NOMADm version 4.6 User's Guide

Mark A. Abramson
Department of Mathematics and Statistics
Air Force Institute of Technology
2950 Hobson Way,
Wright-Patterson AFB, Ohio, 45433-7765 USA
`Mark.Abramson@afit.edu, http://www.afit.edu/en/enc/Faculty/MAbramson/abramson.html`

December 14, 2007

Key words: nonlinear programming, mixed variable programming, pattern search algorithm, derivative-free optimization, optimization software

Abstract: NOMADm is a suite of MATLAB® functions for numerically solving nonlinear and mixed variable optimization problems. The underlying optimizer used in the code is an implementation of the class of mesh adaptive direct search (MADS) filter algorithms. This User's Guide describes the target class of problems, the MADS optimization algorithm, the NOMADm program files, the user files needed to solve a problem, and the graphical user interface (GUI), along with its menu choices and functions.

1 Introduction

NOMADm is a MATLAB® implementation of the class of Mesh Adaptive Direct Search (MADS) filter algorithms for numerically solving nonlinear and mixed variable optimization problems with general nonlinear inequality constraints. The target class of problems for this software has the following properties:

- Variables can be continuous, discrete, or even categorical (variables whose values must always come from a predefined list; *e.g.*, color, shape, material type).
- Constraints can be bound, linear, or nonlinear inequality constraints. They can also change, appear, or disappear, based on the values of some of the categorical variables.
- Objective and constraint functions can be discontinuous, extended-valued, nonsmooth, "black box", and computationally expensive to evaluate. However, more smoothness yields stronger theoretical convergence properties.
- Derivatives are completely optional. The MADS algorithm does not require them to ensure convergence, but any available derivative information – even signs of partial derivatives – can be exploited to reduce computational cost.

The underlying MADS algorithm, which is a generalization of pattern search, is a derivative-free method in the sense that it does not require any derivative information to converge to a limit point that satisfies certain optimality conditions. However, this means that more function evaluations would normally be used by MADS than by a derivative-based approach. Thus, nicely behaved problems, in which derivatives are available, would most likely be solved more efficiently using a derivative-based method. The target class of this software is more difficult problems, most often found in engineering design, in which functions behave badly.

We should note that equality constraints are difficult to handle because of basic properties of the MADS algorithm, and also due to numerical considerations. In order to avoid the problems that equality constraints present, the user should eliminate one variable and reformulate these constraints as inequalities.

The remainder if the document is as follows. As a precursor to the software description, the basics of the algorithm are described in Section **??**, with appropriate references for more detailed descriptions. Installation instructions are given in Section **??**, followed by a detailed description in Section **??** of the data structures used in the code and the required formats for user-provided files that describe the optimization to be solved. Section **??** describes the NOMADm graphical user interface (GUI) and all the associated menu functions. Section **??** describes how to set up user defined search and surrogate optimization procedures, and Section **??** describes the approach for handling optimization functions when the objective and nonlinear constraints are written in FORTRAN or C. Finally, Section **??** gives web site references for certain external MATLAB packages that NOMADm incorporate when using surrogates to perform the optimization.

2 The MADS algorithm

The mesh adaptive direct search (MADS) algorithm [**?**] is basically an extension of pattern search methods [**?**, **?**] to nonlinearly constrained optimization problems, without using any type of penalty function approach or filter. This extension is made possible by generating a dense set of search directions (in the limit) from a core finite set of directions, enabling strong convergence properties to be proved. [**?**, **?**]

The MADS class of algorithms begins with an initial point with finite function value, followed by series of iterations, each of which consists of an optional SEARCH step and a local POLL step. Both steps evaluate points on a carefully constructed mesh in an attempt to find an *improved mesh point*; *i.e.*, one with a lower objective function value than that of the incumbent. The mesh is constructed as a lattice based on a set of directions that form a positive spanning set (a set of vectors such that any vector in the space can be represented by a nonnegative linear combination of the vectors in the set).

In the SEARCH step, any finite set of mesh points can be evaluated (including none). This allows the user great flexibility in choosing strategies. Common practices include random search, a few generations of a genetic algorithm, Latin hypercube search, or orthogonal arrays.

