NOMAD version 4: Nonlinear optimization with the MADS algorithm *

Charles Audet † Sébastien Le Digabel ‡ Viviane Rochon Montplaisir§
Christophe Tribes ¶
April 30, 2021

Abstract: NOMAD is software for optimizing blackbox problems. In continuous development since 2001, it constantly evolved with the integration of new algorithmic features published in scientific publications. These features are motivated by real applications encountered by industrial partners. The latest major release of NOMAD, version 3, dates from 2008. Minor releases are produced as new features are incorporated. The present work describes NOMAD 4, a complete redesign of the previous version, with a new architecture providing more flexible code, added functionalities and reusable code. We introduce algorithmic components, which are building blocks for more complex algorithms, and can initiate other components, launch nested algorithms, or perform specialized tasks. They facilitate the implementation of new ideas, including the MegaSearchPoll component, warm and hot restarts, and a revised version of the PSDMADS algorithm. Another main improvement of NOMAD 4 is the usage of parallelism, to simultaneously compute multiple blackbox evaluations, and to maximize usage of available cores. Running different algorithms, tuning their parameters, and comparing their performance for optimization is simpler than before, while overall optimization performance is maintained between versions 3 and 4. NOMAD is freely available at [www.gerad.ca/nomad](http://www.gerad.ca/nomad) and the whole project is visible at [github.com/bbopt/nomad](https://github.com/bbopt/nomad).

Keywords: Optimization software, blackbox optimization, derivative-free optimization, mesh adaptive direct search.

1 Introduction

NOMAD is software designed for the class of blackbox optimization (BBO) problems [17]. The term blackbox indicates that there is no information except the input and output. There is no analytic description of the objective and/or constraint functions, there are no available derivatives, possibly because they are not differentiable, and the functions may occasionally fail to return valid output and may require significant computational time to evaluate. This makes BBO problems difficult to solve, in the sense that many opti-
mization algorithms and heuristics cannot be applied. A typical BBO problem is a computer simulation of an engineering or physical problem.

The development of NOMAD was initiated in 2001 to implement direct search algorithms, and major version 3 was released in 2008 [45]. The mesh adaptive direct search (MADS) algorithm [12] is at the core of NOMAD 3; it provides a flexible framework and is supported by a rigorous hierarchical convergence analysis based on various degrees of smoothness of the functions defining the problem. Since its original release, minor releases of NOMAD 3 have included several improvements and additions of algorithms to solve a variety of blackbox optimization problems efficiently.

NOMAD and has proven its usefulness in scientific papers as well as in established companies. Our own work includes contributions in hydrology [2, 57, 51], pharmacology [57], metamaterial design [15], alloy design [36, 34, 37], chemical engineering [8, 40] and bioinformatics [33]. Many other researchers use NOMAD in a variety of fields. In astrophysics for example, NOMAD is used for black hole observation [49], for tracking the interstellar object 1I/2017SO [41], for kinematics analysis of galaxies [54], and for gravitational wave detection, in a paper with more than 700 co-authors [1]. Hundreds of applications are reported in the surveys [7, 5, 35], including many on energy, engineering design, and materials science.

In retrospect, the main development avenues of the NOMAD software and the MADS algorithm may be classified into three categories:

- **Algorithmic improvement.** The MADS algorithm was modified to reduce the number of evaluations through constraint handling techniques [13, 22], by dynamically scaling and exploiting the specificity of variables [21, 23, 24], and by the improved integration of surrogates [10, 20, 58, 59, 60].

- **Sub and super-algorithms.** MADS may call other optimization algorithms during its execution. Sub-algorithms, when used under adequate conditions, may produce good candidate points for evaluation. Using the right candidates has a strong influence on the performance of the software. Useful algorithms and techniques are proposed in [3, 18, 6, 9, 27]. Sub-algorithms may also be tailored to exploit surrogate functions [20, 59, 10]. Conversely, the MADS algorithm can be used as part of a broader direct search super-algorithm. For example, BiMADS and MultiMADS [25, 26] solve multiobjective optimization problems by running several MADS instances while managing the progress in obtaining a detailed Pareto front. ROBUSTMADS [19] interrupts sequences of MADS runs by redefining the objective function to take into account noisy values. PsDMADS [14] divides a large problem into problems of smaller dimension and launches instances of MADS in a parallel environment.

- **Performance and parallelism.** A major effort was placed into reducing the wall clock time to obtain good solutions. The opportunistic strategy for evaluating points combined with ordering points to promote the most promising, ensure faster convergence [56]. Quadratic models approximate the problem to rapidly find better points [30]. Methods were developed to span a limited number of directions while maintaining the convergence proof, again to limit the number of blackbox evaluations [19]. Conversely, to maximize core utilization during optimization, subspace exploration strategies [14, 4] as well as parallel strategies in NOMAD 3 [45] were developed. However, these strategies are not well adapted to fully utilize the new abundance of computing resources, with some computer clusters counting cores in thousands.

