Numerical Experience with a Class of Trust-Region Algorithms for Unconstrained Smooth Optimization

Abel Soares Siqueira∗  Geovani Nunes Grapiglia†

Nov, 2016

Abstract

In this paper we investigate the numerical performance of trust-region algorithms in which the trust-region radius is updated by a nonlinear rule according with the quality of the models. This class of algorithms fits into the Nonlinear Stepsize Control framework recently proposed by Toint (Optimization Methods and Software 28: 82–95, 2013). The nonlinear control of the trust-region radius is characterized by a pair \((\alpha, \beta)\) of user-defined parameters. Notable particular cases are the standard trust-region algorithm and the Fan-Yuan trust-region algorithm, which are obtained, respectively, with the traditional choices \((\alpha, \beta) = (1, 0)\) and \((\alpha, \beta) = (1, 1)\). As expected, our numerical results show that the numerical behaviour in this class of trust-region algorithms can vary greatly with different choices for \((\alpha, \beta)\).

Keywords: Unconstrained Optimization, Trust-Region Algorithms and Algorithmic Parameters

1 Introduction

Consider the unconstrained optimization problem

$$\min_{x \in \mathbb{R}^n} f(x),$$

where \(f : \mathbb{R}^n \to \mathbb{R}\) is continuously differentiable and its gradient \(\nabla f : \mathbb{R}^n \to \mathbb{R}^n\) is Lipschitz. There are several iterative methods to solve problem (1). Among them, the class of trust-region algorithms has been subject of intense research for more than forty years (see, for instance, \([12, 19, 28, 32–34, 39]\) and references therein). In summary, at the \(k\)-th iteration of the standard trust-region algorithm, a trial step \(s_k \in \mathbb{R}^n\) is obtained by solving (approximately) the trust-region subproblem

$$\begin{align*}
\min_{s \in \mathbb{R}^n} \quad & m_k(x_k + s) \\
\text{s. t.} \quad & \|s\| \leq \Delta_k,
\end{align*}$$

where \(g_k = \nabla f(x_k), B_k \in \mathbb{R}^{n \times n}\) is the Hessian of \(f\) at \(x_k\) or an approximation thereof, \(\|\cdot\|\) is usually the euclidean norm and \(\Delta_k > 0\) is the so-called trust-region radius. Based on the value of the ratio

$$\rho_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)},$$

the next iterate \(x_{k+1}\) is defined by

$$x_{k+1} = \begin{cases} 
  x_k + s_k, & \text{if } \rho_k \geq \eta_1, \\
  x_k, & \text{otherwise,}
\end{cases}$$

\[\text{Keywords: Unconstrained Optimization, Trust-Region Algorithms and Algorithmic Parameters}\]
while the trust-region radius $\Delta_k$ is updated by the rule

$$
\Delta_{k+1} = \begin{cases} 
\sigma_1 \Delta_k, & \text{if } \rho_k < \eta_1, \\
\Delta_k, & \text{if } \rho_k \in [\eta_1, \eta_2), \\
\sigma_2 \Delta_k, & \text{if } \rho_k \geq \eta_2,
\end{cases}
$$

(5)

where $0 < \eta_1 \leq \eta_2 < 1$ and $0 < \sigma_1 < 1 < \sigma_2$ are constants chosen \textit{a priori} by the user. Finally, a symmetric matrix $B_{k+1} \in \mathbb{R}^{n \times n}$ is computed and the process is repeated with $k := k + 1$.

