



# NOMAD user guide version 3.3

Sébastien Le Digabel

November 16, 2009

## HOW TO USE THIS GUIDE:

- **NEW USERS OF NOMAD:** Section [3](#) describes how to install the software. Section [5](#) describes the simplest usage of NOMAD. NOMAD has default values for all of its internal parameters.
- **ADVANCED FEATURES OF NOMAD:** The more experienced users will find in Section [7](#) and above, ways to tailor the output files, and to modify all internal parameters.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Release notes</b>	<b>4</b>
2.1	Version 3.3 . . . . .	4
2.1.1	Minor changes . . . . .	4
2.1.2	List of modified or new classes or methods . . . . .	5
2.2	Version 3.2 . . . . .	6
2.3	Version 3.1 . . . . .	6
<b>3</b>	<b>Installation</b>	<b>6</b>
3.1	Setting environment variables . . . . .	6
3.2	Manual compilation of the code . . . . .	8
3.2.1	Linux / Unix / Mac OS X . . . . .	8
3.2.2	Windows with <code>minGW</code> . . . . .	8
3.2.3	Windows with <code>Visual C++</code> . . . . .	8
3.2.4	Library compilation . . . . .	8
<b>4</b>	<b>Examples</b>	<b>8</b>
<b>5</b>	<b>NOMAD batch mode</b>	<b>9</b>
5.1	Creation of a basic parameters file . . . . .	10
5.2	Basic instructions on black-box programs . . . . .	10
<b>6</b>	<b>NOMAD library mode</b>	<b>15</b>
6.1	Definition of the problem . . . . .	15
6.2	The main function . . . . .	17
6.2.1	Parameters . . . . .	17
6.2.2	Evaluator declaration and algorithm run . . . . .	19
6.2.3	Access to the solution and to optimization data . . . . .	19
6.3	Other functionalities of the library mode . . . . .	19
6.3.1	Automatic calls to user-defined functions . . . . .	19
6.3.2	Create groups of variables . . . . .	20
6.3.3	Multiple runs . . . . .	20
<b>7</b>	<b>Parameters description</b>	<b>22</b>
7.1	Parameters describing the problem . . . . .	22
7.1.1	Basic . . . . .	22
7.1.2	Advanced . . . . .	23
7.2	Algorithmic parameters . . . . .	23
7.2.1	Basic . . . . .	23
7.2.2	Advanced . . . . .	24
7.3	Output parameters . . . . .	25
7.3.1	Basic . . . . .	25
7.3.2	Advanced . . . . .	26
7.4	Additional information for some parameters . . . . .	26
7.4.1	<code>BB_EXE</code> and <code>SGTE_EXE</code> . . . . .	26
7.4.2	<code>BB_INPUT_TYPE</code> . . . . .	27

7.4.3	BB_OUTPUT_TYPE	27
7.4.4	BB_REDIRECTION	27
7.4.5	Bounds	28
7.4.6	Direction types	28
7.4.7	DISPLAY_STATS and STATS_FILE	29
7.4.8	FIXED_VARIABLE	30
7.4.9	Mesh and poll size parameters	30
7.4.10	Opportunistic strategy	30
7.4.11	TMP_DIR	31
7.4.12	VARIABLE_GROUP	31
7.4.13	X0	31
<b>8</b>	<b>Special functionalities</b>	<b>32</b>
8.1	Categorical variables	32
8.1.1	Algorithm	32
8.1.2	Categorical variables with NOMAD	32
8.2	Bi-objective optimization	33
8.3	Variable Neighborhood Search	35
<b>9</b>	<b>Future versions</b>	<b>35</b>
9.1	Algorithm future developments	35
9.2	Future packages	35
	<b>Related publications</b>	<b>39</b>

## 1 Introduction

NOMAD (Nonsmooth Optimization by Mesh Adaptive Direct Search) is a C++ implementation of the Mesh Adaptive Direct Search (MADS) algorithm [6, 18, 20], designed for constrained optimization of black-box functions in the form

$$\min_{x \in \Omega} f(x) \quad (1)$$

where  $\Omega = \{x \in X : c_j(x) \leq 0, j \in J\} \subset \mathbb{R}^n$ ,  $f, c_j : X \rightarrow \mathbb{R} \cup \{\infty\}$  for all  $j \in J = \{1, 2, \dots, m\}$ , and where  $X$  is a subset of  $\mathbb{R}^n$ .

Developers of the method behind NOMAD include

- Mark A. Abramson ([Mark.A.Abramson@boeing.com](mailto:Mark.A.Abramson@boeing.com)), The Boeing Company.
- Charles Audet ([www.gerad.ca/Charles.Audet](http://www.gerad.ca/Charles.Audet)), GERAD and Département de mathématiques et de génie industriel, École Polytechnique de Montréal.
- J.E. Dennis Jr. ([www.caam.rice.edu/~dennis](http://www.caam.rice.edu/~dennis)), Computational and Applied Mathematics Department, Rice University.
- Sébastien Le Digabel ([www.gerad.ca/Sébastien.Le.Digabel](http://www.gerad.ca/Sébastien.Le.Digabel)), GERAD and Département de mathématiques et de génie industriel, École Polytechnique de Montréal.

Version 3.0 (and above) of NOMAD is developed by Sébastien Le Digabel. Previous versions were written by Gilles Couture (GERAD).

NOMAD is designed to be used in two different modes: batch and library. The batch mode is intended for a basic and simple usage of the MADS method, while the library mode allows more flexibility. For example, in batch mode, users must define their separate black-box program, that will be called with system calls by NOMAD. In library mode, users can define their black-box function as C++ code that will be directly called by NOMAD, without system calls and temporary files. This document explains how to get started with the batch mode in Section 5, and with the library mode in Section 6.

A new user of NOMAD can start to use it easily (see Section 5). NOMAD has default values for all of its internal parameters. The more experienced users will find in this document ways to tailor the output files, and ways to modify the internal parameters.

NOMAD should be cited with references [4, 37]. Other relevant papers by the developers are accessible through the NOMAD website [www.gerad.ca/nomad](http://www.gerad.ca/nomad).

The project started in 2001, and was funded in part by AFOSR, CRIAQ, FQRNT, LANL, NSERC, the Boeing Company, and ExxonMobil Upstream Research Company.

## 2 Release notes

### 2.1 Version 3.3

This new version implements the handling of **categorical variables** for mixed variable problems (MVP). This new feature is described in Section 8.1. Below are other minor changes in version 3.3.

#### 2.1.1 Minor changes

- Change of the default criterion according to which the trial points are sorted before they are evaluated. The previous versions used the lexicographic order, and the new version uses the order in which the trial points were generated. Runs obtained with NOMAD 3.3 may differ from runs with older versions. The example run illustrated in Figure 4 page 14 has been modified accordingly.
- When 'NONE' or a negative number is given as argument to the parameter `SEED`, the default random seed is now the process ID instead of the result of the `time()` function.
- Bi-objective optimization in library mode: tests have been added in order to check if the user correctly indicates a `Multi_Obj_Evaluator` object with `Mads::multi_run()`, and to prevent the call to `Mads::run()`.
- New examples 11 to 14. See Section 4. Example 2 (library basic usage) have been augmented with a bi-objective version.
- Installation on X systems, with the `./configure` command: Both the executable and the library are created with the `make` command. Two separated compilations were necessary in previous versions.
- New parameters `EXTENDED_POLL_ENABLED`, `EXTENDED_POLL_TRIGGER`, `HAS_SGTE`, and `USER_CALLS_ENABLED`.

### 2.1.2 List of modified or new classes or methods

- New class `Extended_Poll` for categorical variables. The `Signature` class has also been modified for the categorical variables introduction.
- New method `Directions::get_max_halton_seed()` to access the highest Halton seed that has been used. This is useful to create new variable groups (see the new section [6.3.2](#) explaining how to create variable groups in library mode).
- The virtual function `Evaluator::update_success()` is now more convenient to use. Three additional methods that are automatically called can now be user defined: `Evaluator::list_of_points_preprocessing()`, `Evaluator::update_iteration()`, and `Multi_Obj_Evaluator::update_mads_run()`. See the new Section [6.3.1](#).
- Methods `Evaluator_Control::add_eval_point()` and `Evaluator_Control::eval_list_of_points()` no longer need feasible and infeasible successful directions as arguments.
- Constructor of class `LH_SEARCH` has been simplified.
- New advanced constructor for the `Mads` class, in order to take into account the `Extended_Poll` object necessary for categorical variables.
- The method `Mads::run()` now takes no arguments.
- Two new methods `Mads::enable_user_calls()` and `Mads::disable_user_calls()` to enable or disable the automatic calls to user-defined functions (see Section [6.3.1](#)).
- New method `Parameters::force_check()`. This function should only be called by advances users.
- Two prototypes have been added for the method `Parameters::set_BB_INPUT_TYPE()` (one with a `vector` and one with a `list` as input arguments).
- New method `Parameters::set_EXTERN_SIGNATURE()` to run NOMAD with an extern signature.
- Other new methods in the `Parameters` class:
  - `Parameters::variable_is_fixed()`.
  - `Parameters::set_FIXED_VARIABLE()`.
  - `Parameters::set_FREE_VARIABLE ()`.
  - `Parameters::reset_fixed_variables()`.
  - `Parameters::reset_periodic_variables()`.
  - `Parameters::reset_bounds()`.
  - `Parameters::reset_variable_groups()`.
- New method `Point::empty()`.
- New methods `Point::project_to_mesh()` and `NOMAD::Double::project_to_mesh()` in order to project points or coordinates on a given mesh.

## 2.2 Version 3.2

- **Variable Neighborhood Search (VNS)**: this new search strategy is described in Section 8.3.
- **Installers for Linux / Unix / Mac OS X**, developed by [Quentin Reynaud](#) (GERAD and ISIMA). See Section 3.
- **Help on parameters** included in the executable: the command '`nomad -h keyword`' displays help on the parameters related to `keyword`. Typing only '`nomad -h`' or '`nomad -help`' displays all the available help: a complete description of all parameters. Also, '`nomad -i`' or '`nomad -info`' displays information on the current release, and '`nomad -v`' displays the current version.

## 2.3 Version 3.1

- **Bi-objective optimization**: see Section 8.2.
- **Periodic variables**: if some variable are periodic, this can be indicated now via parameter `PERIODIC_VARIABLE`. Bounds have to be defined for these variables. The MADS algorithm adapted to periodic variables is described in [23].
- **Groups of variables** can be defined with the parameter `VARIABLE_GROUP`. At every MADS poll, different directions will be generated for each group. For example, for a location problem, if groups correspond to spatial objects, these will be moved one at a time.

# 3 Installation

NOMAD is developed in C++ under linux with the gcc compiler (g++), version 4. It also has been tested on Unix, Mac OS X with Xcode (gcc 4), Windows XP with minGW (gcc for Windows), and visual C++ 2008. NOMAD is freely distributed under the GNU General Public License, that can be read in the file `gpl-3.0.txt` provided by the package, or at [www.gnu.org/licenses](http://www.gnu.org/licenses).

