not so, f can be sensibly redefined with a smaller n. So, we assume that f is already sensibly defined on input.

The following trivial lower bound will be useful later.

Lemma 1. *For any sensibly defined function $f : \mathbb{R}^n \to \mathbb{R}$, the complexity of function value computation satisfies*

$$T_f(n) = \Omega(n),$$

or $T_f(n)$ is at least **linear** *in n.*

Proof. By assumption, $f(x)$ depends on all n coordinates of x. Each coordinate $x(i)$ has to be used at least once as an operand in a flop. Since flops take at most 2 operands, at least $n/2$ flops are needed to compute $f(x)$. ■

Having done all the preliminary hard work, the rest of the analysis is quite easy.

Let $T_{\text{iter}}(n)$ denote the complexity of a single iteration (inner loop) of the algorithm Simplex DSM. Each complete iteration consists of two successive steps: TERM and TRANSF. Therefore, the complexity of a single iteration is

$$T_{\text{iter}}(n) = T_{\text{transf}}(n) + T_{\text{term}}(n). \tag{1}$$

As TRANSF is the effective part of the inner loop, we want to spend as much time as possible in TRANSF doing useful work, and not wasting too much time in TERM. This motivates the following definition of efficiency.

Definition 3. *The efficiency of a single iteration of the* Simplex DSM *algorithm is*

$$E_{\text{iter}}(n) := \frac{T_{\text{transf}}(n)}{T_{\text{iter}}(n)}.$$

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

For the NMS simplex transformation algorithm TRANSF we have the following complexity result.

Theorem 1. *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*

$$T_{\text{transf}}(n) = \begin{cases} \Theta(n) + \Theta(T_f(n)), & \text{if no "shrink" is used,} \\ \Theta(n^2) + \Theta(nT_f(n)), & \text{if "shrink" is used} \end{cases} \tag{2}$$

in step 3 *of* TRANSF.

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

$$T_{\text{transf}}(n) = T_1(n) + T_2(n) + T_3(n). \tag{3}$$

Efficient implementation of step 1 requires $O(n)$ (or, roughly, at most $3n$) comparisons to find the new indices h, s, l. Sorting is neither required, nor necessary, so we have

$$T_1(n) = O(n). \tag{4}$$

At first glance it seems that $\Theta(n^2)$ flops are required to compute the centroid c in step 2

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

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 in the previous iteration. Let primes denote the values of objects in the previous iteration. It is easy to see that

$$c = \begin{cases} c' + \dfrac{1}{n}(x_{h'} - x_h), & \text{if no "shrink",} \\ x_{l'} + \delta(c' - x_{l'}) + \dfrac{1}{n}(x_{h'} - x_h), & \text{if "shrink"} \end{cases} \tag{5}$$

has been used in the previous step 3 to obtain S from S'. Note that all points refer to the current simplex S. Only indices h' and l' have to be saved, and no additional storage is required. As it can happen that $h = h'$, from (5) it is obvious that

$$T_2(n) = O(n), \tag{6}$$

except for the first iteration, when $T_2(n) = \Theta(n^2)$.

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

$$T_3(n) = \begin{cases} \Theta(n) + \Theta(T_f(n)), & \text{if no "shrink" is used,} \\ \Theta(n^2) + \Theta(nT_f(n)), & \text{if "shrink" is used.} \end{cases} \tag{7}$$

Substitution of (4), (6) and (7) in (3) completes the proof. ■

Note that $T_f(n) = \Omega(n)$ from Lemma 1, so we can eliminate the first term $\Theta(n)$ or $\Theta(n^2)$ in (2) and (7).

The efficiency $E_{\text{iter}}(n)$ of a single NMS iteration follows directly from (1) and (2). Of course, it crucially depends on $T_{\text{term}}(n)$. But, if TRANSF is implemented efficiently we can immediately recognize and pinpoint the possible bottleneck in TERM. There simply is no other place where it can be. But is it a real danger? And, if so, how to avoid it?

The following argument relies heavily on the fact that shrinkages are extremely rare in practice. Therefore, the first relation in (2) can be used to judge the efficiency, as it is true for most of the iterations. The same results can be obtained by a probabilistic argument, as soon as $P(\text{shrink}) = O(1/n)$ holds for the probability of "shrink" iterations. This is certainly true in practice, but such a fact would be extremely difficult to **prove**.