For computationally expensive functions, a common choice for the SEARCH step is to initially invest a moderate number of function evaluations to form inexpensive surrogates for the objective and constraint functions, and then, at each subsequent iteration, solve the surrogate problem on the mesh at a fraction of the cost (*e.g.*, see [**?**]). If the surrogate optimization fails to find an improved mesh point, then the surrogate is updated (and made more accurate) with the new point. The use of surrogates often yields significant improvement in the objective function value early on in the iteration process.

If the SEARCH step is empty or unsuccessful in finding an improved mesh point, then the POLL step is invoked. In this more rigidly defined step, the neighboring (adjacent) mesh points to the current iterate are evaluated. The fact that these neighboring points are constructed by means of positive spanning sets drive the convergence theory for the algorithm.

If the SEARCH and POLL steps are both unsuccessful, then the current incumbent is declared to be a *mesh local optimizer*, and the mesh is refined by reducing a single mesh size parameter. If an improved mesh point is found in either step, then the mesh is either retained or coarsened by increasing the mesh size parameter. Rules for mesh refinement and coarsening can be found in numerous sources (see, for example, [?, ?, ?, ?]).

2.1 Constraints

For bound and linearly constrained problems, the only additional requirement over those of unconstrained problems is that the set of poll directions must be chosen so as to conform to the geometry of the constraints. An algorithm for doing this using standard linear algebra tools is described in [?] for the non-degenerate case, and in [?] if a degenerate condition exists. Both have been incorporated into NOMADm.

Nonlinear constraints are treated in NOMADm by using a filter [?], or the MADS-based approach for constructing poll directions [?], or both.

A filter approach attempts to minimize two functions, the objective function f and a function h that measures aggregate constraint violation. In the context of MADS and GPS, a filter is a collection of points, none of which dominates any other in the set with respect to their objective and constraint violation function values. That is, for any two points x, y in the filter, the condition $f(x) < f(y)$ and $h(x) < h(y)$, cannot hold.

Polling is performed around either the best feasible point found thus far, or the least infeasible point in an attempt to add new points to the filter – preferably new poll centers. A much more detailed explanation can be found in [?].

Unfortunately, the theory for the GPS filter algorithm cannot quite ensure convergence to a first-order stationary point because it would require an infinite set of directions. The MADS class of algorithms circumvents this problem by decoupling the mesh size parameter from the "poll parameter" in a clever way. Under mild conditions, both parameters approach zero in the limit, but the mesh size parameter approaches at a faster rate, which has the effect of increasing the number of poll directions toward infinity as the parameters go to zero, thus resulting in a dense set of poll directions [?]. Of course, we cannot realistically poll in all of these directions. The strategy currently used in NOMADm is (at each iteration) to randomly choose a set of positive spanning directions from among the allowable (and ever-increasing) set of directions.

Clearly, this description of the class of MADS algorithms is incomplete, as it has taken several published papers to fully describe. Hopefully, this brief overview describes enough of the algorithm to be of assistance to the user in using the software.

3 Installation and Setup

The NOMADm software files are stored in a zipped file, `nomadm.zip`, which is publicly available for download from the NOMADm web site [?]. A brief description of each file is given as follows:

`nomadm.m`: The NOMADm graphical user interface (GUI) for the MADS optimizer.

`mads_batch.m`: An example batch file for running MADS outside of the NOMADm GUI.

`mads.m`: The filter MADS optimizer.

`mads_defaults.m`: The default labels and options for both NOMADm and the MADS optimizer.

`nomadm_compile.m`: A utility for compiling functions written in FORTRAN and C for use with NOMADm.

`nomadm_help.pdf`: This NOMADm Help file.

`nomadm.txt`: Summary of recent version changes.

`Contents.m`**:** Summary of NOMADm files.

`gpl.txt`**:** The GNU General Public License.

All of these files need to be stored in the same directory, which *must* be added to the MATLAB path. This is done by selecting *Set Path* from the File menu in the main MATLAB window, and then in the Set Path Window, pressing the *Add Folder* button, selecting the correct directory, and pressing the *Save* button. This need only be done once.

For maximum flexibility, the MADS optimizer can be run with or without the NOMADm GUI. The batch file listed above provides an example of how one might set up the necessary variables to run MADS as a standalone optimizer.

NOMADm is also integrated with the DACE Toolbox, NW Toolbox, and a genetic algorithm code, called CMA-ES, for use in the MADS SEARCH step. If any of these publicly available packages is used (links are listed on the NOMADm web site [**?**]), then the location of its files must also be added to the MATLAB path prior to launching NOMADm. Once this is done, help documentation files associated with these software packages will be visible in the NOMADm Help menu.