Global convergence of the standard trust-region algorithm can be proved if at each iteration the trial step $s_k$ provide a sufficient decrease in the quadratic model and if $\|B_k\| \leq c_0 + c_1 k$ for some constants $c_0, c_1 > 0$ (see, for instance, [12]). On the other hand, the local superlinear convergence is usually proved by showing that the trust-region constraint $\|s\| \leq \Delta_k$ is inactive for all $k$ large enough. This guarantees that $B_k s_k = -g_k$ for all $k$ sufficiently large, and so, the $Q$-superlinear convergence of $\{s_k\}$ follows directly from the results of Dennis and Moré [13] under mild conditions (specifically, see Theorem 2.2 in [13]). On its turn, the common way to prove that $\|s\| < \Delta_k$ for $k$ large enough is to show that $s_k \to 0$ and that $\Delta_k$ is bounded away from zero. Inspired by the Levenberg-Marquardt method [26, 27], Fan and Yuan [18] proposed a modified trust-region algorithm, with $Q$-superlinear convergence, which remarkably allows $\Delta_k \to 0$. Their approach consists in update $\Delta_k$ by the rule

$$
\Delta_{k+1} = \delta_{k+1} \|g_{k+1}\|,
$$

(6)

where $\delta_{k+1}$ is chosen in a way similar to (5). The numerical results reported in [18] on the MGH test problems [29] indicated that the Fan-Yuan trust-region algorithm is competitive with the standard trust-region algorithm. This motivated the use of the update rule (6) in combination with a line-search technique [15] and also its adaptation for trust-region algorithms designed to solve nonlinear equations [16, 17].

In a broader context, including trust-region and regularization algorithms, a Nonlinear Stepsize Control (NSC) framework has been proposed by Toint [38] with the purpose to provide an unified setting to prove global convergence results for a wide range of algorithms. In particular, the NSC covers the standard trust-region algorithm [12, 32], the basic ARC algorithm proposed by Cartis et al. [10, 11], the quadratic regularization method for $f(x) = \|F(x)\|$ proposed by Nesterov [30] (as extended by Bellavia et al. [6], the quadratic regularization methods for $f(x) = (1/2)\|F(x)\|^2$ proposed by Zhang and Wang [40] and by Fan [16] and also the Fan-Yuan trust-region algorithm described above. One of the key features of the NSC framework that allows it to include all these algorithms is the control of the stepsizes by a generalized trust-region radius:

$$
\Delta_k = \delta_k^\alpha g_k^\beta,
$$

(7)

where $\alpha \in (0, 1]$, $\beta \in [0, 1]$, $\chi_k > 0$ is a criticality measure and $\delta_k$ is updated by a rule similar to (5). For example, the standard trust-region algorithm corresponds to the pair $(\alpha, \beta) = (1, 0)$, while the Fan-Yuan trust-region algorithm corresponds to $(\alpha, \beta) = (1, 1)$, both with $\chi_k = \|g_k\|$ for all $k$.

Results about the global convergence of the NSC framework were proved by Toint [38] under the assumption that the sequence $\{\|B_k\|\}$ is bounded from above. Grapiglia et al. [21, 22, 23] proved the global convergence of the NSC under a weaker condition and also provided worst-case complexity estimates. Despite these theoretical advances in the understanding of the NSC, the numerical performance of this framework and its dependence with respect to $\alpha$ and $\beta$ have not been addressed yet. Motivated by this observation, in this paper, we investigate the numerical behaviour of a subclass of NSC algorithms composed by “nonlinear” trust-region algorithms in which

$$
\Delta_k = \delta_k^\alpha \|g_k\|^\beta, \text{ for all } k.
$$

(8)

Specifically, we want to answer the following questions:

1. How sensitive is the numerical performance of nonlinear trust-region algorithms with respect to different choices of $\alpha$ and $\beta$?

2. Are the pairs $(\alpha, \beta) = (1, 0)$ and $(\alpha, \beta) = (1, 1)$ the best choices?

Similar analysis has been made regarding other parameters of optimization algorithms. We mention here the work of Gould et al. [20], which we use as a starting point for our analysis. Furthermore, finding optimal parameter through derivative-free optimization was investigated on several works [2, 5, 31].