There are two ways of installing NOMAD: execute the installation program corresponding to your system, or compile the source code. Three files containing the NOMAD package are available on the [website](#). Download the one adapted to your system (Windows, X systems, or Mac). For Windows and Mac, simply execute the downloaded file, and follow the instructions. We suggest that the user choose an installation directory without space in the name to ease the creation of environment variables. Also choose directories for which you have the adequate writing rights. If not, it has been observed that the installation program for Mac systems may sometimes crash. For X systems, decompress the downloaded zip file where you want to install NOMAD, go to the `$NOMAD_HOME/install` directory, and execute the `./configure` command. A makefile will be created there, and from the same directory, execute '`make`' to create both the NOMAD executable in `$NOMAD_HOME/bin` and the NOMAD library in `$NOMAD_HOME/lib`. Note that this installation procedure can also be applied on Mac OS X, if the gcc compiler is installed (if not, install Xcode). After installation, you should have the directory structure described by Figure 1.

## 3.1 Setting environment variables

The installation programs do not set any environment variables. Defining such variables allows more convenient access to NOMAD. The first variable to be defined should be `$NOMAD_HOME`, whose value is the directory where NOMAD has been installed. This variable is used by the makefiles provided

```
$NOMAD_HOME
|- bin
|- doc
|- examples
|   |- ex01_basic_batch
|   |- ex02_basic_library
|   |- ex03_biobjective
|   |- ex04_user_search
|   |- ex05_multi_start
|   |- ex06_nomad2_interface
|   |- ex07_DLL
|   |- ex08_GAMS
|   |- ex09_CUTEr
|   |- ex10_Matlab
|   |- ex11_Plot
|   |- ex12_Fortran
|   |- ex13_Fortran
|   |- ex14_Categorical
|- install
|- lib
|- src
```

Figure 1: Directory structure of the NOMAD package.

in the examples, and is assumed to be defined in this document. Another environment variable to set is the path variable, where `$NOMAD_HOME/bin` should be added. Then you will just have to type `nomad` to execute NOMAD.

To create your environment variables, on X systems, if your shell is `bash`, add the following lines in the file `.profile` located in your home directory:

```
export NOMAD_HOME=YOUR_NOMAD_DIRECTORY
export PATH=YOUR_NOMAD_DIRECTORY/bin:$PATH
```

In case your shell is `csh` or `tcsh`, add the following lines to the file `.login`:

```
setenv NOMAD_HOME YOUR_NOMAD_DIRECTORY
setenv PATH YOUR_NOMAD_DIRECTORY/bin:$PATH
```

In order for your variables to be active, enter the command `'source .profile'` or `'source .login'`, or simply log out and log in. If you use a different shell, please modify your environment variables accordingly.

On Windows, environment variables are accessible in the **Control Panel|System|Advanced|Environment variables** menu. Please note that environment variables are named differently and `$NOMAD_HOME` corresponds to `%NOMAD_HOME%`. We use mainly `$NOMAD_HOME` in this guide.

## 3.2 Manual compilation of the code

If the installation program failed, you need to compile the source code located in `$NOMAD_HOME/src` in order to generate the NOMAD executable. We assume a basic knowledge of makefiles, which are provided for X and Windows systems. They are named `makefile.GCC_X` and `makefile.GCC_WINDOWS`.

### 3.2.1 Linux / Unix / Mac OS X

Enter the command `'make -f makefile.GCC_X'` from a terminal opened in directory `$NOMAD_HOME/src`. This will create the executable file `nomad` located in `$NOMAD_HOME/bin`. If this command fails, try `'gmake'` instead of `'make'`.

### 3.2.2 Windows with minGW

Same procedure as in 3.2.1 except that `makefile.GCC_WINDOWS` must be used after the `-f` flag. The executable is `%NOMAD_HOME%\bin\nomad.exe`.

### 3.2.3 Windows with Visual C++

Create a new console and empty project. Choose a name for your project (`'project_name'` for example), and create the project in `%NOMAD_HOME%`. Then, add all `.cpp` and `.hpp` source files to the project, and compile in **release** mode. This generates the executable file `%NOMAD_HOME%\project_name\Release\project_name.exe`, which can be copied in `%NOMAD_HOME%\bin` for convenience and to stay consistent with this document.

### 3.2.4 Library compilation

To compile the NOMAD library, the procedure is exactly the same as the one for creating the executable, except that the keyword `lib` must be added to the `make` command. For example, on X systems, enter `'make -f makefile.GCC_X lib'`. With **visual C++**, create an empty static library project, insert all files except `nomad.cpp`, and compile.

## 4 Examples

Examples are located in `$NOMAD_HOME/examples`. Some of them use the batch mode described in Section 5, and some of them use the library mode of Section 6. There are 11 examples:

- Examples 1 to 3 correspond to basic examples describing a simple use of the batch and library modes. We will refer to them later on.
- Example 4 describes how users can code their own search strategy. This example corresponds to a search described in [23]. Other examples on how to design a search strategy can be found in files `$NOMAD_HOME/src/Speculative_Search.*pp`, `LH_Search.*pp`, and `VNS_Search.*pp`. Please note that the MADS theory assumes that trial search points have to be lying on the current mesh. Functions `Point::project_to_mesh()` and `NOMAD::Double::project_to_mesh()` are available to perform these projections.
- Example 5: multistart program that launches multiple instances of MADS. The different starting point as generated following a Latin-Hypercube sample strategy.



- Example 6 is a program allowing the use of NOMAD on a problem originally designed for older versions of the software.
- Example 7 uses a black-box that is coded inside a dynamic library (a Windows DLL).
- Example 8 runs on a black-box that is in fact a GAMS [31] program.
- Example 9 shows how to optimize CUTer [41] test problems.
- Example 10: black-box that is a MATLAB function. In order for this last example to work, the MATLAB MCC compiler has to be installed, which allows the creation of stand-alone executables from MATLAB functions. Instruction files are present in most of the examples directories.
- Example 11 illustrates the `Evaluator::update_success()` virtual function allowing to plot information during the NOMAD execution.
- Example 12: black-box problem coded as a FORTRAN routine linked to the NOMAD library.
- Example 13: a more elaborated example mixing FORTRAN and the NOMAD library. A FORTRAN program is used to define the problem and to run NOMAD.
- Example 14: categorical variables on a simple portfolio selection problem.

## 5 NOMAD batch mode

This section explains how to get started with the NOMAD batch mode, and it gives all the steps to solve a black-box problem. The NOMAD batch mode is launched with one argument that corresponds to the name of a parameters text file, and your black-box problem has to be coded as a stand-alone program. The different steps are:

1. Install NOMAD following the instructions of Section 3.
2. Create a directory for your problem. In this document, we use the notation `$PB_DIR` in order to refer to this directory.
3. Create your problem's black-box, which corresponds to an executable located in `$PB_DIR` (see Section 5.2). This program will output the objective and the constraints.
4. Create a parameters file, for example `$PB_DIR/param.txt`, located in the problem directory (see Section 5.1). This file describes where NOMAD will find your problem and what parameters to use.
5. If the NOMAD executable corresponds to the file `$NOMAD_HOME/bin/nomad`, launch the algorithm with `'$NOMAD_HOME/bin/nomad $PB_DIR/param.txt'`.

At any time, you can type `'nomad -h param_name'` to have information on a specific parameter, as described in Section 2.2. Advanced usage of NOMAD was not described in this section. However, all parameters are described in Section 7, and advanced examples are given in `$NOMAD_HOME/examples`.

## 5.1 Creation of a basic parameters file

The parameters file is a text file given as argument to the NOMAD executable with the command '`$NOMAD_HOME/bin/nomad $PB_DIR/param.txt`', where `param.txt` is the parameters file (which has to be located in the problem directory), and `nomad` is the NOMAD executable.

For basic usage, the following parameters have to be defined:

- The number of variables,  $n$  (`DIMENSION`).
- The name of the black-box executable that outputs the objective and the constraints (`BB_EXE`).
- The output types of the black-box executable: objective and constraints (`BB_OUTPUT_TYPE`).
- A starting point (`X0`).
- Some stopping criteria (`MAX_BB_EVAL`, for example).

Bounds on variables are defined with the `LOWER_BOUND` and `UPPER_BOUND` parameters. If no stopping criterion is specified, the algorithm will stop as soon as the mesh size reaches a certain epsilon.

An example is given in Figure 2 that corresponds to the parameters file located in `$NOMAD_HOME/examples/ex01_basic_batch`. Note that all the entries of a line are ignored after the character '#'. The order in which the parameters appear, or their case, is unimportant.

The two constraints defined in the parameters file of Figure 2 are of different type. The first constraint  $c_1(x) \leq 0$  is treated by the progressive barrier approach (PB), which allows constraint violations. The second constraint,  $c_2(x) \leq 0$ , is treated by the extreme barrier approach (EB) that forbids violations.

See Section 7 for the detailed description of all parameters.

## 5.2 Basic instructions on black-box programs

With the batch use of NOMAD, the black-box defining your problem corresponds to a program that will be system-called by the algorithm. It can be coded in any language (even scripts), but has to respect certain conditions. It has to be callable in batch mode as follows: If the black-box executable is `$PB_DIR/bb.exe`, one can execute it with the command '`$PB_DIR/bb.exe x.txt`'. Here `x.txt` is a text file containing a total of  $n=\text{DIMENSION}$  values consisting of one value for each variable, separated by spaces.

The problem directory, where the parameters file is located, can have spaces in its name. The black-box executable can be located in sub-directories of the problem directory, but the names of the sub-directories must be spacing-free.

The black-box program returns the evaluation values by displaying them in the standard output. It also returns the value 0 to indicate that the evaluation went well (a simple '`return 0`' instruction in C). The number of values displayed by the black-box program corresponds to the number of constraints plus one representing the objective function value that one seeks to minimize. The constraints values correspond to constraints of the form  $c_j \leq 0$  (for example, the constraint  $0 \leq x_1 + x_2 \leq 10$  will have to be displayed with the two quantities  $c_1(x) = -x_1 - x_2$  and  $c_2(x) = x_1 + x_2 - 10$ ). The order of the displayed outputs corresponds to the order defined in the parameters file with parameters `BB_EXE` and `BB_OUTPUT_TYPE`. If variables have bound constraints, these are defined in the parameters file with parameters `LOWER_BOUND` and `UPPER_BOUND`. Bounds should not appear in the black-box code.

In basic mode, your black-box program cannot display other data than the objective and constraint values, but the advanced mode allows it to do so. Your code can generate temporary files, but it is preferable to include tag numbers to the file names, because future NOMAD versions will include parallelism. The advanced parameters described in Section 7.2.2 allow you to include these tags in the black-box input files. If you already have a black-box program in a certain format, you need to interface it with a wrapper program in order to match the NOMAD specifications. If your black-box program crashes in batch mode, it will not affect NOMAD: The argument that caused this crash will simply be tagged as a black-box failure.