4 Problem Files and Data Structures

While the MADS optimizer allows great flexibility in file naming conventions, the NOMADm GUI specifies a fixed convention that must be used. Each optimization problem must have a name, which all its associated files use as part of their filenames. All of these files must be in the same directory, and all except the Cache and Session files, which are created by NOMADm, must be supplied by the user. Each file's structure is now described in detail for the default problem name, `myproblem`.

For the discussion below, an `iterate` is a structure with two fields `x` and `p`, written as `iterate.x` and `iterate.p`, the former being a vector containing the values of the continuous variables, and the latter being a cell array containing the values of the categorical variables. The cell array may contain integer, real, or string data. If the problem does not have categorical variables, then `iterate.p` is set to empty (using curly braces). Additional fields are added by the optimizer as the iterate is being evaluated by the objective and constraint functions.

4.1 Functions File

This file, which for the default problem takes the file name, `myproblem.m`, evaluates an iterate and returns corresponding objective and constraint function values, along with their derivatives (if available). Actually, the input argument is not in `iterate` form because this would make it difficult for the function to be used by other applications; rather, it is broken down into its two components, a vector of continuous variables and an (optional) cell array of categorical variables. If there are no categorical variables, the second input argument is ignored. If derivatives are not desired or available, the user can omit them in the output argument list. If there are no nonlinear constraints, they may be left out of the output argument list only if gradients are not requested; otherwise, the user can simply set the nonlinear constraints to be empty. If gradients are requested, but some of the partial derivatives are unavailable, the user must assign the unavailable values to "NaN". Infinite values are also allowed, and may be specified as "Inf" or "-Inf", as appropriate. For the optimization problem `myproblem`, the first line of the function would take on one of the following forms:

```
function [fx,cx,gradfx,gradcx] = myproblem(x,p);
function [fx,cx,gradfx,gradcx] = myproblem(x);
function [fx,cx] = myproblem(x,p);
function [fx,cx] = myproblem(x);
function [fx] = myproblem(x,p);
function [fx] = myproblem(x);,
```

where

`fx`	=	objective function value,
`cx`	=	$1 \times m$ vector of constraint function values,
`gradfx`	=	$n \times 1$ gradient of the objective function,
`gradcx`	=	$n \times m$ matrix of constraint gradients,
`x`	=	$n \times 1$ vector of continuous variable values,
`p`	=	cell array of categorical variable values.

Nonlinear constraints are assumed to be of the form $c(x) \le 0$. However, for scaling purposes, it is best to first normalize them. That is, if a constraint is of the form $\hat{c}(x) \le d$, rather than reformulating it as $c(x) = \hat{c}(x) - d \le 0$, it should be posed as $c(x) = \hat{c}(x)/d - 1 \le 0$.

4.2 Initial Points File

This file (`myproblem_x0.m`) has no input arguments and simply returns the initial points as a vector of iterates. The user can put any finite number of iterates in this file. For the default problem, the first line of the function statement would be:

```
function iterate0 = myproblem_x0;,
```

where `iterate0` is a vector of initial points, each of which is an `iterate`.

Multiple initial points must be stored in this manner:

```
iterate(1).x  =  ...
iterate(1).p  =  ...
iterate(2).x  =  ...
iterate(2).p  =  ...
```

This file is not necessary if a parameter file is used, and the initial point is initialized there.

4.3 Closed Constraints File

This file (`myproblem_X.m`) evaluates an iterate and returns a logical value (0-1) indicating whether or not all the closed constraints in this file are satisfied. Like the Functions file, the iterate is broken down into its components, a vector of continuous variables and an optional cell array of categorical variables. For the default problem, the first line of the function statement would be one of the following:

```
function isFeasible = myproblem_X(x);,
function isFeasible = myproblem_X(x,p);,
```

`isFeasible`	=	logical indicating if feasible,
`x`	=	vector of continuous variables,
`p`	=	cell array of categorical variables,