Over the years, it became increasingly difficult to maintain and enhance the functionalities of NOMAD 3. Recent algorithmic developments required modifications in many portions of the software. The complex interactions between algorithms and sub-algorithms were not sufficiently anticipated. It was therefore decided to completely redesign the software.
The main goal of this new version of NOMAD remains to solve efficiently a variety of constrained and unconstrained BBO problems. In NOMAD 4, the MADS algorithm as well as other algorithms deemed useful in NOMAD 3 have been re-implemented using primitive algorithmic components, which are building blocks for more complex algorithms, and interfaces adapted from the experience gained during the development of NOMAD 3. This approach promotes software maintainability, as components may be reused when adding new algorithms. This is an important requisite of this new version of the software even though it forces the rewriting of most of the source code. The requisite to efficiently use a large number of available cores also had a strong impact on the architecture when redesigning the software. Finally, the ability to tune algorithmic parameters (which control the algorithmic components) and to compare different algorithms is also an important requisite of the development. The optimization performance must be maintained between the versions.

This paper describes the design of NOMAD 4 to achieve this goal with the updated requisites. Sections 2 and 3 present the MADS algorithm and other algorithmic components re-implemented from NOMAD 3. The strategies for parallel blackbox evaluations in NOMAD 4 are presented in Section 4. The software architecture and development is presented in Section 5. New algorithmic developments in NOMAD must be assessed in terms of optimization performance and compared with other blackbox optimizers on a large variety of problems: Section 6 compares the performance of the NOMAD 3 and 4 versions and illustrates the gains produced by the use of multiple cores in NOMAD 4. Finally, Section 7 discusses future developments.

2 The MADS algorithm

NOMAD solves optimization problems of the form

$$\min_x \{ f(x) : x \in \Omega \}$$

where \( f : \mathbb{R}^n \to \mathbb{R} \cup \{-\infty\} \) is the objective function and \( \Omega \subseteq \mathbb{R}^n \) is the feasible region. Allowing the objective function to take the value \(-\infty\) is useful to exclude trial points for which the evaluation failed to return valid output, for example, when the blackbox crashes or returns an error message. The original MADS paper [12] handled the constraint set by minimizing the unconstrained extreme barrier function \( f_\Omega : \mathbb{R}^n \to \mathbb{R} \cup \{-\infty\} \) defined as

$$f_\Omega(x) := \begin{cases} f(x) & \text{if } x \in \Omega, \\ \infty & \text{if } x \not\in \Omega. \end{cases}$$

Later, the progressive barrier [13] approach was proposed to exploit the amount by which constraints are violated. The optimization problem is redefined as

$$\min_{x \in X \subseteq \mathbb{R}^n} \{ f(x) : c(x) \leq 0 \}, \quad (1)$$

where \( f : X \subseteq \mathbb{R}^n \to \mathbb{R} \cup \{-\infty\} \) and \( c : X \subseteq \mathbb{R}^n \to (\mathbb{R} \cup \{-\infty\})^m \) are functions with \( c = (c_1, c_2, \ldots, c_m) \), and \( X \) is some subset of \( \mathbb{R}^n \). Again, the entire feasible region is denoted by \( \Omega = \{ x \in X : c(x) \leq 0 \} \). The set \( X \) is frequently taken as being \( \mathbb{R}^n \), the space of continuous variables, or as the set of nonnegative variables \( \mathbb{R}^n_+ \).

Each iteration of the MADS algorithm explores the space of variables through a global exploration called “search step”, and a local exploration called “poll step”. Both these steps generate trial points, which are candidates for evaluation, on a discretization of \( X \) called the mesh. At iteration \( k \), let \( x^k \) denote the current best-known solution. The mesh is defined as \( M^k := \{ x^k + \delta^k y : y \in \mathbb{N}^p \} \subset \mathbb{R}^n \) where \( \delta^k \in \mathbb{R} \) is the mesh size parameter and \( D \in \mathbb{R}^{n \times p} \) is a positive spanning set of \( p \) directions that satisfies specific requirements.
The simplest possible set \( D \) is the union of all positive and negative coordinate directions, and is the one implemented in NOMAD, hence the following redefinition of the mesh at iteration \( k \):

\[
M^k := \{ x^k + \delta^k y : y \in \mathbb{Z}^n \} \subset \mathbb{R}^n.
\]

The search step is flexible, and allows the user to explore any finite number of mesh points in the set named \( S^k \). NOMAD 4 proposes a one-point rudimentary line search in the direction of the previous success [12], a Nelder Mead inspired search step [27] and a search based on the minimization of a quadratic model [30, 6]. Additional search strategies include basic Latin hypercube sampling [48] and others based on advanced statistical surrogates [20, 59, 60]. The Variable Neighbourhood Search search step [52, 9] available in NOMAD 3 will be added in the future. The user may also integrate their own search strategy.