A basic C++ program example is given in Figure 3 for the following problem with 5 variables and 2 constraints:

$$\begin{aligned} & \min_{x \in \mathbb{R}^5} f(x) = x_5 \\ & \text{subject to} \quad \left\{ \begin{array}{ll} c_1(x) = \sum_{i=1}^5 (x_i - 1)^2 - 25 & \leq 0 \\ c_2(x) = 25 - \sum_{i=1}^5 (x_i + 1)^2 & \leq 0 \\ & x_i \geq -6 \quad i = 1, 2, \dots, 5 \\ & x_1 \leq 5 \\ & x_2 \leq 6 \\ & x_3 \leq 7 . \end{array} \right. \end{aligned}$$

With `gcc`, you can compile this example with `'g++ -o bb.exe bb.cpp'`, and test it with the text file `x.txt` containing `'0 0 0 0 0'`, by entering the command `'bb.exe x.txt'`. This should display `'0 -20 20'`, which means that the point  $x = (0 \ 0 \ 0 \ 0 \ 0)^T$  has an objective value of  $f(x) = 0$ , but is not feasible, since the second constraint is violated ( $c_2(x) = 20 > 0$ ).

DIMENSION	5	# number of variables
BB_EXE	bb.exe	# 'bb.exe' is a program that
BB_OUTPUT_TYPE	OBJ PB EB	# takes in argument the name of
		# a text file containing 5
		# values, and that displays 3
		# values that correspond to the
		# objective function value (OBJ),
		# and two constraints values g1
		# and g2 with form $g1 \leq 0$ and
		# $g2 \leq 0$ ; 'PB' and 'EB'
		# correspond to constraints that
		# are treated by the Progressive
		# and Extreme Barrier approaches
		# (all constraint handling
		# options are described in the
		# detailed parameters list)
X0	( 0 0 0 0 0 )	# starting point
LOWER_BOUND	* -6	# all variables are $\geq -6$
UPPER_BOUND	( 5 6 7 - - )	# $x_1 \leq 5$ , $x_2 \leq 6$ , $x_3 \leq 7$
		# $x_4$ and $x_5$ have no bounds
MAX_BB_EVAL	100	# the algorithm terminates when
		# 100 black-box evaluations have
		# been made
TMP_DIR	/tmp	# indicates a directory where
		# temporary files are put
		# (increases performance by ~100%
		# if you're working on a network
		# account and if TMP_DIR is on a
		# local disk).

Figure 2: Example of a basic parameters file. All parameters are detailed in Section 7 and via the command 'nomad -h param\_name'.

```

#include <cmath>
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main ( int argc , char ** argv ) {

    double f = 1e20, c1 = 1e20 , c2 = 1e20;
    double x[5];

    if ( argc >= 2 ) {
        c1 = 0.0 , c2 = 0.0;
        ifstream in ( argv[1] );
        for ( int i = 0 ; i < 5 ; i++ ) {
            in >> x[i];
            c1 += pow ( x[i]-1 , 2 );
            c2 += pow ( x[i]+1 , 2 );
        }
        f = x[4];
        if ( in.fail() )
            f = c1 = c2 = 1e20;
        else {
            c1 = c1 - 25;
            c2 = 25 - c2;
        }
        in.close();
    }
    cout << f << " " << c1 << " " << c2 << endl;
    return 0;
}

```

Figure 3: Example of a basic black-box program. This code corresponds to the file `bb.cpp` in `$NOMAD_HOME/examples/ex01_basic_batch`.

```

NOMAD - Nonsmooth Optimization by Mesh Adaptive Direct search - version 3.3.0
      www.gerad.ca/nomad

Copyright (C) 2001-2008
      Mark A. Abramson      - The Boeing Company
      Charles Audet        - Ecole Polytechnique de Montreal
      Gilles Couture       - Ecole Polytechnique de Montreal
      John E. Dennis, Jr.  - Rice University
      Sebastien Le Digabel - Ecole Polytechnique de Montreal

funded in part by AFOSR and Exxon Mobil

GNU General Public License: '$NOMAD_HOME/src/gpl-3.0.txt'
      user guide: '$NOMAD_HOME/doc/user_guide.pdf'
      examples: '$NOMAD_HOME/doc/examples'

please report bugs at nomad@gerad.ca

begin of Mads::run()

      stats:
      BBE      OBJ

      3         0
      12        -1
      34        -2
      100       -2

end of Mads::run(), max number of black-box evaluations reached

black-box evaluations: 100
best infeasible point: (4.4 1.2 0 -1 -2) h=0.6 f=-2
best feasible point   : (4.4 1.2 1.3 -1 -2) h=0 f=-2

```

Figure 4: Output given by NOMAD on the black-box problem coded in Figure 3, with parameters file of Figure 2.

NOMAD is flexible enough that black-box codes can be coded differently and with more sophistication in the advanced mode (see Section 6).

Figure 4 shows the display that the execution of NOMAD produces for the black-box program of Figure 3 with the parameters file of Figure 2. Notice that the first feasible point has been found after 8 black-box evaluations. In this case, the starting point  $x = (0 \ 0 \ 0 \ 0 \ 0)^T$  violates the second constraint, which is treated by the extreme barrier approach. In such a situation, NOMAD launches a phase one step, during which the value of the constraint violation is minimized. Once a feasible point is generated with this phase one, the original objective function is considered again.

## 6 NOMAD library mode

This section explains how to create a C++ program able to call the NOMAD routines, using the pre-compiled NOMAD static library. We suppose that the library has been correctly installed and that the environment variable `$NOMAD_HOME` has been defined. If not, you will have to enter manually the installation directory of NOMAD in the makefile. Explanations are given for `linux` and `g++`, but are similar for Windows and `minGW` or `visual C++`. A basic knowledge of object oriented programming with C++ is assumed.

The use of the standard C++ types for reals and vectors is of course allowed within your code, but it is suggested that you use the NOMAD types as much as possible. For reals, NOMAD uses the class `NOMAD::Double`, and for vectors, the class `Point`. A lot of functionalities have been coded for these classes, which are visible in files `Double.hpp` and `Point.hpp`. All the NOMAD class files are named like the classes and are located in the directory `$NOMAD_HOME/src`. Other NOMAD types (essentially enumeration types) are also defined in `defines.hpp`. Some utility functions on these types can be found in `utils.hpp`.

The example shown in this section corresponds to files located in the directory `$NOMAD_HOME/examples/ex02_basic_library`. It is identical to the example shown in Section 5, except that no temporary files are used, and no system calls are made. For this example, just one C++ source file is used, but there could be a lot more. Other examples can be found in `$NOMAD_HOME/examples` and in the main function of NOMAD (file `$NOMAD_HOME/src/nomad.cpp`). This file implements the NOMAD batch mode, and it could have been compiled in library mode. This illustrates the fact that even in library mode a parameters file can be used and system calls performed.

As a first task, a makefile has to be created in the directory where your source code is located. An example of such a makefile is shown on Figure 5. Notice that each line after `':'` has to begin with a tabulation. Two such makefiles are given in `$NOMAD_HOME/examples/ex02_basic_library`. You can rename one of those to `makefile` in order to use the `make` command without the `-f` flag.

We now describe the other steps for the creation of the source file `basic_lib.cpp`, which includes the header file `nomad.hpp`, and which is divided into two parts: a class for the description of the problem, and the main function. Once compiled with the makefile (type `make`), the binary file `basic_lib` is created and can be executed.

### 6.1 Definition of the problem

Describing the black-box problem directly in the code that calls NOMAD avoids the use of temporary files and system calls by the algorithm. This is achieved by defining a derived class `My_Evaluator` that inherits from the class `Evaluator` in single-objective optimization and from `Multi_Obj_Evaluator` in multi-objective mode (see header files `Evaluator.hpp` and `Multi_Obj_Evaluator.hpp`). An example of such a class is shown in Figure 7.

The objective of this user class is to redefine the virtual method `eval_x()` that will be automatically called by the algorithm. The prototype of `eval_x()` is given in Figure 6. Note that the non-const version of the method is also available.

The argument `x` (in/out) corresponds to an evaluation point, i.e. a vector containing the coordinates of the point to be evaluated, and also the result of the evaluation. The coordinates are accessed with the operator `[]` (`x[0]` for the first coordinate), and outputs are set by the method `set_bb_output` (`x.set_bb_output[0]` to set the objective function value, if the objective has been defined to be the first output). Constraints have to be represented by values  $c_j$  for a constraint  $c_j \leq 0$ . Please refer to files `Eval_Point.hpp` and `Point.hpp` for details about the classes defining NOMAD vectors.

```

EXE      = basic_lib
COMPILATOR = g++
OPTIONS  = -ansi -Wall -O3 -DGCC_X
L1       = $(NOMAD_HOME)/lib/nomad.a
LIBS     = $(L1) -lc -lm
INCLUDE  = -I$(NOMAD_HOME)/src -I.
COMPILE  = $(COMPILATOR) $(OPTIONS) $(INCLUDE) -c
OBJS     = basic_lib.o

$(EXE): $(OBJS)
        $(COMPILATOR) -o $(EXE) $(OBJS) $(LIBS) $(OPTIONS)

basic_lib.o: basic_lib.cpp $(L1)
        $(COMPILE) basic_lib.cpp

clean:
        @echo "    cleaning obj files"
        @rm -f $(OBJS)

```

Figure 5: Example of a makefile for a single C++ file linked with the NOMAD library.

```

bool eval_x ( Eval_Point      & x          ,
              const NOMAD::Double & h_max    ,
              bool             & count_eval ) const;

```

Figure 6: Prototype of method `Evaluator::eval_x()`.

The second argument, the real `h_max` (in), corresponds to the current value of the barrier  $h_{max}$  parameter. It is not used in this example, but it can be used in order to interrupt an expensive evaluation, if the constraint violation value  $h$  can be more quickly seen to be larger than  $h_{max}$ . See [20] for the definition of  $h$  and  $h_{max}$  and of the progressive barrier method for handling constraints.

The third argument, `count_eval` (out), needs to be set to `true` if the evaluation counts as a black-box evaluation, and `false` otherwise (for example, if the user interrupts an evaluation with the  $h_{max}$  criterion before it costs some expensive computations, then set `count_eval` to `false`).

If a surrogate function is to be used, then its evaluation routine should be coded in the method `eval_x()`. First, in order to indicate that a surrogate can be computed, the user has to set the parameter `HAS_SGTE` to `yes`, via the method `Parameters::set_HAS_SGTE()`. Then, in `eval_x()`, the test `'if (x.get_eval_type()==_SGTE_)`' must be made in order to differentiate an evaluation with the true function  $f$  or with the surrogate.

Another possibility for the designer of `eval_x()` is the ability to define a priority for the trial point  $x$ , via the method `Eval_Point::set_user_eval_priority()`. Points with higher priorities will be evaluated first. For more details, see the code in `Priority_Eval_Point.cpp`.

Finally, `eval_x()` should return `true` if the evaluation succeeded, and `false` if the evaluation failed.