4.4 Omega File

This file (`myproblem_Omega.m`) returns the parameters that define Ω, the feasible region with respect to bound and simple linear constraints, given the number of continuous variables and a vector of categorical variables (if there are any). The file may be omitted if the optimization problem is unconstrained with respect to bound and linear constraints and if the problem has no categorical variables. Specifically, the Omega file returns A, ℓ, and u, where $\Omega = \{x \in \mathbb{R}^n : \ell \leq Ax \leq u\}$ is the feasible region. The function also optionally returns a list of allowable values for all the categorical variables, if any. The user should note that the Omega parameters may be set to change for different values of the categorical variables. This is done by specifying a second input argument. For our problem, `myproblem`, the first line of the Omega file must be written in one of the following three forms:

```
function [A,l,u,plist] = myproblem_Omega(n,p);
function [A,l,u,plist] = myproblem_Omega(n);
function [A,l,u] = myproblem_Omega(n);
```

where

`n`	=	number of continuous variables,
`p`	=	cell array of categorical variables,
`A`	=	matrix of coefficients of the linear and bound constraints,
`l`	=	vector of lower bounds on linear and bound constraints,
`u`	=	vector of upper bounds on linear and bound constraints,
`plist`	=	lists of feasible values for each categorical variable.

The two additional variables `plist` and `p` are for MVP problems. The output variable `plist` is a required cell array with size equal to the number of categorical variables. Each element of `plist` is itself a cell array containing a list of all possible values that the corresponding categorical variable is allowed to take on. Use of the input argument `p` will depend on the problem to be solved. It should only be included if the linear constraints can change for different values of the categorical variables. Otherwise, the optimizer will be slowed considerably.

4.5 Neighbors File

This file (`myproblem_N.m`) returns the discrete neighbors of a `iterate` if the optimization problem has categorical variables; otherwise, the file is omitted. The set of neighbors it returns is a vector, each element of which is an `iterate`. For our default problem, `myproblem`, the function call must be of the form

```
function N = myproblem_N(Problem,iterate,plist,delta);
```

where

`N`	=	vector of neighbors, with each element an `iterate` structure,
`Problem`	=	structure containing data identifying the optimization problem,
`iterate`	=	current iterate,
`plist`	=	cell array of allowable values for each categorical variable,
`delta`	=	mesh size parameter – to make sure a neighbor stays on the mesh.

The Problem structure is typically not used, but it is included as an input argument in case the user wishes to access the values of Omega to ensure that a neighbor point is actually feasible. In this case, the Omega parameters A, ℓ, and u are stored in `Omega.A`, `Omega.l`, and `Omega.u`, respectively.

4.6 Parameter File

This optional file (`myproblem_Param.m`) usually (but not always) has no input arguments, and it returns a structure whose fields contain *any* parameters the user wishes to store there. The `processInput` function within MADS runs this file, storing the structure of output data as workspace application data with the name of `PARAM`. The main reason for doing this is to avoid excessive computational cost. Unlike the functions file which is rerun at every function evaluation, the parameter file is only run once at the beginning of the iteration (actually, it might be run once more at the end). For example, if the objective function involves fitting an interpolating function from tables of data and computing integrals of the resulting functions, many (but clearly not all) of the calculations can be pre-computed in this file and stored in `Param`. This would avoid making these computations every time the Functions file is called, resulting in a substantial savings in computational cost.

In order to actually use the data that are stored as unchanging parameters, the statement,

```
Param = getappdata(0,'PARAM');
```

must be included near the beginning of any function that using the data. This command retrieves all the application data stored in `PARAM` and stores it in the variable, `Param` for use by the calling user function.

In particular, if the fields `.PollBasis` and `.PollOrder` are used, then `Param.PollBasis` will be the matrix whose columns are the basis directions used to construct the $2n$ or $n + 1$ poll directions, and the polling order specified by `Param.PollOrder` will be used through the run.

For the default problem, the first line of the Parameter File function statement would be:

```
function Param = myproblem_Param;
function Param = myproblem_Param(varargin);
```

The `varargin` variable, which indicates a variable number of input arguments, is only used in cases where unique post-processing is needed. In this case, loading parameters is still done exactly as before, but a separate chain of logic is stored inside the `myproblem_Param` function for the post processing, and is only exercised when there are input arguments passed through the function. Code to do this would have to be included within the `myproblem_Param` file, and would typically consist of a simple `if-endif` block, conditioned on the number of arguments, inside of which, the first statement would be the same `getappdata` statement listed above.