The poll step follows more rigid rules than the search step. Poll points are confined to a so-called frame around \( x^k \) whose dimension is set by a frame size parameter \( \Delta^k \) which is always greater than or equal to the mesh size parameter \( \delta^k \). The key elements of the poll step is that the poll set \( P^k \) must lie within the frame, and \( \{ x - x^k : x \in P^k \} \) must be a positive spanning set for \( \mathbb{R}^n \). For MADS, as \( k \) goes to infinity, the union of these normalized directions becomes dense in the unit sphere. NOMAD 3 includes many examples of poll steps, including coordinate search [31], generalized pattern search [63], ORTHOMADS with \( 2n \) [3] and \( n + 1 \) [18] directions. Of these, NOMAD 4 currently implements ORTHOMADS with \( 2n \) directions; some simple direction strategies are also included; other types of poll steps will be included in the future.

Algorithm 2.1 shows MADS with the extreme barrier to handle constraints. It is close to the one presented in the textbook [17].

**Algorithm 2.1** Mesh adaptive direct search (MADS)

Given \( f : \mathbb{R}^n \to \mathbb{R} \cup \{ \infty \} \), starting point \( x^0 \in \Omega \), and barrier function \( f_{\Omega}(x) \)

0. Initialization

\[
\begin{align*}
\Delta^0 &\in (0, \infty) & \text{initial frame size parameter} \\
\tau &\in (0, 1), \text{ with } \tau \text{ rational} & \text{mesh-size adjustment parameter} \\
\varepsilon_{\text{stop}} &\in [0, \infty) & \text{stopping tolerance} \\
k &\leftarrow 0 & \text{iteration counter}
\end{align*}
\]

1. Parameter Update

\[
\text{set the mesh size parameter to } \delta^k = \min \{ \Delta^k, (\Delta^k)^2 \}
\]

2. Search

\[
\begin{align*}
\text{if } f_{\Omega}(t) < f_{\Omega}(x^k) \text{ for some } t \text{ in a finite subset } S^k \text{ of the mesh } M^k &
\text{ set } x^k+1 \leftarrow t \text{ and } \Delta^{k+1} \leftarrow \tau^{-1} \Delta^k \text{ and go to 4} \\
\text{otherwise go to 3}
\end{align*}
\]

3. Poll

\[
\begin{align*}
\text{let } P^k \text{ be a poll set constructed around } x^k \text{ using a positive spanning set} &
\text{ if } f_{\Omega}(t) < f_{\Omega}(x^k) \text{ for some } t \in P^k &
\text{ set } x^k+1 \leftarrow t \text{ and } \Delta^{k+1} \leftarrow \tau^{-1} \Delta^k \\
\text{otherwise} &
\text{ set } x^k+1 \leftarrow x^k \text{ and } \Delta^{k+1} \leftarrow \tau \Delta^k
\end{align*}
\]

4. Termination

\[
\begin{align*}
\text{if } \Delta^{k+1} \geq \varepsilon_{\text{stop}} &
\text{ increment } k \leftarrow k + 1 \text{ and go to 1} \\
\text{otherwise stop and return } x^* \leftarrow x^{k+1}
\end{align*}
\]
The MADS algorithm with the progressive barrier to handle constraints is slightly more complex. The rules for accepting a new incumbent solution \( x^{k+1} \) are based on both the objective function value and a second function that aggregates the constraint violations. The rules also depend on whether \( x^{k+1} \) is feasible or not. A high-level description is found in Chapter 12 of [17] and the detailed presentation appears in [13].

3 Algorithmic components of MADS

This section describes how the main elements of the MADS algorithm are encoded in NOMAD 4. The pseudo-code from Algorithm 2.1 offers flexibility and may be coded in different ways. The MADS algorithm takes as input an initial point \( x^0 \) which is not required to be within the feasible region, a set of algorithmic parameters \( P \), an objective function \( f \) and some constraint functions \( c \). These functions are provided through an executable code considered as a blackbox. MADS provides the best incumbent solution \( x^* \) found for this problem.

3.1 Terminology

When running an optimization program from given inputs, the code instructions are executed following a specific logic. The present section defines the terminology to describe the execution of MADS.

A task is defined as a group of code instructions adapted for human understanding. In our context, tasks can be generic or specialized, and they can be broken down into smaller tasks. Grouping tasks into algorithmic components allows to visualize the structure of the optimization program and its unfolding. The name and purpose of a component come from the algorithm and the tasks that it performs. The use of generic components and tasks during the design of an optimization program favors maintainability, which is a requisite for the revised version of NOMAD. Hence, the components can be readily reused in different optimization programs, and can be upgraded when new features are introduced.