The rest of this paper is organized as follows. In Section 2, we describe the class of nonlinear trust-region and we present a short review of the main theoretical results already known for these algorithms. In Section 3, we give a detailed explanation of the workflow of our numerical tests. Finally, in Section 4 we report the results. As expected, they show that the performance of nonlinear trust-region algorithms is very sensitive to the choices of the parameters, including $\alpha$ and $\beta$. In particular, we have identified pairs $(\alpha, \beta)$ which are more efficient than the traditional choices $(\alpha, \beta) = (1, 0)$ and $(\alpha, \beta) = (1, 1)$.

2 Nonlinear trust-region algorithms

Roughly speaking, NSC algorithms are specified by the following elements:

1. models $m_k(x_k + s) : \mathbb{R}^n \to \mathbb{R}$ of $f(x_k + s)$ from which the trial steps $s_k$ are computed;
2. sequences $\{\omega_k\}$, $\{\phi_k\}$, $\{\psi\}$ and $\{\chi_k\}$ that play the role of criticality measures; and
3. powers $\alpha \in (0, 1]$ and $\beta \in [0, 1]$ that provide the nonlinear control of stepsizes by means of the generalized trust-region.

We refer the reader to [38] for more details. Considering this framework, we shall define the class of nonlinear trust-region algorithms as being the class specified by the choices:

1. $m_k(x_k + s) = f(x_k) + g_k^T s + \frac{1}{2} s^T B_k s$, for all $k$;
2. $\omega_k = 1$, $\phi_k = \psi_k = \chi_k = \|g_k\|$, for all $k$; and
3. $\alpha \in (0, 1]$ and $\beta \in [0, 1]$.

Thus, a generic nonlinear trust-region algorithm can be described as follows:

\begin{algorithm}
\caption{Nonlinear Trust-Region Algorithm}
\begin{algorithmic}[1]
\STATE Parameters: $\alpha, \beta \in [0, 1]$, $\alpha \neq 0$, $0 < \eta_1 \leq \eta_2 < 1$ and $0 < \sigma_1 < 1 < \sigma_2$.
\STATE Given $x_0 \in \mathbb{R}^n$, $B_0 \in \mathbb{R}^{n \times n}$ symmetric, $\delta_0 > 0$, define $\Delta_0 = \delta_0^\alpha \|g_0\|^\beta$ and $k = 0$.
\FOR{$k = 0, 1, \ldots$}
\STATE Compute a trial step $s_k$ by solving (approximately) the trust-region subproblem $\mathbf{(2)}$.
\STATE Compute $\rho_k$ by $\mathbf{(3)}$.
\STATE Set $x_{k+1}$ by $\mathbf{(4)}$.
\STATE Set $\delta_{k+1}$ by the rule
\[ \delta_{k+1} = \begin{cases} 
\sigma_1 \delta_k, & \text{if } \rho_k < \eta_1, \\
\delta_k, & \text{if } \rho_k \in [\eta_1, \eta_2), \\
\sigma_2 \delta_k, & \text{if } \rho_k \geq \eta_2. 
\end{cases} \]
\STATE Define
\[ \Delta_{k+1} = \delta_{k+1}^\alpha \|g_{k+1}\|^\beta. \]
\STATE Compute $B_{k+1} \in \mathbb{R}^{n \times n}$.
\ENDFOR
\end{algorithmic}
\end{algorithm}

In order to present a summary of the theoretical results already known for Algorithm 1, we consider the following assumptions:

A1 The objective function $f : \mathbb{R}^{n \times n} \to \mathbb{R}$ is continuously differentiable and its gradient $\nabla f : \mathbb{R} \to \mathbb{R}^n$ is Lipschitz.

A2 The level set $L_0 = \{x \in \mathbb{R}^n | f(x) \leq f(x_0)\}$ is bounded.
A3 For all \( k \geq 0 \), the trial step \( s_k \) satisfies
\[
m_k(x_k) - m_k(x_k + s_k) \geq \kappa_c \|g_k\| \min \left\{ \frac{\|g_k\|}{1 + \|B_k\| \Delta_k} \right\},
\]
for some constant \( \kappa_c > 0 \).

A4 For all \( k \geq 0 \), \( \|B_k\| \leq c_0 + c_1 k \) for some constants \( c_0, c_1 > 0 \).