If a Parameter File is used, then it can be used to set any initial points in lieu of an Initial Points file. This is done by using the following syntax inside the Parameter File:

```
Param.iterate(1).x  =  ...
Param.iterate(1).p  =  ...
Param.iterate(2).x  =  ...
Param.iterate(2).p  =  ...
```

4.7 Cache, Session, and Debug Files

The Cache file (`myproblem_Cache.mat`) stores all previously computed iterates and their function values, along with a tolerance parameter, some matrices for plotting, and all of the filter parameters, the latter being used to keep track of the current poll center. It is created and reused by NOMADm, as chosen by the user. NOMADm stores all of its previously computed iterates in a structure with the variable name `Cache`, which is stored as workspace application data with the name of `CACHE`. If the cost of evaluating a function is very expensive, saving the Cache structure variable to a file can result in a substantial savings in time and effort if the problem needs to be solved again later.

The Session file (`myproblem_Session.mat`) stores all the MADS and NOMADm options and parameters. It is created and reused by NOMADm, as chosen by the user through Session Menu choices.

The Debug file (`myproblem_Debug.txt`) is a log of messages that also appear on screen during the room. It is activated by an Options Menu choice.

5 The NOMADm Display Screen and Menu Functions

To launch NOMADm, enter “nomadm” at the MATLAB command window prompt. The main display screen consists of a menu bar at the top, a gray center area containing various data in text format, and a run status bar at the bottom. At startup, the gray center area includes a text label, *Optimization Problem* and a light gray *MADS Parameters* box. The *Optimization Problem* field is left blank until a problem has been selected, after which its name will appear in red immediately following this label. The values that appear in the boxes are default values, as stored in the defaults file, `mads_defaults.m`. If the user wishes to permanently change default values, this can be done by editing the defaults file. However, this is *not* recommended, since the user must be well acquainted with the inner structure of the NOMADm code, and once these defaults are changed, they cannot be recovered, except by reinstalling the software from the download site. As an alternative, it is far better to make changes within the NOMADm GUI and use the *Save Session* Options menu choice to save the changed values, so that they can be reloaded later. The various strategies and parameters will be discussed as they come in the description of the main menu functions. Termination criteria can be turned off and on by marking or unmarking the corresponding check boxes in the gray center area.

The remainder of this section describes in detail each choice in the NOMADm menu. Each subsection represents an entry on the NOMADm menu bar. They are designed so that the user will naturally move left-to-right in setting up the problem and options, running the optimizer, and retrieving results.

5.1 Problem

The Problem menu allows the user to specify the optimization problem and certain parameters and edit its files. The menu choices are as follows:

New Optimization Problem: Change the optimization problem from a list. Remember to choose the Functions file.

Edit ... File: Use the MATLAB editor to edit one of the 5 specified optimization problem files.

Exit: Exit the program.

5.2 MADS

The MADS menu allows the user to specify and edit MADS optimizer settings before making an actual run. The menu options are described in the paragraphs that follow.

Select Search Strategies: Select the Search methods from a Search menu. When this option is chosen, a Search Input Screen is launched. This screen allows the user to specify the number N of Search types, and for each of the N searches, the type used, the number of iterations to use it, the number of points to sample, and a checkbox in which the user may specify a "complete" Search, in which *all* Search points are evaluated, as opposed to points being evaluated only until an improvement is found.

Aside from doing no search, the search strategies that NOMADm specifically supports include standard and gradient polling around the first n filter points, Latin hypercube sampling (LHS), a few iterations of a particle swarm (PS) or genetic algorithm (GA), and one of six choices for optimizing a surrogate problem: optimize a DACE Toolbox, Nadaraya-Watson (NW), or radial basis function (RBF) surrogate; or optimize a simplified physics model using DACE, NW or RBF to model the difference between the surrogates and true functions. The GA code, called `cmaes`, was written by folks in Germany, and the DACE Toolbox is a MATLAB code written by folks at the Danish Technical University near Copenhagen, Denmark. The Nadaraya-Watson code was written by the author (with contributions by former graduate students, Todd Sriver and John Dunlap), and the RBF surrogate package was written by the author. All three packages are downloadable using the links at the end of this document, or directly from the NOMADm web site.