Theorem 2. *Assume that the NMS algorithm* TRANSF *is implemented as efficiently as possible and that shrinkages are rare. If, for a given function f,*

$$T_{\text{term}}(n) = \omega(T_f(n)), \tag{8}$$

then the NMS algorithm is inefficient for f, or the termination test TERM *will be a bottleneck. To avoid it for all functions, the complexity of* TERM *must satisfy*

$$T_{\text{term}}(n) = o(n), \tag{9}$$

or the NMS algorithm is efficient for all functions f.

Proof. If there are no shrinkages in TRANSF, from (1), (2) and Lemma 1, we have

$$T_{\text{iter}}(n) = \Theta(T_f(n)) + T_{\text{term}}(n),$$

or the efficiency is

$$E_{\text{iter}}(n) := \frac{T_{\text{transf}}(n)}{T_{\text{iter}}(n)} = \frac{\Theta(T_f(n))}{\Theta(T_f(n)) + T_{\text{term}}(n)}.$$

If (8) holds, it follows that $E_{\text{iter}}(n) = o(1)$, or the efficiency becomes negligible. On the other hand, if (9) is true, we have

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

and $E_{\text{iter}}(n) = 1 - o(1)$, so the algorithm is efficient for all f. ■

It is interesting to take a look at the limiting cases of (8) and (9). If

$$T_{\text{term}}(n) = \Theta(T_f(n)) \tag{10}$$

then $E_{\text{iter}}(n) = O(1)$, but is certainly less than 1. As the hidden constant in Θ in (2) is small (at most 2 function evaluations), we cannot expect high efficiency even if the hidden constant in (10) is small.

Now suppose that we have a linear TERM test

$$T_{\text{term}}(n) = \Theta(n). \tag{11}$$

This will certainly be 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 again $E_{\text{iter}}(n) = O(1)$ and the overall efficiency will be low.

4. Efficiency of some implementations

Having identified the possible computational bottleneck, it is an interesting question whether it occurs in some of the most common implementations of the NMS. Let *tol_x* and *tol_f* be the prescribed tolerances for *term_x* and *term_f* tests, respectively. For each implementation we state the definition of tests used in that implementation. Unused tests should be interpreted as *true* in our notation.

IMSL — subroutines UMPOL/DUMPOL [11]:

$$term\_f := f_h - f_l \le tol\_f \cdot (1 + |f_l|) \quad \textbf{or} \quad \sum_{j=0}^{n} (f_j - f_{\mathrm{mean}})^2 \le tol\_f,$$

where $f_{\mathrm{mean}} := \dfrac{1}{n+1} \sum_{j=0}^{n} f_j$.

NAG — subroutine E04CCF [6]:

$$term\_f := \sqrt{\frac{1}{n+1} \sum_{j=0}^{n} (f_j - f_{\mathrm{mean}})^2} \le tol\_f, \quad \text{with } f_{\mathrm{mean}} \text{ as above.}$$

Numerical Recipes — subroutine amoeba [8]:

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

Compact Numerical Methods — Algorithm 19, procedure nmmin [4]:

$$term\_f := f_h \le f_l + tol\_f \cdot (|f_{\mathrm{init}}| + tol\_f), \quad \text{where } f_{\mathrm{init}} = f(x_{\mathrm{init}}).$$

MATLAB — file fmins.m [3]:

$$term\_f := \max_{j \ne l} |f_j - f_l| \le tol\_f, \quad \text{(equivalent to } term\_f := f_h - f_l \le tol\_f),$$

$$term\_x := \max_{j \ne l} \|x_j - x_l\|_\infty \le tol\_x.$$

Note: $\| \ \|_1$ has been used instead of $\| \ \|_\infty$ in version 3.5g.

Higham — file nmsmax.m [1]:

$$term\_x := \frac{\max\limits_{j \ne l} \|x_j - x_l\|_1}{\max\{1, \|x_l\|_1\}} \le tol\_x.$$

Rowan — file simplx.f [9]:

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

where $x_l^{(0)}$, $x_h^{(0)}$ denote the best and the worst point in the initial simplex S_{init}.

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

The *term_f* test complexity is $\Theta(1)$ when only 2 or 3 function values are used to compute it, or $\Theta(n)$, if it is based on all $n+1$ function values. Even without a *term_x* test, the latter case may be inefficient according to (11).