Of course, more elaborated `Evaluator` subclasses can be designed in order to consider some additional problem-related parameters. Such an example can be found in `$NOMAD_HOME/src/Multi_Obj-`



```

class My_Evaluator : public Evaluator {
public:
    My_Evaluator ( const Parameters & p ) :
        Evaluator ( p , cout ) {}

    ~My_Evaluator ( void ) {}

    bool eval_x ( Eval_Point          & x          ,
                  const NOMAD::Double & h_max      ,
                  bool                & count_eval ) const {
        NOMAD::Double c1 = 0.0 , c2 = 0.0;
        for ( int i = 0 ; i < 5 ; i++ ) {
            c1 += (x[i]-1).pow2();
            c2 += (x[i]+1).pow2();
        }
        x.set_bb_output ( 0 , x[4] ); // objective value
        x.set_bb_output ( 1 , c1-25 ); // constraint 1
        x.set_bb_output ( 2 , 25-c2 ); // constraint 2

        count_eval = true; // count a black-box evaluation
        return true;       // the evaluation succeeded
    }
};

```

Figure 7: Example of a user class defining a hard-coded black-box problem.

`Evaluator.*pp`, where some weights are defined to change the objective function of the problem between successive optimizations (this example correspond to the BiMADS algorithm [25]).

The virtual method `update_success()` can also be redefined in subclasses deriving from `Evaluator`. This method will be automatically invoked every time a new improvement is made. Note that the automatic calls to this method can be enabled/disabled with `Evaluator_Control::set_call_user_update_success()`.

Another virtual method defined in the class `Evaluator` is `compute_f()`. This method allows the user to compute the value of the objective function directly from the black-box outputs. This is used by the BiMADS algorithm. If `compute_f()` is not user-defined, then NOMAD simply takes the value of  $f$  as the first OBJ output from the black-box.

## 6.2 The main function

Once your problem has been defined, the main function can be written. NOMAD routines can throw C++ exceptions, so it is recommended that you put your code into a `try` block.

### 6.2.1 Parameters

First, a `Parameters` object has to be declared. Parameters are defined similarly as in batch mode: each parameter `PNAME` is set with the method `set.PNAME()` of the class `Parameters`. In order to see all the options, use the help `'nomad -h param_name'`, or refer to the detailed list of parameters in

Section 7, or to the header file `Parameters.hpp`. NOMAD additional C++ types necessary for the calls to `Parameters`'s functions can be found in file `defines.hpp`. An example is given in Figure 8. This example is taken from file `basic_lib.cpp` located in `$NOMAD_HOME/examples/ex02_basic_library` and corresponds to the same parameters file example shown in Figure 2, except that no problem executable is used.

No parameters file is needed, but it is possible to take the parameters from such a file, with `Parameters::read("param.txt")` where `param.txt` is a valid parameters file. If a directory path is included in the name of the file, this path will be considered as the problem's path instead of the default location `'./'`. To display the parameters described by a `Parameters` object `p`, use the instruction `'cout << p << endl;'`.

Finally, once that all parameters are set, the method `Parameters::check()` must be invoked in order to validate the parameters. The algorithm will not run with a non-checked `Parameters` object. It is not even possible to access `Parameters` data while not checked. If parameters are changed, `check()` must be invoked again before a new run can be conducted.

```
// parameters creation:
Parameters p ( cout );

p.set_DIMENSION (5);           // number of variables

vector<bb_output_type> bbot (3); // definition of
bbot[0] = _OBJ_;                // output types
bbot[1] = _PB_;
bbot[2] = _EB_;
p.set_BB_OUTPUT_TYPE ( bbot );

p.set_X0 ( Point ( 5 , 0.0 ) ); // starting point

p.set_LOWER_BOUND ( Point ( 5 , -6.0 ) ); // all var. >= -6
Point ub ( 5 );                // x_4 and x_5 have no bounds
ub[0] = 5.0;                   // x_1 <= 5
ub[1] = 6.0;                   // x_2 <= 6
ub[2] = 7.0;                   // x_3 <= 7
p.set_UPPER_BOUND ( ub );

p.set_MAX_BB_EVAL (100);       // the algorithm terminates
                                // after 100 bb evaluations

p.set_TMP_DIR ("/tmp");        // repertory for
                                // temporary files

// parameters validation:
p.check();
```

Figure 8: Example of parameters creation in library mode.

### 6.2.2 Evaluator declaration and algorithm run

The MADS algorithm is implemented with the `Mads` class. Objects of this class are created with a `Parameters` object and an `Evaluator` object. In the example described here, the `Evaluator` object corresponds to an object of type `My_Evaluator`. A NULL pointer can also be used instead of the `Evaluator` object: in this case, the default evaluator will be used. Assuming that the parameter `BB_EXE` has been defined, this default evaluator consists in evaluating the objective function via a separated black-box program and system calls. When an `Evaluator` object is used, parameters `BB_EXE` and `SGTE_EXE` are ignored. A more advanced `Mads` constructor with user-created caches is also available in `$NOMAD_HOME/src/Mads.hpp`.

Once that the `Mads` object is declared, run the algorithm with `Mads::run()` (or `Mads::multi_run()` for multi-objective optimization). An example is shown in Figure 9.

```
// custom evaluator creation:
My_Evaluator ev ( p );

// algorithm creation and execution:
Mads mads ( p , &ev , cout );
mads.run();
```

Figure 9: Evaluator and Mads objects usage.

It is also possible for the user to redefine the virtual method `Evaluator::list_of_points_preprocessing()` in order to indicate a preprocessing strategy that will be applied by the algorithm before each series of evaluations is made. All points can then be modified according to this strategy. See `$NOMAD_HOME/src/Evaluator.hpp` for the header of this method.

### 6.2.3 Access to the solution and to optimization data

In the example of `$NOMAD_HOME/examples/ex02_basic_library`, final information is displayed via a call to the operator `<<` at the end of `Mads::run()`. More specialized access to solution and optimization data is allowed. To access the best feasible and infeasible points, use the methods `Mads::get_best_feasible()` and `Mads::get_best_infeasible()`. To access optimization data or statistics, call the method `Mads::get_stats()` which returns access to a `Stats` object. Then, use the access methods defined in `Stats.hpp`. For example, to display the number of black-box evaluations, write:

```
cout << "bb eval = " << mads.get_stats().get_bb_eval() << endl;
```

## 6.3 Other functionalities of the library mode

### 6.3.1 Automatic calls to user-defined functions

Virtual methods are automatically invoked by NOMAD at some special events of the algorithm. These methods are left empty by default and the user can redefine them so that its own code can be automatically called. These virtual methods are defined in the `Evaluator` and `Multi_Obj_Evaluator` classes and are detailed below:

- `Evaluator::list_of_points_preprocessing()`: Called before the evaluation of a list of points (it allows the user to pre-process the points to be evaluated).

- `Evaluator::update_iteration()`: Called every time a MADS iteration is terminated.
- `Evaluator::update_success()`: Invoked when a new incumbent is found (single-objective) or when a new Pareto point is found (bi-objective).
- `Multi_Obj_Evaluator::update_mads_run()`: For bi-objective problems, this method is called every time a single MADS run is terminated.

It is possible to disable the automatic calls to these methods, with the functions `Mads::enable_user_calls()` and `Mads::disable_user_calls()`, or with the parameters `USER_CALLS_ENABLED` and `EXTENDED_POLL_ENABLED`. These parameters are automatically set to `yes`, except during the extended poll and the VNS search.

### 6.3.2 Create groups of variables

This section gives some explanations about creating groups of variables in library mode. See Section 7.4.12 for defining such group in batch mode. Groups of variable are created with the method `Parameters::set_VARIABLE_GROUP()` which has two different prototypes. The method has to be called each time a new group is created. For both versions of the function, the user has to give a list of the indexes of the variables composing the group. In NOMAD, a group of variable generates its own polling directions. The most complete prototype of `set_VARIABLE_GROUP()` allows to choose the types of these directions, for the primary and secondary polls. The detailed types of directions can be found in file `defines.hpp` and the enum type `direction_type`. The simplified prototype uses `ORTHOMADS` types of directions by default. In all cases a Halton seed must be provided, which is not considered if direction types do not correspond to `ORTHOMADS`. Otherwise, a value must be provided. This value should be larger than the  $n$ th prime number, and ideally be different for each group of variables. The method `Directions::get_max_halton_seed()` is available in order to get the highest Halton seed that has been used, and help determine such a value. Finally the function `Parameters::reset_variable_groups()` can be called to reset the groups of variables. Remember also that after a modification to a `Parameters` object is made, the method `Parameters::check()` has to be called.

### 6.3.3 Multiple runs

The method `Mads::run()` can be invoked more than once, for multiple runs of the MADS algorithm.

A first solution for doing that is simply to declare the `Mads` object, as in Figure 10. But, in this case, the cache, containing all points from the first run, will be erased between the runs (since its it created and deleted with `Mads` objects).

A better solution consists in using the `Mads::reset()` method between the two runs and to keep the `Mads` object in a more global scope. The method takes two boolean arguments (set to `false` by default), `keep_barriers` and `keep_stats`, indicating if the barriers (true and surrogate) and statistics have to be reseted between the two runs. An example is shown in Figure 11.

An example showing multiple MADS runs is described by the files located in `$NOMAD_HOME/examples/ex05_multi_start`.

```
{
  Mads mads ( p , &ev , cout );

  // run #1:
  mads.run();
}
{
  Mads mads ( p , &ev , cout );

  // run #2:
  mads.run();
}
```

Figure 10: Two runs of MADS with a Mads object at local scope. The cache is erased between the two runs.

```
Mads mads ( p , &ev , cout );

// run #1:
mads.run();

mads.reset();

// run #2:
mads.run();
```

Figure 11: Two runs of MADS with a Mads object at a more global scope. The cache is kept between the two runs.

## 7 Parameters description

Parameters described in this section correspond to those that can be entered in a parameters file that the batch mode will load. The same parameters description can be found by entering the command `'nomad -h'`, to see all the parameters, or `'nomad -h param_name'` for a particular parameter.

In library mode, parameters are defined via a `Parameters` object and methods `Parameters::set_PARAM_NAME()`, where `PARAM_NAME` is the name used in this section. It is also possible to read a parameters file in library mode, with the method `Parameters::read()`. In batch mode, the problem directory is automatically determined by `NOMAD`. It can be defined in library mode with `Parameters::set_PROBLEM_DIR()`.

All the entries of a line are ignored after the character `'#'`. Except for the file names, all strings and parameter names are case insensitive (`'DIMENSION 2'` is the same as `'Dimension 2'`). File names refer to files in the problem directory. To indicate a file name containing spaces, use quotes (`"name"` or `'name'`). These names can include directory information, that has to be relative to the problem directory. The problem directory will be added to the names, unless the `'$'` character is used in front of the names. For example, if a black-box executable is run by the command `'python script.py'`, define parameter `BB_EXE` with argument `'$python script.py'`.

Some parameters require the entry of variable indexes. These indexes begin at 0, and can be entered directly or as index ranges, with format `'i-j'`. Character `'*'` can be used to replace `'0-n-1'` (where  $n$  is the number of variables).

Other parameters require arguments of type `bool`: these values can be entered with the strings `yes`, `no`, `y`, `n`, `0`, or `1`.