The program consists of a hierarchy of components, which are executed one at a time, depth-first. The components are named after the algorithms they implement, for example, LH for the Latin hypercube sampling search step, or Mads for the MADS algorithm. The execution of a component unfolds by performing the generic tasks named Start, Run and End, in that order. Specialized tasks are tasks other than the generic tasks, with their name describing their purpose. The Start task may initiate another component, or perform a list of specialized tasks (from top to bottom in the figures below). The End task may call specialized tasks required for the closure of the component. Start and End tasks may also be void. The Run task may initiate another component, or combine specialized tasks with iterating or testing. Nested components and tasks model the execution of the entire program.

3.2 Execution model for algorithms

The execution model is illustrated in Figure 1 with the Latin hypercube sampling algorithm, and in Figure 2 with the more complex MADS algorithm, which requires connecting several nested components and specialized tasks. The LH component presented in Figure 1 sequentially performs four groups of specialized tasks: Generate trial points (called by Start), Insert trials point in evaluation queue (called by Run), Evaluate trial points in queue (called by Run), and Display results (called by End).
LH component can be called during the MADS search step (see Section 2), or as a standalone optimization program. In both cases, all trial points are generated during the Start task of the component.

The LH trial point generation details are found in [61]. Some tasks specialized for the generation and evaluation of trial points are presented in detail as they are common to all blackbox optimization programs. Some specialized tasks are optional, or are only necessary in some cases; they are written within square brackets. For example, trial points are required to be located on the mesh when the LH component is called during the MADS search step. In this case, trial points must be projected on the mesh prior to the actual evaluation. However, when the LH component is called as a standalone algorithm, there is no mesh and therefore no projection is required.

Figure 1: The LH component (Latin hypercube sampling algorithm) with its connected tasks. Tasks within brackets are optional.

When enabled, the cache contains the set of previously evaluated points, and the incumbent point is the best solution found yet. Each trial point is looked up in the cache. If it is not found, then it is added to the evaluation queue of points to be evaluated by the blackbox. Evaluations are run, possibly in parallel when multiple cores are available. If a trial point evaluation is better than the incumbent point evaluation, it is possible to skip the points remaining in the queue and to save the cost of evaluating them: this strategy is called opportunism. There is a direct correspondence between some statements of Algorithm 2.1 and the tasks/sub-tasks. The comparison between $f_\Omega(t)$ and $f_\Omega(x^k)$ seen in the search and poll steps corresponds to the tasks Call blackbox on trial point, Retrieve evaluation, and Check for success.

The Start task of the Mads component executes the Initialization component referring to Step 0 in Algorithm 2.1. As illustrated in the left part of Figure 2, the Run task of the Initialization component first performs the mesh initialization and then conducts the provided initial point evaluation. The task Evaluate $x^0$ in Figure 2 is identical to the previously described tasks Insert trials point in evaluation queue and Evaluate trial points in queue, with the trial point set to the initial point $x^0$. The Run task of the Initialization component concludes with the initialization of the progressive barrier parameters to handle the constraints. The Run task of the Mads component then repeatedly executes Iteration components until a termination criterion is met. Each Iteration component involves respectively an Update, multiple Search, and a Poll components. Each Search component can initiate a sequence of nested algorithmic components. The MADS algorithm offers the flexibility to use any type of algorithm during the search step, as long as a finite number of points is generated, that they are projected on the current mesh, and that the evaluation budget from the set of algorithmic parameters $P$ is not exceeded.
A component can call another instance of itself, directly or indirectly, as long as it is ensured that no infinite recursion is induced. The nested components of the Search component depend on which search step is performed. An example of search step based on quadratic models is described in Section 3.3 and Figure 3. The Poll and Search components perform the tasks Generate trial points (using different strategies), Insert trial points in evaluation queue, Evaluate trial points in queue and Update barrier. While it is deployed, the program alternates generation and evaluation of trial points, which can be a limiting factor to the number of parallel evaluations. Section 4 presents a different way to deploy the execution of MADS to exploit parallel blackbox evaluations.

3.3 Combining algorithmic components

Algorithmic components that represent algorithms can be run standalone. For instance, using NOMAD, it is possible to run the NelderMead algorithm to optimize a problem, and the solution may be compared to the solution found using other algorithms such as MADS. Algorithmic components are building blocks that may be combined and connected together to produce new algorithms. The LH component presented in Figure 1 can be used as a sub-algorithm by the Search component of MADS to generate trial points, as long as the mesh projection is performed. For the same purpose, we developed a QuadraticModelSearch component, in which previous blackbox evaluations are used to construct a quadratic model. This model is used as a surrogate problem and is optimized to provide new trial points; this optimization is performed by a new instance of MADS, with the quadratic model search step disabled to avoid infinite recursion. Therefore, we have a Mads component which, through a QuadraticModelSearch component and the task Quadratic model optimization, starts the execution of another instance of the Mads component. This is illustrated in Figure 3 with a dashed line connecting the two items.