A4' For all \( k \geq 0 \), \( \|B_k\| \leq c_0 \), for some constant \( c_0 > 0 \).

Regarding the global convergence of Algorithm 1, a liminf-type result is given by the theorem below.

**Theorem 1** Suppose that A1-A4 hold and let \( \{x_k\} \) be a sequence generated by Algorithm 1. Then,
\[
\liminf_{k \to +\infty} \|g_k\| = 0.
\]
Therefore, at least one limit point of \( \{x_k\} \) (if any exists) is a critical point of \( f \).

**Proof** See Corollary 1 in [22]. \( \square \)

With respect to the worst-case complexity of Algorithm 1, we have the following result when \( f \) is possibly nonconvex:

**Theorem 2** Suppose that A1-A3 and A4' hold. Then, given \( \varepsilon \in (0,1) \), Algorithm 1 takes at most \( O(\varepsilon^{-2}) \) iterations to ensure \( \|g_k\| \leq \varepsilon \).

**Proof** See Theorem 2 in [23]. \( \square \)

When \( f \) is convex or strongly convex, improved complexity bounds can be obtained under the additional assumption

A5 The powers \( \alpha \) and \( \beta \) satisfy the equality \( \alpha + \beta = 1 \).

In this case, we have the following result:

**Theorem 3** Suppose that A1-A3, A4' and A5 hold and let \( \varepsilon \in (0,\varepsilon^{-2}) \). If \( f \) is convex, then Algorithm 1 takes at most \( O(\varepsilon^{-1}) \) iterations to achieve
\[
f(x_k) - f_* \leq \varepsilon,
\]
where \( f_* \) is the global minimum of \( f \). Additionally, if \( f \) is strongly convex and \( \eta_1 \) is sufficiently small, then this worst-case complexity bound is reduced to \( O(\log(\varepsilon^{-1})) \) iterations.

**Proof** See Theorem 3.1 in [21]. \( \square \)

**Corollary 1** Under the same assumptions of Theorem 3, Algorithm 1 takes at most \( O(\log(\varepsilon^{-1})) \) iterations to ensure \( \|\nabla f(x_k)\| \leq \varepsilon \) when \( f \) is strongly convex and \( \eta_1 \) is sufficiently small.

**Proof** See Lemma 3.2 and Theorem 3.3 in [21]. \( \square \)

### 3 Numerical Experiment

To verify how the choice of \( \alpha \) and \( \beta \) affects the performance of the Algorithm 1 we’ve chosen to run the algorithm for many different choices of \( (\alpha, \beta) \) and analyze the results. We used the traditional parameter choices
\[
\eta_1 = \frac{1}{4}, \quad \eta_2 = \frac{3}{4}, \quad \sigma_1 = \frac{1}{6}, \quad \sigma_2 = 4,
\]
and used the Steihaug-Toint method [37] to solve the subproblem (2), with \( B_k = \nabla^2 f(x_k) \). We stop the execution of Algorithm 1 when \( \|\nabla f(x_k)\| \leq \varepsilon \) for \( \varepsilon = 10^{-8} \). In this case we declare convergence. We also stop our algorithm if it exceeds 1000 iterations or 1 minute of elapsed time, declaring failure to converge.
We considered a small set of 58 CUTEst problems, obtained from the all unconstrained problems in CUTEst with 10 or less variables, removing 6 problems that were failing for most choices of \((\alpha, \beta)\). All problems would usually run in a few milliseconds to either a failure or convergence. To prevent errors in the time measurement, we used the strategy suggested in [11], which consist of running the same problem repeatedly until the accumulated time is close to 0.1 seconds and taking the average. This is repeated three times, and the best result is used as the elapsed time of the algorithm on the problem. In practice, first we run the problem a single time, obtaining an elapsed time \(t_{\text{first}}\). If the problem does not converge, we simply declare failure, and go to the next problem. Otherwise, we estimate the number of times \(N\) needed to run the problem for an accumulated time of 0.1 seconds, that is,

\[
N = \left\lceil \frac{0.1}{t_{\text{first}}} \right\rceil.
\]

Then, we run the algorithm \(3N\) times and compute three averages of \(N\) runs. The best average is used.