Finally, some parameters need vectors as arguments. Use `(v1 v2 ... vn)` for those. Characters `'-'`, `'inf'`, `'-inf'` or `'+inf'` are accepted to enter undefined real values. The following subsections show tables describing all `NOMAD` parameters. Parameters are classified into different classes (problem, algorithm and output parameters). For each of these classes, basic and advanced parameters are described separately.

### 7.1 Parameters describing the problem

#### 7.1.1 Basic

name	arguments	description	default
<code>BB_EXE</code>	list of strings; see <a href="#">7.4.1</a>	black-box executables (required in batch mode)	none
<code>BB_INPUT_TYPE</code>	see <a href="#">7.4.2</a>	black-box input types	<code>* R</code>
<code>BB_OUTPUT_TYPE</code>	see <a href="#">7.4.3</a>	black-box output types (required)	none
<code>DIMENSION</code>	integer	$n$ the number of variables (required)	none
<code>LOWER_BOUND</code>	see <a href="#">7.4.5</a>	lower bounds	none
<code>UPPER_BOUND</code>	see <a href="#">7.4.5</a>	upper bounds	none

### 7.1.2 Advanced

name	arguments	description	default
FIXED_VARIABLE	see 7.4.8	fixed variables	none
PERIODIC_VARIABLE	index range	define variables in the range to be periodic (bounds required)	none
SGTE_COST	integer $c$	the cost of $c$ surrogate evaluations is equivalent to the cost of one black-box evaluation	$\infty$
SGTE_EVAL_SORT	bool	if surrogates are used to sort list of trial points	yes
SGTE_EXE	list of strings; see 7.4.1	surrogate executables	none
VARIABLE_GROUP	index range	defines a group of variables; see 7.4.12	none

Surrogates, or surrogate functions, are cheaper black-box functions that are used, at least partially, instead of the true function  $f$  to minimize. If such functions are defined by the user, NOMAD will use them to drive its search. See [30] for a survey on surrogate optimization.

## 7.2 Algorithmic parameters

### 7.2.1 Basic

name	arguments	description	default
DIRECTION_TYPE	see 7.4.6	type of directions for the poll	ORTHO
F_TARGET	reals, $f$ or ( $f1$ $f2$ )	NOMAD terminates if $f_i(x_k) \leq f_i$ for all objective functions	none
HALTON_SEED	integer	Halton seed for ORTHO-MADS [6]	$n$ th prime number
INITIAL_MESH_SIZE	see 7.4.9	$\Delta_0^m$ [18]	r0.1 or 1.0
LH_SEARCH	2 integers: $p0$ and $pi$	LH (Latin-Hypercube) search ( $p0$ : initial, $pi$ : iterative); see 8.2 for bi-objective	none
MAX_BB_EVAL	integer	maximum number of black-box evaluations; see 8.2 for bi-objective	none
MAX_TIME	integer	maximum wall-clock time (in seconds)	none
MULTI_NB_MADS_RUNS	integer	number of MADS runs	see 8.2
MULTI_OVERALL_BB_EVAL	integer	max number of black-box evaluations for all MADS runs	see 8.2
OPPORTUNISTIC_EVAL	bool	opportunistic strategy; see 7.4.10	yes
OPPORTUNISTIC_LH	bool	opportunistic strategy for LH search; see 8.2 for bi-objective	see 7.4.10
SEED	integer or NONE	random seed; NONE or a negative integer to define a seed that will be different at each run	0
TMP_DIR	string	temporary directory for black-box i/o files; see 7.4.11	problem directory
VNS_SEARCH	bool or real	VNS search; see 8.3	no
XO	see 7.4.13	starting point(s)	best point from a cache file or from an initial LH search

## 7.2.2 Advanced

name	arguments	description	default
BB_INPUT_INCLUDE_SEED	bool	if the random seed is put as the first entry in black-box input files	no
BB_INPUT_INCLUDE_TAG	bool	if the tag of a point is put as an entry in black-box input files	no
BB_REDIRECTION	bool	if NOMAD manages the creation of black-box output files; see 7.4.4	yes
EPSILON	real	precision on reals	1E-13
EXTENDED_POLL_ENABLED	bool	if no, the extended poll for categorical variables is disabled	yes
EXTENDED_POLL_TRIGGER	real	trigger for categorical variables; value can be relative; see 8.1	r0.1
H_MAX_0	real	initial value of $h_{max}$ (will be eventually decreased throughout the algorithm)	1E+20
H_MIN	real $v$	$x$ is feasible if $h(x) \geq v$	0.0
H_NORM	norm type in {L1, L2, Linf }	norm used to compute $h$	L2
HAS_SGTE	bool	indicates if the problem has a surrogate (only necessary in library mode)	no or yes if SGTE_EXE is defined
INITIAL_MESH_INDEX	integer	initial mesh index $\ell_0$ [6]	0
L_CURVE_TARGET	real	NOMAD terminates if it detects that the objective can not reach this value	none
MAX_CACHE_MEMORY	integer	NOMAD terminates if the cache reaches this memory limit expressed in MB	none
MAX_EVAL	integer	max number of evaluations (includes cache-hits and black-box evaluations, does not include surrogate eval)	none
MAX_ITERATIONS	integer	max number of MADS iterations	none
MAX_MESH_INDEX	integer	max mesh index $\ell_{max}$ [6]	none
MAX_SGTE_EVAL	integer	max number of surrogate evaluations	none
MAX_SIM_BB_EVAL	integer	max number of simulated black-box evaluations (includes initial cache hits)	none



name	arguments	description	default
MESH.COARSENING_EXPONENT	integer	$w^+$ [18]	1
MESH.REFINING_EXPONENT	integer	$w^-$ [18]	-1
MESH.UPDATE_BASIS	real	$\tau$ [18]	4.0
MIN_MESH_SIZE	see 7.4.9	$\Delta_{min}^m$ [18]	none
MIN_POLL_SIZE	see 7.4.9	$\Delta_{min}^p$ [18]	none
MULTI_F_BOUNDS	4 reals	see 8.2	none
MULTI_FORMULATION	string	see 8.2	PRODUCT or DIST_L2
MULTI_USE_DELTA_CRIT	bool	see 8.2	no
OPPORTUNISTIC_LUCKY_EVAL	bool	see 7.4.10	none
OPPORTUNISTIC_MIN_EVAL	integer	see 7.4.10	none
OPPORTUNISTIC_MIN_F_IMPRVMT	real	see 7.4.10	none
OPPORTUNISTIC_MIN_NB_SUCCESS	integer	see 7.4.10	none
OPT_ONLY_SGTE	bool	minimize only with surrogates	no
RHO	real	$\rho$ parameter of the progressive barrier	0.1
SEC_POLL_DIR_TYPE	see 7.4.6	type of directions for the secondary poll	see 7.4.6
SNAP_TO_BOUNDS	bool	snap to boundary trial points that are generated outside bounds	yes
SPECULATIVE_SEARCH	bool	MADS speculative search [18]	yes
STAT_SUM_TARGET	real	NOMAD terminates if STAT_SUM reaches this parameter's value	none
STOP_IF_FEASIBLE	bool	NOMAD terminates if it generates a feasible solution	no
USER_CALLS_ENABLED	bool	if no, the automatic calls to user functions are disabled	yes

## 7.3 Output parameters

### 7.3.1 Basic

name	arguments	description	default
CACHE_FILE	string	cache file; if the file does not exist, it will be created	none
DISPLAY_DEGREE	integer in $[0; 4]$ or one string with four digits; see 7.3.2	0: no display; 4: full display	2
DISPLAY_STATS	list of strings	what informations is displayed at each success; see 7.4.7	see 7.4.7
HISTORY_FILE	string	file containing all trial points with format ( x1 x2 ... xn ) on each line	none
SOLUTION_FILE	string	file to save the current best feasible point	none
STATS_FILE	a string file_name plus a list of strings	the same as DISPLAY_STATS but for a display into file file_name	none

### 7.3.2 Advanced

name	arguments	description	default
ADD_SEED_TO_FILE_NAMES	bool	if the seed is added to the file names corresponding to parameters <code>HISTORY_FILE</code> , <code>SOLUTION_FILE</code> and <code>STATS_FILE</code>	yes
CACHE_SAVE_PERIOD	integer <i>i</i>	the cache files are saved every <i>i</i> iterations (disabled for bi-objective)	25
DISPLAY_DEGREE	one string with four digits, each in [0; 4]	1st digit: general display; 2nd digit: search display; 3rd digit: poll display; 4th digit: iterative display; example: <code>DISPLAY_DEGREE 0010</code>	2222
POINT_DISPLAY_LIMIT	integer	maximum number of point coordinates that will be displayed at screen (-1 for no limit)	20
SGTE_CACHE_FILE	string	surrogate cache file (can not be the same as <code>CACHE_FILE</code> )	none

## 7.4 Additional information for some parameters

### 7.4.1 BB\_EXE and SGTE\_EXE

In batch mode, `BB_EXE` indicates the names of the black-boxes executables. In library mode, it is optional as a custom `Evaluator` class can be written with its own `eval_x()` method. A single string can be given if a single black-box is used and gives several outputs. It is also possible to indicate several black-box executables. If the character '\$' is put at first position of a string, this string is considered as a global command or file and no path is added. We give the following examples:

```

BB_EXE          bb.exe          # defines that 'bb.exe' is an
BB_OUTPUT_TYPE  OBJ EB EB       # executable with 3 outputs

BB_EXE          bb1.exe bb2.exe # defines two black-boxes
BB_OUTPUT_TYPE  OBJ      EB     # 'bb1.exe' and 'bb2.exe'
                                # with one output each

BB_EXE "dir $with $spaces/bb.exe # use '$' to describe a
                                # path with spaces

BB_EXE "$python bb.py"          # the black-box is a python
                                # script: it is run with
                                # command
                                # 'python PROBLEM_DIR/bb.py'

BB_EXE "$nice bb.exe"           # to run PROBLEM_DIR/bb.exe
                                # in nice mode on X systems

```

The parameter `SGTE_EXE` associates surrogate executables with black-box executables. It can be entered with two formats: '`SGTE_EXE bb_exe sgte_exe`' to associate executables `bb_exe` and `sgte_exe`, or '`SGTE_EXE sgte_exe`' when only one black-box executable is used. Surrogates have to display the same number of outputs as their associated black-boxes.

### 7.4.2 BB\_INPUT\_TYPE

This parameter indicates the types of each variable. It can be defined once with a list of  $n$  input types with format ( `t1 t2 ... tn` ) or several times with index ranges and input types. Input types are values in  $\{R, C, B, I\}$  or  $\{Real, Cat, Bin, Int\}$ . `R` is for real/continuous variables, `C` for categorical variable (not yet supported), `B` for binary variables, and `I` for integer variables. The default type is `R`.