The *term_x* test is the real bottleneck. Only Rowan's *term_x* test complexity is $\Theta(n)$, and all the other (when present) require $\Theta(n^2)$ operations.

We can conclude that none of these implementations can handle discontinuous functions efficiently!

5. Some typical examples

The efficiency of the NMS depends on how fast the termination test is with respect to the function evaluation. This is clearly illustrated by the following examples.

Example 1. *Let a_i, $b_i \in \mathbb{R}$, for $i \in \mathbb{N}$, be given numbers and define $f : \mathbb{R}^n \to \mathbb{R}$ as*

$$f(x) = \sum_{i=1}^{n} a_i (x(i) - b_i)^2, \quad x \in \mathbb{R}^n.$$

$T_f(x, n)$ obviously depends only on n, and we have $T_f(n) = \Theta(n)$. From Lemma 1, this is (almost) the fastest possible function evaluation. Even a linear TERM *test affects the efficiency, and a $\Theta(n^2)$ test seriously degrades it.*

Example 2. *For each $n \in \mathbb{N}$, we have $m(n) \in \mathbb{N}$ measured data points $(y_k, g_k) \in \mathbb{R}^2$. Measured values g_k at y_k have to be well approximated by a function $g : \mathbb{R}^{n+1} \to \mathbb{R}$, which depends on n unknown parameters $x(i)$, $i = 1, \ldots, n$, and $y \in \mathbb{R}$. To determine the parameter vector $x \in \mathbb{R}^n$, we can use the discrete least squares approximation and minimize $f : \mathbb{R}^n \to \mathbb{R}$ defined by*

$$f(x) = \sum_{k=1}^{m(n)} (g(x, y_k) - g_k)^2, \quad x \in \mathbb{R}^n.$$

Suppose that g is a "simple" function to compute, such that $T_g(x, y) = \Theta(n)$, regardless of x or y. To compute the value of f, we have to evaluate g at all $m(n)$ points y_k, for a given x. Again, T_f does not depend on x and $T_f(n) = \Theta(m(n)n)$. In all applications we have $m(n) \geq n$, or $m(n) = \Omega(n)$, so $T_f(n) = \Omega(n^2)$. The evaluation of f is so slow that even a $\Theta(n^2)$ termination test barely affects the efficiency.

Our final example is from [10] and lies between these two "extremes".

Example 3. *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 the maximal values of*

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

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

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

In this case, a linear term_x is admissible, as it is $\Theta(m^2)$. Unfortunately, Rowan's test performance is quite poor in practice. Obviously, a $\Theta(n^2) = \Theta(m^4)$ test is out of question, as the evaluation of f is slow enough by itself.

6. Conclusion

As we have seen, none of the current termination tests satisfies (9). Consequently, none of them is efficient for **all** functions f.

The question remains, whether we can design a simple and efficient domain convergence test. Indeed, such a test has been constructed, but the description and analysis of this test is beyond the scope of this paper. The new test is currently undergoing an extensive numerical testing and verification. These results will be published in a future report.

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, Sept. 1995.

[2] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.

[3] The MathWorks, Inc., *MATLAB Language Reference Manual, Version 5*, Natick, Massachusetts, 1996.

[4] J. C. Nash, *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, second ed., Adam Hilger, Bristol, 1990. (with Errata and Related Notes, 23 Jan 1995.).

[5] J. A. Nelder and R. Mead, *A simplex method for function minimization*, Comput. J., 7 (1965), pp. 308–313.

[6] The Numerical Algorithms Group, Ltd., *NAG Fortran Library Manual, Mark 18*, Oxford, England, 1997.

[7] J. M. Parkinson and D. Hutchinson, *An investigation into the efficiency of variants on the simplex method*, in Numerical Methods for Nonlinear Optimization, F. A. Lootsma, ed., 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. Reprinted with corrections 1994., corrected to software version 2.04.

[9] T. H. Rowan, *Functional Stability Analysis of Numerical Algorithms*, Ph.D. thesis, University of Texas, Austin, May 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, Sep 30 – Oct 2, 1998., I. Aganović, T. Hunjak, and R. Scitovski, eds., Osijek, 1999, Croatian Operational Research Society, pp. 169–176.

[11] Visual Numerics, Inc., *IMSL MATH/LIBRARY User's Manual, Version 3.0*, Houston, Texas, 1994.