Our implementation of the algorithm was written in C, and can be obtained on [35]. We ran our code on a computer with an i5-4440 processor and 8Gb of RAM. We discretized the domain for \((\alpha, \beta)\) into a grid \(50 \times 51\). For each \((\alpha, \beta)\) pair in that grid, we ran the algorithm on the set of problems and collected the output that may be of interest. This defines, effectively, 2550 different algorithms, and we’re gonna compare these algorithms using the raw data and the performance profile [14, 36] as well. All plots that are not performance profile were created in the Julia language [7, 8], with the Plots package [9] and PyPlot backend [24, 25].

### 3.1 Elapsed time and number of iterations

The first way to analyze the results is to consider only those problems in which all algorithms converged. This reduces the test set to 30 problems. For each algorithm, we compute the total elapsed time and total number of iterations for these problems, creating Figure 1. The solid and dashed black lines mark the position of the algorithms defined by

\[(\alpha, \beta) = (1, 0)\] and \((\alpha, \beta) = (1, 1)\), respectively.

Figure 2 shows the same plot zooming in the lower left corner of this comet-like graphic. In this comet we can separate the “best” algorithms, that is, those that are not dominated by any other \((\alpha, \beta)\) selection in neither iterations.

---

1The problems were HIELOW, LOGHAIRY, MEYER3, OSCIPATH, STRATEC and HEART6LS. Except for HEART6LS, the others didn’t work for any choice of \((\alpha, \beta)\).
nor elapsed time. These are numbered on Figure 2a and their \((\alpha, \beta)\) values are shown on Figure 2b. These figures

Figure 2: Best algorithms for these 30 problems

are already a good indication of the large variation of efficiency among the choice of \((\alpha, \beta)\). Interestingly enough, the
best algorithms form two clusters in the \(\alpha \times \beta\) space, one for \(\alpha\) around 0.5 and \(\beta\) smaller than 0.2, and another which
consists of only the point \((\alpha, \beta) = (0.62, 0.96)\).

To further investigate this efficiency, let’s look at two other situations, the “bad” algorithms, and the “good” al-
gorithms. The bad algorithms are those that are many times worse than the best algorithm in either iteration count
or elapsed time. The good algorithms are those that are not much worse than the best algorithm in neither iteration
time nor elapsed time. The quantities related to how many times worse were chosen so that the number of good
and bad algorithms were about 10% of the total number of algorithms. Figure 3 shows this classification in both the
iteration \(\times\) elapsed times axis and the \(\alpha \times \beta\) space.

Figure 3: Classification of good and bad algorithms

The bad algorithms cluster confirms an expect characteristic of choosing a very small \(\alpha\). The trust region update
will follow mostly \(\|g_k\|^{\beta}\), which does not have the control imposed by \(\delta_k\). For a model that accurately represents \(f\)
everywhere, this can be a good thing, since the solution of both the model and the \(f\) will be close. However, for most
nonlinear problems this is a bad thing, because the model won’t be good and there won’t be anything limiting the steps.
If both \(\alpha\) and \(\beta\) are small, though, the algorithms perform a little better, possibly because the steps are close to a fixed
length. On the other hand, we have many good algorithms for larger values of \( \alpha \), and different values of \( \beta \). Around \((\alpha, \beta) = (0.4, 0.2)\) we have a large cluster of good algorithms. we have a small cluster close to \( \alpha = 1 \) and \( \beta = 0.2 \), and around the segment \( \beta = 1 \), \( 0.4 \leq \alpha \leq 1 \) we have a few good algorithms, with one being very close to \((1, 1)\). However, the actual performances of \((1, 0)\) and \((1, 1)\) are not commendable. \((1, 0)\) has about \(32\%\) more time than the best values, and \((1, 1)\) has about \(73\%\) more time than the best values. At this point, we can already answer partially the questions posed. The choice of \((\alpha, \beta)\) does affect a great deal the results of the algorithm, at least for these 30 problems. All algorithms shown have the same robustness, but one of them takes about \(5\) times more iterations and almost \(10\) times more time than the best algorithms. Furthermore, \((1, 0)\) is not the best choice in neither iterations nor time, and \((1, 1)\) is even worse. Nevertheless, further investigation is required for a more definitive answer.