### 7.4.3 BB\_OUTPUT\_TYPE

This parameter defines the types of the values that the black-box displays. The arguments are a list of  $m$  types, where  $m$  is the number of outputs of the black-box. At least one of these values has to correspond to the objective function that NOMAD minimizes. If two outputs are tagged as objectives, then the BiMADS algorithm will be executed. Other values typically are constraints of the form  $c_j(x) \leq 0$ , and the black-box has to display the left hand side of the constraint with this format. A certain terminology is used to describe the different types of constraints. This terminology can be consulted in [20]. `EB` constraints correspond to constraints that need to be always satisfied (*unrelaxable constraints*). The technique used to deal with those is the *extreme barrier* approach, consisting in simply rejecting the unfeasible points. `PB`, `PEB`, and `F` constraints correspond to constraints that need to be satisfied only at the solution, and not necessarily at intermediate points (*relaxable constraints*). More precisely, `F` constraints are treated with the *filter* approach [17], and `PB` constraints are treated with the *progressive barrier* approach [20]. `PEB` constraints are treated first with the progressive barrier, and once satisfied, with the extreme barrier [22]. There can be another type of constraints, the *hidden constraints*, but these only appear inside the black-box during an execution, and thus they cannot be indicated in advance to NOMAD (when such a constraint is violated, the evaluation simply fails and the point is not considered). If the user is not sure about the nature of its constraints, we suggest using the keyword `CSTR`, which correspond by default to `PB` constraints.

There can be other types of outputs. All the types are:

<code>CNT_EVAL</code>	Must be 0 or 1: count or not the black-box evaluation.
<code>CSTR</code>	The same as <code>PB</code> .
<code>EB</code>	Constraint treated with Extreme Barrier (infeasible points are ignored).
<code>F</code>	Constraint treated with filter approach [17].
<code>NOTHING</code> or <code>-</code>	The output is ignored.
<code>OBJ</code>	Objective value to minimize.
<code>PB</code>	Constraint treated with Progressive Barrier [20].
<code>PEB</code>	Hybrid constraint <code>PB/EB</code> [22].
<code>STAT_AVG</code>	Average of this value will be computed for all black-box calls (has to be unique).
<code>STAT_SUM</code>	Sum of this value will be computed for all black-box calls (has to be unique).

Please note that `F` constraints are not compatible with `CSTR`, `PB` or `PEB`. However, `EB` can be used with `F`, `CSTR`, `PB` or `PEB`.

### 7.4.4 BB\_REDIRECTION

If this parameter is set to **yes** (default), NOMAD manages the creation of the black-box output file when the black-box is executed via a system call (the redirection `>>` is added to the sys-

tem command). If no, then the black-box has to manage the creation of its output file named `TMP_DIR/nomad.SEED.TAG.output`. Values of `SEED` and `TAG` can be obtained in the black-box input files created by `NOMAD` and given as first argument of the black-box, only if parameters `BB_INPUT_INCLUDE_SEED` and `BB_INPUT_INCLUDE_TAG` are both set to `yes`. `TMP_DIR` is specified by the user. If no, `TMP_DIR` is the problem directory.

#### 7.4.5 Bounds

Parameters `LOWER_BOUND` and `UPPER_BOUND` are used to define bounds on variables, and take similar arguments as parameter `FIXED_VARIABLE` (see 7.4.8). For example, with  $n = 7$ ,

<code>LOWER_BOUND</code>	<code>0-2</code>	<code>-5.0</code>
<code>LOWER_BOUND</code>	<code>3</code>	<code>0.0</code>
<code>LOWER_BOUND</code>	<code>5-6</code>	<code>-4.0</code>
<code>UPPER_BOUND</code>	<code>0-5</code>	<code>8.0</code>

is equivalent to

```
LOWER_BOUND ( -5 -5 -5 0 - -4 -4 ) # '-' or '-inf' means that x_4
                                     # has no lower bound
UPPER_BOUND ( 8 8 8 8 8 8 inf )   # '-' or 'inf' or '+inf' means
                                     # that x_6 has no upper bound.
```

These two sequences define the following bounds

$$\left\{ \begin{array}{lll} -5 \leq & x_1 & \leq 8 \\ -5 \leq & x_2 & \leq 8 \\ -5 \leq & x_3 & \leq 8 \\ 0 \leq & x_4 & \leq 8 \\ & x_5 & \leq 8 \\ -4 \leq & x_6 & \leq 8 \\ -4 \leq & x_7 & . \end{array} \right.$$

#### 7.4.6 Direction types

The types of direction correspond to the arguments of parameters `DIRECTION_TYPE` and `SEC_POLL_DIR_TYPE`. Up to 4 strings can be employed to describe one direction type. These 4 strings are `s1` in `{ORTHO,LT,GPS}`, `s2` in `{0,1,2,N+1,2N}`, `s3` in `{0,STATIC,RANDOM}`, and `s4` in `{0,UNIFORM}`. If only 1,2 or 3 strings are given, defaults are considered for the others. Combination of these strings can describe up to 14 different direction types that are:

	s1	s2	s3	s4	direction types
1	ORTHO	1			ORTHOMADS, 1
2	ORTHO	2			ORTHOMADS, 2
3	ORTHO				ORTHOMADS, 2n
3	ORTHO	2N			ORTHOMADS, 2n
4	LT	1			LT-MADS, 1
5	LT	2			LT-MADS, 2
6	LT	N+1			LT-MADS, n+1
7	LT				LT-MADS, 2n
7	LT	2N			LT-MADS, 2n
8	GPS	BIN			GPS for binary variables
9	GPS	N+1			GPS, n+1, static
9	GPS	N+1	STATIC		GPS, n+1, static
10	GPS	N+1	STATIC	UNIFORM	GPS, n+1, static, uniform angles
11	GPS	N+1	RAND		GPS, n+1, random
12	GPS	N+1	RAND	UNIFORM	GPS, n+1, random, uniform angles
13	GPS				GPS, 2n, static
13	GPS	2N			GPS, 2n, static
13	GPS	2N	STATIC		GPS, 2n, static
14	GPS	2N	RAND		GPS, 2n, random.

GPS directions correspond to the coordinate directions. LT and ORTHO directions correspond to the implementations LT-MADS [18] and ORTHOMADS [6] of MADS. The integer indicated after GPS, LT and ORTHO corresponds to the number of directions that are generated at each poll. The 14 different direction types can be chosen together by specifying `DIRECTION_TYPE` or `SEC_POLL_DIR_TYPE` several times. If nothing indicated, ORTHO is considered for the primary poll, and default direction types for the secondary poll are ORTHO 2, LT 2 and GPS N+1 STATIC depending on the value of `DIRECTION_TYPE`.

#### 7.4.7 DISPLAY\_STATS and STATS\_FILE

These parameters display information each time a new feasible incumbent is found. `DISPLAY_STATS` displays in the standard output and `STATS_FILE` writes a file. These parameters need a list of strings as argument, which can include these keywords:

BBE	Black-box evaluations.
BBO	All black-box outputs.
EVAL	Evaluations (includes cache-hits).
MESH_INDEX	Mesh index $\ell$ [6].
OBJ	Objective function value.
SGTE	Number of surrogate evaluations.
SIM_BBE	Simulated black-box evaluations (includes initial cache-hits).
SOL	Solution, with format <code>iSOLj</code> where <code>i</code> and <code>j</code> are two (optional) strings: <code>i</code> will be displayed before each coordinate, and <code>j</code> after each coordinate (except the last).
STAT_AVG	The AVG statistic (argument <code>STAT_AVG</code> of <code>BB_OUTPUT_TYPE</code> ).
STAT_SUM	The SUM statistic defined by argument <code>STAT_SUM</code> for parameter <code>BB_OUTPUT_TYPE</code> .
TIME	Wall-clock time.

For example, `'DISPLAY_STATS $BBE$ & ( $SOL, ) & $OBJ$ \\'` will display lines similar to `'$1$ & ( $10$ , $5$ ) & $-703.4734809$ \\'`, which can be directed copied into L<sup>A</sup>T<sub>E</sub>X tables. Default values are `'DISPLAY_STATS BBE OBJ'` and `'DISPLAY_STATS OBJ'` for single and bi-objective optimization, respectively (there is no need to enter OBJ twice in order for the two objective values to be displayed).

To write these outputs into the file `output.txt`, simply add the file name as first argument of `STAT_FILE`: for example `'STAT_FILE output.txt BBE ( SOL ) OBJ'`.

#### 7.4.8 FIXED\_VARIABLE

This parameter is used to fix some variables to a value. This value is optional if at least one starting point is defined. The parameter can be entered with several types of arguments:

- A string indicating a text file containing  $n$  values. Variables will be fixed to the values that are not defined with the character `'-'`.
- A vector of  $n$  values with format `(v0 v1 ... vn-1)`. Again, character `'-'` can be used for free variables.
- An index range if at least one starting point has been defined (see 7.4.13 for practical examples of index ranges).
- An index range and a real value, with format `'FIXED_VARIABLE i-j v'`: variables  $i$  to  $j$  will be fixed to the value  $v$  ( $i-j$  can be replaced by  $i$ ).

#### 7.4.9 Mesh and poll size parameters

The initial mesh size parameter  $\Delta_0^m$  [18] is decided by `INITIAL_MESH_SIZE`. In order to achieve the scaling between variables, NOMAD considers the mesh size parameter as a vector of  $n$  elements. The same logic applies to the poll size parameter  $\Delta_k^p$ . If  $d0$  is a positive real value, `INITIAL_MESH_SIZE` can be entered with the following formats:

- `INITIAL_MESH_SIZE d0`: initial mesh size for all variables.
- `INITIAL_MESH_SIZE (d0 d1 ... dn-1)`: for all variables (`'-'` can be used, and defaults will be considered).
- `INITIAL_MESH_SIZE i d0`: initial for variable  $i$ .
- `INITIAL_MESH_SIZE i-j d0`: initial for variables  $i$  to  $j$ .

The minimum mesh size  $\Delta_{min}^m$  and the minimum poll size  $\Delta_{min}^p$  (stopping criteria) can be defined the same way via parameters `MIN_MESH_SIZE` and `MIN_POLL_SIZE`. All values can also be preceded by `'r'` to indicate a value relative to the bounds. For example, `'INITIAL_MESH_SIZE r0.1'` means that  $\Delta_0^m = (ub - lb)/10$  with  $lb, ub \in \mathbb{R}^n$  and  $lb \leq x \leq ub$  for all  $x \in X$ . Default is `r0.1` for bounded variables and `1.0` otherwise.

#### 7.4.10 Opportunistic strategy

The opportunistic strategy consists in terminating the evaluations of a list of trial points as soon as an improved value is found. This strategy is decided with the parameter `OPPORTUNISTIC_EVAL` and

applies to both the poll and search steps. For the LH search, the strategy can be chosen independently with `OPPORTUNISTIC_LH`. If this parameter is not defined, the parameter `OPPORTUNISTIC_EVAL` applies to the LH search. Other defaults are considered for bi-objective optimization (see 8.2).

If the opportunistic strategy is enabled, some options can be defined via the following parameters:

- `OPPORTUNISTIC_MIN_NB_SUCCESS` *i*: do not terminate before *i* successes.
- `OPPORTUNISTIC_MIN_EVAL` *i*: do not terminate before *i* evaluations.
- `OPPORTUNISTIC_MIN_F_IMPRVMT` *r*: terminate only if *f* is reduced by *r*%.
- `OPPORTUNISTIC_LUCKY_EVAL` *yes/no*: perform an additional black-box evaluation after an improvement.