Various components can be used during a search with some control other the evaluation budget or the number of iterations. A SpeculativeSearch component generates trial points by using the direction of last success, which is speculated as a possible direction of improvement, starting from the current incumbent solution. We also developed a NelderMead component based on the Nelder-Mead algorithm to iteratively generate and evaluate trial points. Version 3 of NOMAD has a VariableNeighborhoodSearch...
The Mads component, with all its nested components and tasks, can itself be used within a super-algorithm that does not necessarily rely on a mesh: see for example [46] where NOMAD is hybridized with a mesh-free linesearch method. It may also be executed repeatedly to solve a biobjective optimization problem through a series of single-objective formulations [25]. Version 3 of NOMAD already has the corresponding BiObjective component that will be integrated in NOMAD 4 in the future.

Another example of the Mads component being used as part of a super-algorithm is in ParallelSpace-Decomposition (PSDMADS [14]), where large problems are solved using an asynchronous parallel algorithm in which the parallel processes are launched on subproblems over subsets of variables. A version of PSDMADS is implemented in NOMAD 4 using available algorithmic components. It is described in Section 4.5.

4 Parallel blackbox evaluations

A typical user of NOMAD with access to a specific computational capacity would like to obtain the best possible solution for an optimization problem within a certain time limit. This implies that NOMAD must efficiently exploit all available cores. In some cases, the blackbox evaluation itself runs in parallel, using all cores, but that is not always the case. An assumption for the software development is that running a blackbox evaluation requires significantly more computational time compared to the other algorithmic tasks. Therefore, NOMAD must efficiently distribute the blackbox evaluations in parallel, on secondary threads, while all other tasks are executed on a single main thread. Several strategies for such parallel blackbox evaluations are presented in this section.

4.1 The evaluation queue

Regardless of the strategy used to manage the parallel evaluations, an evaluation queue is maintained to manage evaluations, in which the elements are trial points to be evaluated. It behaves as a priority queue, and is implemented as a sorted vector. When a trial point is generated, it is inserted in the queue, provided
that it was not previously evaluated. The trial points may be ordered, so that the most promising one are evaluated first. Sorting the points is important when opportunism is used, because when the evaluation of a point leads to a new success, the remaining points in the queue are not evaluated. Different ordering strategies are available to sort the trial points in the queue: based on the direction of the last success, on the order in which trial points were generated, on the lexicographic order of their coordinates, or simply mixed randomly.

4.2 Grouping evaluations

NOMAD provides the option to group trial points, with a given maximum group size, for evaluation. With this strategy, users are in charge of managing the dispatching of the groups of points, depending on the specifics of their blackbox and computers, in order to maximize core usage. Nevertheless, grouping points for evaluation is not ideal because there may be an insufficient number of trial points in the queue to fill a group to its maximum size, resulting in unexploited cores. Exploratory work on filling groups of poll sets appears in [44] and will eventually be incorporated into NOMAD.

4.3 Parallel evaluations on multiple threads

NOMAD manages one or more main threads and, if additional cores are available, optional secondary threads. In this Subsection we consider a single main thread. See Section 4.5 for a case where multiple main threads are used. The main thread performs all algorithmic tasks, including some of the evaluations of trial points. The secondary threads only execute evaluation tasks. These tasks, independently of the algorithm, are grouped under the name “Evaluate one trial point in queue”, and are the same as those listed in “Evaluate trial points in queue” in Figure 1. Each thread performs one evaluation task at a time.

Figure 4 illustrates the tasks workflow performed in the main thread and two secondary threads. The specialized tasks for an algorithm are not presented. In the main thread, the iterative process of dequeuing and evaluating points terminates when there are no more points in the queue, or when some criteria is met (budget of evaluations is filled, opportunism condition is met, etc.) At this stage, the evaluation queue is cleared from any remaining unevaluated trial points, the progressive barrier parameters are updated, new trial points are generated, and the algorithm continues. In the meantime, the secondary threads keep on working and the iterative process of dequeuing and evaluating points goes on until the algorithm terminates in the main thread. All information relative to a trial point (success, objective and constraint values) that is evaluated in a secondary thread is made available to the other tasks of the algorithm, and this information is taken into account in the main thread for the continuation of the algorithm.

The management of threads is currently done using OpenMP. The user may provide the number of threads $n_t$ to efficiently access the computer cores; otherwise, OpenMP computes the number of available threads.

4.4 Grouping trial points generation

In every Search and Poll components of the generic MADS algorithm described in Section 2 evaluations are performed immediately after the trial points are generated. This approach generates few points, sometimes a single point, to be evaluated, which makes it difficult to exploit multiple cores for evaluations and to use parallelism at its full capacity. We developed a new combination of Search and Poll components, called MegaSearchPoll (see Figure 5). It generates all the trial points for the Search and Poll components, and only then the points are inserted in the queue and evaluated in parallel. This way, more points are evaluated at a given time. Additionally, search strategies like the speculative search were reworked to
provide more points. Different strategies to enlarge the number of points generated by the poll step are also proposed and examined in [44].