### 3.2 Robustness

So far, we’ve considered 30 problems from the set of 58, which amounts to \(51.7\%\) of the total number of problems. However, all algorithms have greater robustness than that. The least number of problems solved by an algorithm was 45 \((77.6\%)\), while the most was 56 \((96.6\%)\), all in the set of 58 problems. Hence, we investigate how the algorithms behave regarding robustness. First, Figure 4 show a heatmap in the \(\alpha \times \beta\) space, where the darkest points are more robust. The fact that Figure 4a shows no discernible pattern is another indication of how sensitive the algorithm is to these parameters. In fact, the traditional choice \((1, 0)\) has \(89.66\%\) robustness and it’s neighbour \((0.98, 0)\) has \(94.83\%\) robustness. In addition, look at the two points inside the circle close to \((0.3, 0.1)\). The black one has \(94.83\%\) robustness, while the white one has only \(81.03\%\) robustness. This is the most disparate pair of adjacent points, regarding robustness. Another factor to pay attention is that the robustness does not increase or decrease in a special way inside the regions that were considered “good” and “bad”. In Figure 4b, we have the top three tiers of robustness, showing that some of the most robust options are clustered in the lower right side, that is, larger \(\alpha\) and smaller \(\beta\). Notice, however, that there are many good options scattered around.

### 3.3 Efficiency

On the search for the best \((\alpha, \beta)\), we’re gonna create performance profiles [14] for every algorithm \((\alpha, \beta)\) comparing against \((1, 0)\). The advantage of using performance profiles is that they compare robustness too, so we won’t be limiting this comparison to the 30 problems, but to all 58 problems. Given a set of problems \(P\) and a set of algorithms \(S\), we define \(c_{s,p}\) as the cost for algorithm \(s \in S\) to solve a problem \(p \in P\). This cost is usually time or number of iterations.
In our case, we’ll use the latter. Then, for each algorithm \( s \in S \) that solves problem \( p \), we define the performance ratio

\[
    r_{s,p} = \frac{c_{s,p}}{\min\{c_{s',p} \mid s' \in S \text{ solves } p\}}.
\]

If algorithm \( s \) does not solve \( p \), then we simply set \( r_{s,p} = +\infty \).

Finally, we define the performance function of an algorithm \( s \) as

\[
    P_s(t) = \frac{|\{p \in P \mid r_{s,p} \leq t\}|}{|P|}.
\]

To compare the algorithms, we usually plot their performance functions on a finite interval \([1, r_f]\) for which every function stabilizes. The value \( P_s(t) \) computes the fraction of problems algorithm \( s \) can solve with at most \( t \) many times the cost of the best algorithm. \( P_s(1) \) corresponds to the number of problems that algorithm \( s \) solved faster, or as fast, as the other algorithms in \( S \). The value \( P_s(r_f) \), for a large enough \( r_f \), corresponds the maximum number of problems that algorithm \( s \) has solved. The quantities \( P_s(1) \) and \( P_s(r_f) \) are called efficiency and robustness of \( s \), respectively. Since we can’t visually examine all performance profile plots in a timely manner, we look only at the efficiency and the robustness of each algorithm against \((1,0)\).

On Figure 5a, we show the efficiency and robustness of each pair. In this figure, there are only two pairs of robustness and efficiency values that are not dominated by any others, and at each point there are two \((\alpha, \beta)\) values. Namely, points \((0.78,0.18)\) and \((0.96,0.04)\) have robustness of 94.8% and efficiency of 77.6%, and points \((0.7,0.2)\) and \((0.96,0.18)\) have robustness of 96.6% and efficiency of 70.7%. Figure 5b shows these four \((\alpha, \beta)\) values. These 4 \((\alpha, \beta)\) values are now candidates for best choice. Figure 6 shows the performance profiles of each of them against \((1,0)\).