#### 7.4.11 `TMP_DIR`

If NOMAD is installed on a network, with the batch mode use, the cost of read/write files will be high if no local temporary directory is defined. On `Linux/Unix/Mac OS X` systems, the directory `/tmp` is local and we advise the user to define `'TMP_DIR /tmp'`.

#### 7.4.12 `VARIABLE_GROUP`

This parameter can be entered several times to define several groups of variables. Variables in one group can be of different types (except for categorical variables). To define some particular types of directions or a particular Halton seed for this group, use the NOMAD library and `Parameters::set_VARIABLE_GROUP()`. In addition to the groups defined by parameters, NOMAD creates one group for all continuous, integer, and binary variables, and one group for categorical variables. If a group contains only binary variables, directions of type `_GPS_BINARY_` will be automatically used.

#### 7.4.13 `X0`

Parameter `X0` indicates the starting point of the algorithm. Several starting points can be provided by entering this parameter several times. If no starting point is indicated, NOMAD considers the best evaluated point from an existing cache file (parameter `CACHE_FILE`) or from an initial Latin-Hypercube search (argument `p0` of `LH_SEARCH`). The `X0` parameter can take several types of arguments:

- A string indicating an existing cache file, containing several points (they can be already evaluated or not). This file can be the same as the one given to `CACHE_FILE`. If so, this file will be updated during the program execution, else the file will not be modified.
- A string indicating a text file that contains the coordinates of one point (values are separated by spaces or line breaks).
- *n* real values with format `(v0 v1 . . . . vn-1)`.
- One integer (or range of integers) and one real:
  - `'X0 i v'`: (*i*+1)th coordinate set to *v*.
  - `'X0 i-j v'`: coordinates *i* to *j* set to *v*.

– ‘X0 \* v’: all coordinates set to v.

- One integer, another integer (or index range) and one real: the same as above except that the first integer k refers to the (k+1)th starting point.

The following example with  $n = 3$  corresponds to the two starting points (5 0 0) and (−5 1 1):

X0	*	0.0
X0	0	5.0
X0	1	* 1.0
X0	1	0 −5.0

## 8 Special functionalities

### 8.1 Categorical variables

Categorical variables are discrete variables that can take a finite number of values. These are not integer or binary variables as there is no ordering property amongst the different values that can take the variables. In order to decide these values and to define a neighborhood structure, a user written procedure must be provided.

The algorithm used by NOMAD to treat such variables is defined in references [1, 8, 9, 15, 43]. It works as follows.

#### 8.1.1 Algorithm

At the end of an iteration, if no improvement has been made, a special step occurs, the *extended poll*. The extended poll first calls the user-provided procedure defining the neighborhood of categorical variables. The procedure returns a list of points that has been defined as neighbors of the current iterate. These points are called the *extended poll points*. The functions defining the problem are then evaluated at each of these points, and the objective values are compared to the current best value.

If the difference between the objective value at the current iterate and at a extended poll point is less than a parameter called the *extended poll trigger*, this extended poll point is called an *extended poll center* and a new MADS run is performed from this point. This run is called an *extend poll descent*, and occurs on meshes that cannot be shrunk more than the mesh of the beginning of the extended poll. If the opportunistic strategy is active, the different extended poll descents are performed until a new success is achieved.

If surrogates are available, they can be used to evaluate on the neighbors, and during the extended poll descent. The true functions will then be evaluated only on the most promising points. With surrogates, the extended poll costs at most the same number of true evaluations than the number of neighbors determined by the user-provided procedure.

#### 8.1.2 Categorical variables with NOMAD

We suggest the reader to follow this section in parallel with the new example 14 illustrating a simple mixed variable problem optimization. Dealing with such problems in NOMAD implies the use of the library mode in order to define the categorical variables neighborhoods, although it is still possible to use a parameters file and a separated black-box executable.

Several steps are necessary. First the BB\_INPUT\_TYPE parameter has to be defined with the value ‘C’ to identify categorical variables.



Then the library mode is necessary in order to define the procedure for the categorical variable neighborhoods. This procedure corresponds to the virtual method `Extended_Poll::construct_extended_points()` and the user has to design its own `Extended_Poll` subclass in which `construct_extended_points()` is coded. This method takes as argument a point (the current iterate) and constructs the list of extended poll points (the neighbors of the current iterate). These points are registered with the method `Extended_Poll::add_extended_poll_point()`. In its `main` function, the user gives its own `Extended_Poll` object to the `Mads` object used to optimize the problem. If no `Extended_Poll` is provided to the `Mads` object, the program will generate an error.

An important feature of NOMAD about the handling of categorical variables is its ability to treat points with a variable number of variables. This behavior often happens in mixed variable problems, for example when a categorical variable indicates the number of continuous variables.

To deal with this, NOMAD uses the concept of *signature*, implemented in the `Signature` class. Each point in the algorithm possesses a signature, indicating the characteristics related to the variables. These characteristics are the number of variables, the types of the variables, their bounds, fixed and periodic variables, and some information on the initial mesh size parameter for each variable (this determines the scaling of the variables).

In the user-provide `Extended_Poll` subclass, for each extended poll point, a signature has to be provided. If the extended poll point have the same number of variables than the current iterate, the signature of the current iterate can be used. Otherwise, a new creature must be created. See the `Signature` class for details about creating signatures.

The number of outputs of the evaluations must remain constant: if a problem has been defined with one objective and two constraints, even if the number of variables can change, each evaluation should return three values.

The parameter `DIMENSION` has still to be defined, and it corresponds to the number of variable of the provided starting points. For these starting point, the `Parameters` class will automatically create a standard signature. If no starting point is provided, the standard signature cannot be created, and a Latin-Hypercube search cannot be executed, because it has no reference for defining values for categorical variables. So not providing starting points with mixed variable problems is not allowed.

The main parameter for mixed variable optimization is the extended poll trigger. Its value is given with the parameter `EXTENDED_POLL_TRIGGER`, and can be indicated as a relative value. The extended poll trigger is used to compare the objective values at an extended poll point  $y$  and at the current iterate  $x_k$ . If  $f(y) < f(x_k) + \text{trigger}$ , then  $y$  becomes an extended poll center from which a MADS run is performed. The default trigger value is `r0.2`, meaning that an extended poll point will become an extended poll center if  $f(y)$  is less than  $f(x_k) + f(x_k) \times 20\%$ . See the function `Extended_Poll::check_trigger()` for the details of this test, and for the cases where infeasible points or surrogate evaluations are considered.

Finally, note that the parameter `EXTENDED_POLL_ENABLED` can simply disable the extended poll. In this case, categorical variable are simply fixed.

## 8.2 Bi-objective optimization

NOMAD can perform bi-objective optimization through the BiMADS algorithm described in [25]. Handling of more than two objective functions will be implemented in future versions.

The BiMADS algorithm solves biobjective problems of the form

$$\min_{x \in \Omega} F(x) = (f_1(x), f_2(x)). \quad (2)$$

The algorithm launches successive runs of MADS on single-objective reformulations of the problem. The Pareto front, or the list of points that are dominant following the definition of [25], is constructed with the evaluations performed during these MADS runs.

Two considerations have to be taken into account when generating Pareto fronts: the quality of approximation of the dominant points, and the repartition of these points. The quality of approximation can be measured with the `surf` criterion which gives the ratio of the area under the graph of the front relatively to a box enclosing all points (small values indicates a good front).

The quality of the coverage of the Pareto front is measured by the  $\delta$  criterion, which corresponds to the largest distance between two successive Pareto points.

To define that a problem has two objectives, two arguments of the parameter `BB_OUTPUT_TYPE` have to be set to `OBJ`. Then, NOMAD will automatically run the BIMADS algorithm. Additional parameters are:

- `MULTI_F_BOUNDS f1_min f1_max f2_min f2_max` (real values): these 4 values are necessary to compute the `surf` criterion. If not entered or if not valid (for example if `f1_min` is too big), then `surf` is not computed.
- `MULTI_FORMULATION` (string): single-objective reformulation [26]. This is how NOMAD computes one value from the two objective values. The argument must be in `{NORMALIZED, PRODUCT, DIST_L1, DIST_L2, DIST_LINF}` (`DIST_LINF` and `NORMALIZED` are equivalent). The default formulation is `PRODUCT` when VNS is not used, and `DIST_L2` otherwise.
- `MULTI_NB_MADS_RUNS` (integer): the number of MADS runs.
- `MULTI_OVERALL_BB_EVAL` (integer): the maximum number of black-box evaluations over all MADS runs.
- `MULTI_USE_DELTA_CRIT` (bool, default to `no`): use or not a stopping criterion based on the  $\delta$  measure.

Default values are considered if these parameters are not entered. All other MADS parameters are considered and apply to single MADS runs, with some adaptations:

- The `MAX_BB_EVAL` parameter corresponds to the maximum number of black-box evaluations for one MADS run.
- The `F_TARGET` parameter is adapted to bi-objective: it has to be given with two values  $z_1$  and  $z_2$ . If a point  $x$  is generated such that  $f_1(x) \leq z_1$  and  $f_2(x) \leq z_2$ , then the algorithm terminates.
- Latin-Hypercube (LH) search (`LH_SEARCH p_0 p_1`): in single-objective optimization, `p_0` and `p_1` correspond to the initial number of search points and to the number of search points at each iteration, respectively. In the bi-objective context, `p_0` is the number of initial search points generated in the first MADS run, and `p_1` is the number of points for the second MADS run. If no LH search is defined by user, and if only `MULTI_OVERALL_BB_EVAL` is defined, then a default LH search is performed. Moreover, this default LH search is non-opportunistic (`OPPORTUNISTIC_LH` set to `no`).
- The parameter `SOLUTION_FILE` is disabled.

The NOMAD solution represents an approximation of the Pareto front, and is accessible via the `DISPLAY_STATS` or `STATS_FILE` parameters. If `DISPLAY_DEGREE` is greater than 2, then the two measures `surf` and  $\delta$  are displayed.

For a given budget of black-box evaluations (`MULTI_OVERALL_BB_EVAL`), if the quality of approximation is desired (small value for `surf`), then single MADS optimizations will have to terminate after more severe criteria (for example a large number of black-box evaluations, via `MAX_BB_EVAL`). If a better repartition of the points is desired (small value for  $\delta$ ), then the number of MADS runs should be larger, with less severe stopping criteria on single-objective optimizations.

### 8.3 Variable Neighborhood Search

This search strategy is described in [12]. It is based on the Variable Neighborhood Search meta-heuristic [47, 48] as a search strategy in order to escape local minima. VNS should only be used for problems with a several of such local optima. It will cost some additional evaluations, since each search performs another MADS run from a perturbed starting point. Though, it will be a lot cheaper if a surrogate is provided via parameter `HAS_SGTE` or `SGTE_EXE`. We advise the user not to use VNS with bi-objective optimization, as the BIMADS algorithm already performs multiple MADS runs.