---

**Figure 4:** Workflow of the tasks for the evaluation queue using parallel threads.

---

**Figure 5:** The MegaSearchPoll version of MADS with its nested tasks and algorithmic components. A MegaSearchPoll component is added to the Mads component of Figure 2. The MegaSearchPoll component and its dependant tasks and components are presented in a dashed box. The Run and End tasks of the Search and Poll are not called.
4.5 **PsDMads: Mads with parallel space decomposition**

The sequential implementation of Mads is recommend from problems whose dimension is reasonably small ($n \leq 50$). PsDMads was developed [14] to solve larger problems using space decomposition and parallelism. In PsDMads, the problem is divided into random subspaces of dimension $n_s$, much smaller than $n$. Values of $n_s$ ranging from 2 to 4 are frequently used. Each subproblem is solved by launching a Mads algorithmic component called Worker. An additional Mads algorithmic component called Pollster is launched in the space of dimension $n$, but evaluates a single point generated by a Poll component. In practice, the Pollster rarely improves the incumbent solution but its presence is necessary to ensure that the theoretical convergence results of PsDMads are satisfied. The Pollster and Workers are repeatedly launched within an Iteration component. The mesh sizes of the Pollster and Workers are bounded by a master mesh that is updated at every Iteration. These bounds, the Pollster’s single evaluation, and the Worker’s small dimension, allow for a fast resolution of each Mads. Figure 6 presents the main algorithmic components involved in PsDMads.

![Figure 6: Description of PsDMads using algorithmic components. The Pollster and subproblem Workers are run in parallel. The nested components and tasks of Mads are not presented.](image)

In the original PsDMads implementation described in [14], the management of parallel processes is done using Message Passing Interface (MPI). In the new implementation, OpenMP manages the parallel execution of algorithmic components on main threads. The total number of available threads is $n_t$. The Pollster and Workers are run by Mads components on $n_{mt}$ main threads, with $n_{mt} \leq n_t$ as illustrated by Figure 7. Main thread 0 is used for the Pollster. Main threads 1 to $n_{mt} - 1$ are used for the Workers. Additional threads $n_{mt}$ to $n_t - 1$ are secondary threads.

The main threads are used for algorithmic components and evaluations, whereas the secondary threads are used only for evaluations. The master mesh size update (enlarged or refined) depends on the success of one of the Workers or Pollster. In the new implementation, a finer update control delays the mesh size update until a prescribed minimum number of variables are explored by subproblems solved by Workers.
5 Software architecture and development

This section reviews high level choices made for the different software components of NOMAD. Next, details relative to the processes and tools used while creating this software are mentioned.

5.1 Software architecture

The previous section illustrated how NOMAD 4 and algorithmic components facilitate the construction of new algorithms, such as PSDMADS.

Since this new version is written from scratch, an effort was placed in the development of a modular and reusable architecture. The algorithmic components described in Section 3 are building blocks for algorithms. The evaluation queue presented in Section 4 is designed to launch evaluations in parallel. The code is implemented using object-oriented programming. Algorithmic components and evaluation queue are translated in code as classes and objects. Polymorphism is used: for instance, all algorithmic components are of the base class Step; classes Algorithm and SearchMethod inherit from Step; class SpeculativeSearch derives from SearchMethod. Efforts are made to ensure that the code is clear and generic enough to be easily understood and modified, for example by implementing a new search method.

The code is organized into libraries that provide a range of functionalities for programming algorithms to solve optimization problems:

- **Utils**: Math functionalities; parameter definition and checking; output management, including an output queue to correctly display information coming from different threads; file utilities, clock, and other utilities.

- **Eval**: All that relates to the management of evaluations: Evaluation queue, evaluator, results of evaluations, and cache for points that have already been evaluated.

- **Algos**: Algorithmic components and algorithms: LH, Mads, NelderMead, QuadraticModelSearch, ParallelSpaceDecomposition, and SgтелibModelSearch, where the SgтелibModelSearch is a more general case of the QuadraticModelSearch algorithm.
• sgtelib [58], a library containing many surrogate models developed by B. Talgorn.

5.2 Software development

The development process of NOMAD 4 is inspired by the Agile software development values. The team meets over daily scrums and biweekly group meetings. They work closely with students and business partners. Features and issues are discussed and added timely to the code. Code quality is verified through unit tests, for classes and methods, and through integration tests, for algorithmic functionality. Performance profiles (presented in section 6) are processed regularly, comparing NOMAD 4 with NOMAD 3 or with previous versions of NOMAD 4, to establish that development is going in the right direction to efficiently solve optimization problems.