Naturally, the \((\alpha, \beta)\) choices are better in robustness and efficiency, but all of them lose to \((1,0)\) at some point. Also, notice the scale. The differences are not very significative. For a more complete analysis, we make new computational tests using all 173 unconstrained problems in CUTest for these specifics \((\alpha, \beta)\) values and \((1,0)\) and \((1,1)\). First, in Figure 7 we show a comparison of \((1,0)\) and \((1,1)\) in this simple implementation of a trust region algorithm. We can see that the choice \((1,0)\) is significantly superior than \((1,1)\), not only in efficiency and robustness, but for all performance ratio values. This is very interesting since \((1,1)\) has been proposed as an improvement over \((1,0)\). One could argue that this happens due to the nature of the problems, since CUTest is a very general collection, and these choices have been analyzed in [18] on nonlinear least squares problems. However, as we shall see in the next section, \((1,0)\) is also superior than \((1,1)\) on nonlinear least squares problems.
Next, Figure 8 shows the performance profile of the same \((\alpha, \beta)\) values in Figure 6. These figures are very interesting, since, except for \((0.96, 0.18)\), the other \((\alpha, \beta)\) choices are more efficient than \((1, 0)\), but on the other hand, \((1, 0)\) is more robust then each other choice of \((\alpha, \beta)\), albeit slightly. This happens, despite the \((\alpha, \beta)\) chosen being the most...
robust against \((1,0)\). This indicates that the analysis on the smaller set does not reflect the results of the larger set, which has already been observed in many other works.

### 3.4 Subset of nonlinear least squares problems

A natural question that arises in parameter choices is how the type of problems in the repository affect the best parameters for the algorithm. In this section we’ll analyze the \((\alpha, \beta)\) choices for the subset of nonlinear least squares problems.

The nonlinear least squares problem can be stated as

\[
\min \frac{1}{2} \| r(x) \|^2 = \frac{1}{2} \sum_{i=1}^{m} r_i(x)^2,
\]

where \(r_i: \mathbb{R}^n \rightarrow \mathbb{R}\) for \(i = 1, \ldots, m\), for some \(m\) usually much larger than \(n\). This problem is also described by saying that the objective function of the unconstrained problem is a sum-of-squares function.

Inside CUTEst, there are 85 unconstrained problems with sum-of-squares functions. Of those, 33 fall under the 58 initial problems we considered, and 17 of those converge in every choice of \((\alpha, \beta)\). We can make similar plots using this restricted set, but we can focus on just a few of these plots instead, since much is the same, like the comet-like structure. The most important plots, are the ones showing the \((\alpha, \beta)\) values, namely, figures 2b, 3b and 5b. The corresponding plots are in Figure 9. Specifically, Figure 9a shows the best choices for \((\alpha, \beta)\) considering the 17 converging problems, based on the time-iteration non-dominated points. Figure 9b shows the good and bad choices of this same selection. Finally, Figure 9c shows the best choices considering the performance profiles against \((1,0)\).
The most noteworthy difference to the general problems is the absence of good algorithms for larger values of $\beta$. Here, at least for the 17 converging problems, there are no good algorithms with $\beta \geq 0.8$. With respect to the performance profiles, we now have three best choices. They are $(0.76, 0.18)$, $(0.78, 0.08)$ and $(0.96, 0.04)$, all with $96.97\%$ of convergence at most and $75.76\%$ at the efficiency point. Notice how $(0.96, 0.04)$ appears both in this subset as well as the larger set, and also that $(0.76, 0.18)$ here and $(0.78, 0.18)$ in the larger set are neighbours. Figure 10 shows the performance profile of these choices against $(1, 0)$ in the 85 sum-of-squares problems, as well as $(1, 0)$ vs $(1, 1)$. Once more we see that $(1, 0)$ is much superior than $(1, 1)$, and furthermore, all other choices are more efficient.
that \((1, 0)\). However, the robustness of \((1, 0)\) is not much superior in this case. In fact, both \((0.96, 0.04)\) and \((0.76, 0.18)\) solves one problem more than \((1, 0)\).