In order to use the VNS search, which is disabled by default, the user has to define the parameter `VNS_SEARCH`, with a boolean or a real. This expected real value is the *VNS trigger*, which corresponds to the maximum desired ratio of VNS black-box evaluations over the total number of black-box evaluations. For example, a value of 0.75 means that NOMAD will try to perform a maximum of 75% black-box evaluations from the VNS search. If a boolean is given as value to `VNS_SEARCH`, then a default of 0.75 is taken for the VNS trigger.

From a technical point of view, VNS is coded as a `Search` sub-class, and it is a good example of how a user-search can be implemented. See files `$NOMAD_HOME/src/VNS_Search.*pp` for details.

## 9 Future versions

### 9.1 Algorithm future developments

- Compatibility with AMPL [39].
- Parallel versions.
- Dynamic surrogates.
- Use of simplex gradients [33, 34].

### 9.2 Future packages

- COOP-MADS: several MADS instances launched in parallel with a cache server.
- PSD-MADS: parallel space decomposition of MADS [21].
- MULTI-MADS: multi-objective variant of MADS [25], with 3 and more objective functions.

## Acknowledgments

Developers of NOMAD wish to thank Mohamed Sylla and Quentin Reynaud, both from [ISIMA](#), for their contribution to the project during summer internships. Thanks also to Maud Bay, Eve Bélisle, Alexander Lutz, and Rosa-Maria Torres-Calderon, for their tests and feedback.

## Related publications

- [1] M. A. Abramson. Mixed variable optimization of a load-bearing thermal insulation system using a filter pattern search algorithm. *Optimization and Engineering*, 5(2):157–177, 2004.
- [2] M. A. Abramson. Second-order behavior of pattern search. *SIAM Journal on Optimization*, 16(2):315–330, 2005.
- [3] M. A. Abramson and C. Audet. Convergence of mesh adaptive direct search to second-order stationary points. *SIAM Journal on Optimization*, 17(2):606–619, 2006.
- [4] M. A. Abramson, C. Audet, G. Couture, J. E. Dennis, Jr., and S. Le Digabel. The NOMAD project. Software available at <http://www.gerad.ca/nomad>.
- [5] M. A. Abramson, C. Audet, and J. E. Dennis, Jr. Generalized pattern searches with derivative information. *Mathematical Programming*, Series B, 100:3–25, 2004.
- [6] M. A. Abramson, C. Audet, J. E. Dennis, Jr., and S. Le Digabel. OrthoMADS: A deterministic MADS instance with orthogonal directions. *SIAM Journal on Optimization*, 20(2):948–966, 2009.
- [7] M. A. Abramson, O. A. Brezhneva, J. E. Dennis Jr., and R. L. Pingel. Pattern search in the presence of degenerate linear constraints. *Optimization Methods and Software*, 23(3):297–319, 2008.
- [8] M.A. Abramson, C. Audet, J.W. Chrissis, and J.G. Walston. Mesh adaptive direct search algorithms for mixed variable optimization. *Optimization Letters*, 3(1):35–47, January 2009.
- [9] M.A. Abramson, C. Audet, and J. E. Dennis, Jr. Filter pattern search algorithms for mixed variable constrained optimization problems. *Pacific Journal on Optimization*, 3(3):477–500, 2007.
- [10] C. Audet. Convergence Results for Pattern Search Algorithms are Tight. *Optimization and Engineering*, 5(2):101–122, 2004.
- [11] C. Audet, V. Béchar, and J. Chaouki. Spent potliner treatment process optimization using a MADS algorithm. *Optimization and Engineering*, 9(2):143–160, 2007.
- [12] C. Audet, V. Béchar, and S. Le Digabel. Nonsmooth optimization through mesh adaptive direct search and variable neighborhood search. *Journal of Global Optimization*, 41(2):299–318, June 2008.
- [13] C. Audet, A. J. Booker, J. E. Dennis, Jr., P. D. Frank, and D. W. Moore. A surrogate-model-based method for constrained optimization. Presented at the 8th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, 2000.

- [14] C. Audet, A. L. Custódio, and J. E. Dennis, Jr. Erratum: Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, 18(4):1501–1503, 2008.
- [15] C. Audet and J. E. Dennis, Jr. Pattern search algorithms for mixed variable programming. *SIAM Journal on Optimization*, 11(3):573–594, 2000.
- [16] C. Audet and J. E. Dennis, Jr. Analysis of generalized pattern searches. *SIAM Journal on Optimization*, 13(3):889–903, 2003.
- [17] C. Audet and J. E. Dennis, Jr. A pattern Search Filter Method for Nonlinear Programming without Derivatives. *SIAM Journal on Optimization*, 14(4):980–1010, 2004.
- [18] C. Audet and J. E. Dennis, Jr. Mesh Adaptive Direct Search Algorithms for Constrained Optimization. *SIAM Journal on Optimization*, 17(1):188–217, 2006.
- [19] C. Audet and J. E. Dennis, Jr. Nonlinear programming by mesh adaptive direct searches. *SIAG/Optimization Views-and-News*, 17(1):2–11, 2006.
- [20] C. Audet and J. E. Dennis, Jr. A Progressive Barrier for Derivative-Free Nonlinear Programming. *SIAM Journal on Optimization*, 20(4):445–472, 2009.
- [21] C. Audet, J. E. Dennis, Jr., and S. Le Digabel. Parallel space decomposition of the mesh adaptive direct search algorithm. *SIAM Journal on Optimization*, 19(3):1150–1170, 2008.
- [22] C. Audet, J. E. Dennis, Jr., and S. Le Digabel. Globalization strategies for mesh adaptive direct search. *To appear in Computational Optimization and Applications*, 2009.
- [23] C. Audet and S. Le Digabel. The mesh adaptive direct search algorithm for periodic variables. Technical Report G-2009-23, Les cahiers du GERAD, 2009.
- [24] C. Audet and D. Orban. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization*, 17(3):642–664, 2006.
- [25] C. Audet, G. Savard, and W. Zghal. Multiobjective Optimization Through a Series of Single-Objective Formulations. *SIAM Journal on Optimization*, 19(1):188–210, 2008.
- [26] C. Audet, G. Savard, and W. Zghal. A mesh adaptive direct search algorithm for multiobjective optimization. *To appear in European Journal of Operational Research*, 2009.
- [27] A. J. Booker, E. J. Cramer, P. D. Frank, J. M. Gablonsky, and J. E. Dennis, Jr. Movars: Multidisciplinary optimization via adaptive response surfaces. AIAA Paper 2007–1927, Presented at the 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Honolulu, 2007.
- [28] A. J. Booker, J. E. Dennis, Jr., P. D. Frank, D. W. Moore, and D. B. Serafini. Managing surrogate objectives to optimize a helicopter rotor design – further experiments. AIAA Paper 1998–4717, Presented at the 8th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, 1998.
- [29] A. J. Booker, J. E. Dennis, Jr., P. D. Frank, D. B. Serafini, and V. Torczon. Optimization using surrogate objectives on a helicopter test example. In J. Borggaard, J. Burns, E. Cliff, and S. Schreck, editors, *Optimal Design and Control*, Progress in Systems and Control Theory, pages 49–58, Cambridge, Massachusetts, 1998. Birkhäuser.

- [30] A. J. Booker, J. E. Dennis, Jr., P. D. Frank, D. B. Serafini, V. Torczon, and M. W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural Optimization*, 17(1):1–13, February 1999.
- [31] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: A Users' Guide*. The Scientific Press, Danvers, Massachusetts, 1988.
- [32] E. J. Cramer, J. E. Dennis, Jr., P. D. Frank, R. M. Lewis, and G. R. Shubin. Problem formulation for multidisciplinary optimization. In *AIAA Symposium on Multidisciplinary Design Optimization*, September 1993.
- [33] A. L. Custódio, J. E. Dennis, Jr., and L. N. Vicente. Using simplex gradients of nonsmooth functions in direct search methods. *IMA Journal of Numerical Analysis*, 28(4):770–784, 2008.
- [34] A. L. Custódio and L. N. Vicente. Using sampling and simplex derivatives in pattern search methods. *SIAM Journal on Optimization*, 18(2):537–555, May 2007.
- [35] J. E. Dennis, Jr., C. J. Price, and I. D. Coope. Direct search methods for nonlinearly constrained optimization using filters and frames. *Optimization and Engineering*, 5(2):123–144, June 2004.
- [36] J. E. Dennis, Jr. and V. Torczon. Direct search methods on parallel machines. *SIAM Journal on Optimization*, 1(4):448–474, November 1991.
- [37] S. Le Digabel. NOMAD: Nonsmooth Optimization with the MADS algorithm. Technical Report G-2009-39, Les cahiers du GERAD, 2009.
- [38] S. Le Digabel. NOMAD user guide. Technical Report G-2009-37, Les cahiers du GERAD, 2009.
- [39] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Thomson/Brooks/Cole, Pacific Grove, California, second edition, 2003.
- [40] K.R. Fowler, J.P. Reese, C.E. Kees, J.E. Dennis Jr., C.T. Kelley, C.T. Miller, C. Audet, A.J. Booker, G. Couture, R.W. Darwin, M.W. Farthing, D.E. Finkel, J.M. Gablonsky, G. Gray, and T.G. Kolda. Comparison of derivative-free optimization methods for groundwater supply and hydraulic capture community problems. *Advances in Water Resources*, 31(5):743–757, May 2008.
- [41] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTer (and SifDec): a constrained and unconstrained testing environment, revisited. *ACM Transactions on Mathematical Software*, 29(4):373–394, 2003.
- [42] R. E. Hayes, F. H. Bertrand, C. Audet, and S. T. Kolaczowski. Catalytic combustion kinetics: Using a direct search algorithm to evaluate kinetic parameters from light-off curves. *The Canadian Journal of Chemical Engineering*, 81(6):1192–1199, 2003.
- [43] M. Kokkolaras, C. Audet, and J. E. Dennis, Jr. Mixed variable optimization of the number and composition of heat intercepts in a thermal insulation system. *Optimization and Engineering*, 2(1):5–29, 2001.
- [44] A. L. Marsden, M. Wang, J. E. Dennis, Jr., and P. Moin. Optimal aeroacoustic shape design using the surrogate management framework. *Optimization and Engineering*, 5(2):235–262, 2004.

- [45] A. L. Marsden, M. Wang, J. E. Dennis, Jr., and P. Moin. Suppression of airfoil vortex-shedding noise via derivative-free optimization. *Physics of Fluids*, 16(10):L83–L86, 2004.
- [46] A. L. Marsden, M. Wang, J. E. Dennis, Jr., and P. Moin. Trailing-edge noise reduction using derivative-free optimization and large-eddy simulation. *Journal of Fluid Mechanics*, 572:13–36, February 2007.
- [47] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.
- [48] P. Hansen N. Mladenović. Variable neighborhood search: principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001.
- [49] M. S. Ouali, H. Aoudjit, and C. Audet. Optimisation des stratégies de maintenance. *Journal Européen des Systèmes Automatisés*, 37(5):587–605, 2003.
- [50] T. A. Sriver, J. W. Chrissis, and M. A. Abramson. Pattern search ranking and selection algorithms for mixed variable stochastic optimization, 2004. Preprint.