Customer collaboration is key to development. For instance, one of our key users asked for hot/warm restart. This new feature makes it possible to continue the solving process after it has started, without having to restart it from the beginning. In the case of hot restart, the user interrupts the solver to change the value of a parameter. With warm restart, the user changes a parameter from a resolution that has already reached a termination condition. In both cases, the solving process is then continued from its current state. This feature was discussed with the user, and added to NOMAD 4. The user could test it promptly.

NOMAD 4 is a standalone program coded in C++14 using OpenMP when available, on Linux and macOS. A Windows version will be available soon. CMake is used for compilation. Google Test is used for unit tests. Stable code is available and updated frequently at [github.com/bbopt/nomad](https://github.com/bbopt/nomad).
6 Computational results

The code of NOMAD 4 differs significantly from that of NOMAD 3; only a few base classes were preserved. Comparing the performance of the two versions is crucial to validate that algorithms have been correctly coded.

Tests are conducted using an in-house application called the Runner. The Runner is designed to compare the performance of different optimization software including different versions of NOMAD and various algorithmic choices. The benchmark tests presented below include constrained and unconstrained analytical problems, engineering test problems from the literature, as well as tests involving parallelism.

Comparisons are made through data profiles [53]. The vertical axis shows the proportion of problem solved within a prescribed tolerance of a parameter $\tau$ and the horizontal axis measures the effort deployed by the compared methods in terms of groups of $n+1$ function evaluations. A steep curve indicates that the corresponding method rapidly improves the solutions. A method having its curve above the others performs better for the prevailing test conditions. The optimization runs are conducted on a series of problems for a given evaluation budget. In the tests below, each graph has two curves, one for NOMAD 3 and the other for NOMAD 4 with all default parameters, except that the direction type is set to ORTHO 2N (NOMAD 3 has a default called ORTHO N+1 QUAD direction type that is not yet implemented in NOMAD 4) and the ordering of the points before evaluation is set to the last direction of success (NOMAD 3 has a default ordering based on quadratic models that is not yet implemented in NOMAD 4).

6.1 Validation on analytical problems

Figure 9 compares NOMAD 3 and 4 on a collection of 53 unconstrained smooth problems [53] with a number of variables $n$ ranging from 2 to 12 and with a budget of $400(n+1)$ function evaluations. Each version is launched 10 times with different random seeds. The profiles on the left use a tolerance of $\tau = 10^{-2}$ (the reader is invited to consult [53] for the precise description of the role of $\tau$) and the ones on the right use a smaller tolerance of $\tau = 10^{-4}$. In both cases, the two versions exhibit comparable performance, and there is no clear dominance of one over the other.

Figure 10 compares NOMAD 3 and NOMAD 4 on the collection of 18 constrained problems listed in Table 1 with a budget of $1000(n+1)$ function evaluations. The number of variables ($n$) varies from 2 to 20, the number of constraints ($m$) ranges from 1 to 15, and 13 of the problems have bounds on the variables. Again, each version is launched with 10 random seeds. The profiles on the left use a tolerance of $\tau = 10^{-2}$ and the ones on the right use a smaller tolerance of $\tau = 10^{-4}$. As with the unconstrained case, NOMAD 3 and 4 have a similar performance.

6.2 Tests on PsDMADS

The implementations of PsDMADS in NOMAD 3 and in NOMAD 4 are significantly different, principally because of the strategy used to perform parallel subproblem optimizations. We present a comparison of the two PsDMADS implementations on two bound constrained variants of the Rosenbrock test problem [38]. The two variants, called SRosenbr50 and SRosenbr250, have $n = 50$ and $n = 250$ variables respectively, with all starting point coordinates set to 0.5. The lower bounds are all set to -10 and the upper bounds are set to 10.

Because of the stochastic nature of PsDMADS, both instances of the problem are solved 30 times each to perform a fair comparison. Figure 11 plots the average incumbent objective function value versus the number of function evaluations. The plot also shows the best and worst objective function values.
Figure 9: Data profiles obtained on 53 smooth unconstrained problems. The precision to detect if a problem is solved is set to $\tau = 10^{-2}$ (left) and $\tau = 10^{-4}$ (right).