3.5 Sensitivity of the stopping tolerance parameter

Theorem 2 stated that the number of iterations to reach the stopping criterium is bounded above by \(O(\varepsilon^{-2})\) when \(f\) is nonconvex, so a natural question is how the choice of \(\varepsilon\) affects the best values found for \((\alpha, \beta)\). Notice that the number of iterations and elapsed time will change, as the problem becomes “easier” to solve, but most of the analysis is made by comparison, so that won’t be an issue. This section presents some of the figures of previous sections alongside plots obtained for \(\varepsilon = 10^{-4}, 10^{-6}, 10^{-8}\). Figures 11 to 15 show these comparisons.

Some strange things happen on some of these figures that are worthy mentioning. First, the number of iterations for \(\varepsilon = 10^{-4}\) in Figure 11 is larger. This happens because more problems are being solved, and that is the total sum of iterations for these problems.

Second, the middle plot of Figure 11, the time-iteration comet for \(\varepsilon = 10^{-6}\), has a strange leap. After some investigation, we found this was mostly due to the problem BIGGS6, where some algorithms solved it with 35 iterations.
or less, and the rest took 195 iterations or more. There isn’t a clear relation on the values of $(\alpha, \beta)$ and the number of iterations needed for this problem. The problem BIGGS6 is a classical sum-of-squares problem. It’s definition is

$$
\min \sum_{n=1}^{13} \left(5e^{-n} - e^{-0.1n} - 3e^{-0.4n} + x_3e^{-0.1n_1} - x_4e^{-0.1n_2} + x_6e^{-0.1n_3}\right)^2,
$$

and despite having some easy solutions, it is computationally hard.

Last, the swarm of points in the plots of Figure 14 indicates that the vast majority of $(\alpha, \beta)$ options have a very good convergence. This is expected, of course. The minimum robustness for these tolerances are 65.5% (38 problems) and 79.3% (46 problems) for $\varepsilon = 10^{-4}$ and $10^{-6}$, respectively.

Finally, the best points from the performance profile, in Figure 15, have some difference, but they all have some point in the vicinity of $(0.96, 0.04)$.

### 4 Conclusion

We conclude that the choice of $(\alpha, \beta)$ can influence a great deal on the algorithmic results. In fact, under the conditions of these comparisons, we can say that $(1, 0)$ is not the best choice, i.e., the traditional trust region algorithm is not the best option inside the NSC framework, and $(1, 1)$ is definitely not better. For a specific type of problem, with enough test problems, we believe that a better option can be found with some experimentation. On the other hand, a best-for-all solution may not be possible, due to the high sensitivity to the parameters. In either case, a larger number of test problems is desirable, though that will take much more time to complete. Our suggestion, in any case, is to give $\beta$ a little nudge away from 0. Namely, try $(\alpha, \beta) = (0.96, 0.04)$, which was competitive in both general problems and sum-of-squares problems.

For future works in this area, we recommend ignoring $\alpha < 0.2$, as we saw that it is largely a waste of time. A different approach can also be taken, completely ignoring the elapsed time, and since the time measurement strategy won’t be used, this will allow the fast problems to be solved more quickly. Although, if using larger problems, they usually take a little longer than 0.1s to be solved, and therefore there won’t be a large economy of time. Another
possible work is the parameter optimization using a derivative-free solver. With the high sensitivity to the parameters, there are many local minimizers, and therefore the problem is very challenging.
References


[38] Toint, Ph.L.: Nonlinear stepsize control, trust regions and regularizations for unconstrained optimization. Optimization Methods and Software 28(1), 82–95 (2013)