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Source</th>
<th>n</th>
<th>m</th>
<th>Bnds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CHENWANG_F2</td>
<td>29</td>
<td>8</td>
<td>6</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>CHENWANG_F3</td>
<td>59</td>
<td>10</td>
<td>8</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>CRESCENT</td>
<td>13</td>
<td>10</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>DISK</td>
<td>13</td>
<td>10</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>G210</td>
<td>14</td>
<td>10</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>G220</td>
<td>14</td>
<td>20</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>HS19</td>
<td>42</td>
<td>2</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>HSS3</td>
<td>42</td>
<td>5</td>
<td>6</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>HS114</td>
<td>47</td>
<td>9</td>
<td>6</td>
<td>yes</td>
</tr>
<tr>
<td>10</td>
<td>MAD6</td>
<td>47</td>
<td>5</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td>11</td>
<td>MDO</td>
<td>64</td>
<td>10</td>
<td>10</td>
<td>yes</td>
</tr>
<tr>
<td>12</td>
<td>MEZMONTES</td>
<td>50</td>
<td>2</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>13</td>
<td>OPTENG_RBF</td>
<td>43</td>
<td>3</td>
<td>4</td>
<td>yes</td>
</tr>
<tr>
<td>14</td>
<td>PENTAGON</td>
<td>47</td>
<td>6</td>
<td>15</td>
<td>no</td>
</tr>
<tr>
<td>15</td>
<td>SNAKE</td>
<td>13</td>
<td>2</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>16</td>
<td>SPRING</td>
<td>55</td>
<td>3</td>
<td>4</td>
<td>yes</td>
</tr>
<tr>
<td>17</td>
<td>TAOWANG_F2</td>
<td>63</td>
<td>7</td>
<td>4</td>
<td>yes</td>
</tr>
<tr>
<td>18</td>
<td>ZHAOWANG_F5</td>
<td>65</td>
<td>13</td>
<td>9</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1: Description of the set of 18 analytical problems with constraints.

The two implementations of PsDMADS use 4 workers to perform subproblem optimizations with 2 randomly selected variables among the 60 available. NOMAD 3 uses MPI with 6 processes (1 process for the pollster, 3 processes for the regular workers, 1 process for the cache server and 1 process for the master). To obtain a comparable task distribution, NOMAD 4 uses only $n_t = 4$ main threads for OpenMP (no secondary thread) for the pollster and regular workers (see Figure 7).

For both variants, the PsDMADS version in NOMAD 4 outperforms that of NOMAD 3. The worst performance of the 30 PsDMADS runs using NOMAD 4 is significantly better than the best run of NOMAD 3.
6.3 Improvements in solution times

The previous section compared the performance of PSDMADS in terms of number of function evaluations. We next study the impact of using multiple cores on the overall computational time. The Rosenbrock test problem is not adequate for such comparisons, as it is evaluated nearly instantaneously. We present results on the problem Solar 7 [32], which requires approximately 5 seconds for each evaluation. This problem simulates the operation of a solar thermal power plant. It has 7 variables, one of which is integer, 6 constraints,
and variables are bounded.

Figure 12a shows data profiles, for different parameter settings of NOMAD 4, where the x-axis represents the wall-clock time in seconds rather than the number of function evaluations. A method having its corresponding curve above the others performs faster than the others. Figure 12b illustrates the speed-up, which plots the wall clock time in seconds in terms of the number of function evaluations. Low values on the plot indicate better performance. Three cases were tested on a machine containing 8 cores, by varying \( n_t \), the number of threads used, which here is equal to the number of cores used. In the first case, a single core is used \((n_t = 1)\). In the second case, 8 cores are used \((n_t = 8)\). The third case combines 8 cores with the MegaSearchPoll component described in Section 4.4. Each case is launched ten times with different random seeds.

![Figure 12: Effect of using multiple cores for NOMAD 4, for constrained problem Solar 7, where a single evaluation takes about five seconds.](image)

Without surprise, using multiple cores in parallel allows NOMAD 4 to generate solutions faster. In addition, using the MegaSearchPoll component improves the speed even more. In summary, running NOMAD 4 on Solar 7 using 8 cores as well as the MegaSearchPoll component leads to an overall computational time up to 3.3 times faster than using a single core.

7 Conclusion

The NOMAD blackbox optimization package has been completely redesigned. The new design defines easily interfaceable building blocks named algorithmic components, for constructing elaborate algorithms. This approach promotes software maintainability and modularity. The architecture is strongly impacted by the requirement of efficiently using a large number of cores.

The new version’s numerical performance is comparable to that of the previous version. However, the code is easily accessible for students and developers. The modularity of the algorithmic components makes the code flexible, reusable and allows for easy development of new algorithms. Because the architecture is designed with parallelism in mind, NOMAD is now able to manage advantageously a large number of cores.
The first release of NOMAD 4 will serve as a basis for future developments, starting with the integration of some of the improvements from the last 12 years:

- BiMADS [25] and MultiMADS [26] and DMulti-MADS [28] for multiobjective optimization;
- ROBUSTMADS [19] and StoMADS [16] for robust and stochastic optimization;
- Variable Neighbourhood Search [52, 9] to escape locally optimal solutions;

Another research direction is the application of the software to real industrial and engineering optimization problems to facilitate modeling, solving, analyzing and finding solutions for users. Each application has its specificity, that may result in a new generic feature within NOMAD. Close collaborations with industry users is crucial for the development of NOMAD. Contact us for projects, we look forward working with you.

Acknowledgments: This work is supported by the NSERC CRD RDQPI 490744-15 grant coupled with an InnovÉE grant, both in collaboration with Hydro-Québec and Rio Tinto, and by the NSERC Alliance grant 544900-19 in collaboration with Huawei-Canada.

References