If the user selects an option that makes use of surrogates, three additional (optional) columns appear: the name of a surrogate file that specifies which DACE, NW, or RBF function to use, the name of a user surrogate file, and a popup menu to choose if the surrogate is constructed from objective function values or some other parameter. If the user chooses the latter, the user must use a Parameter file, initialize `Param.param = 0` inside the Parameter file, and use the following commands inside the functions file:

```
Param = getappdata(0,'PARAM');
Param.param = ...;
setappdata(0,'PARAM',Param);
```

where the unfinished second line is filled in by the user.

Also on the Search Input screen is a popup menu for choosing which optimizer will be used to numerically solve surrogate optimization problems and, for mixed variable problems, a check box to determine if only one surrogate will be used for the entire run (as opposed to a different surrogate each time a new set of categorical variable values are observed). Only one surrogate optimizer may be used during a run. Choices currently supported by NOMADm are: `fmincon` (from the MATLAB Optimization Toolbox), `mads` (the NOMADm solver), `ps` (a NOMADm internal particle swarm optimizer function) and `cmaes`. Since the Optimization Toolbox is not required to run NOMADm, `fmincon` will not appear as an option if a license for that toolbox is not found.

If the *DACE Toolbox Surrogate*, *Nadaraya-Watson Surrogate*, or *Radial Basis Function Surrogate* option is chosen as a Search type, then a Toolbox Options figure window is launched (called DACE, NW, or RBF, respectively), which contains a number of objects: 1-3 popup menus, three parameter value fields, and some checkboxes. The DACE Toolbox has three popup menus, two of which require the user to specify a regression and correlation function to be used. Each possible choice corresponds to a function in the DACE Toolbox. The third popup menu sets how often the surrogate is to be recalibrated; namely, only in forming the initial surrogate, at every unsuccessful iteration, or at every iteration. The NW and RBF Toolboxes have two popup menus, the first of which requires the selection of a kernel function, and the second indicates how often to recalibrate the surrogate. Within the surrogate recalibration process, a function is often optimized with respect to certain parameters. For DACE, a maximum likelihood estimator problem is solved to optimize a vector of θ parameters used by the correlation function, and for NW, the Nadaraya-Watson estimator is optimized for the σ (bandwidth) parameter. RBF simply computes interpolation coefficients. The DACE and NW figure windows have three input fields in which the user can specify an initial guess, along with lower and upper bounds used in the optimizing these parameters. On the DACE screen, there is also an *Isotropic* checkbox that, if checked, forces all the *Theta* parameters to be the same. Checking this box makes the surrogate less expensive to recalibrate but also less accurate. The DACE screen also contains a *Set Bounds Manually* checkbox that, if checked, will use the bounds provided to optimize theta; otherwise, it computes reasonable values for them. Finally, all three Toolbox figure windows have two other checkboxes for special purposes, one for using a trust region and the other for using a merit function to penalize points that are too close together. If the first checkbox is marked, a trust region is established, such that any point lying outside the region is assigned a very large function value. If the second checkbox is marked a penalty term is added to the Nadaraya-Watson function that penalizes data sites that are close to other points in the model. This tends to reward sets of points that are more spread out in the region. The penalty term is divided in half at every iteration. The DACE and NW Toolboxes both have user guides that should be consulted for further details.

If the user chooses *Custom Search Function* or *Custom Surrogate Function*, then an appropriately structured MATLAB function must be provided and stored in a directory in the MATLAB path, which will perform the SEARCH step. This is an advanced feature, and should be used with caution. When either of these is selected, the user will have to select the file from a list of existing files. Details for setting up such a file are given in Section **??**.

If the user chooses *SPS with DACE Surrogate*, *SPS with Nadaraya-Watson Surrogate*, or *SPS with Radial Basis Function Surrogate*, then elements of the previous three descriptions will occur. First, the user is prompted to select from a list the user-provided file containing the simplified physics simulation (SPS) model. Once selected, the DACE, N-W, or RBF screen will appear, so that the appropriate options may be stored. The DACE, Nadaraya-Watson, or RBF surrogate is constructed, based not the true function values, but on the difference between the true function and the SPS model. In order for either option to work properly, any user-provided function must reside in a directory on the MATLAB path, such as the one containing the optimization problem files or the NOMADm files.

Select Poll Directions: This option allows the user to select which directions to poll in. This is the option that separates MADS from GPS. The convergence theory requires the set of polling directions to form a positive spanning set; *i.e.*, any vector in the solution space may be represented as a nonnegative linear combination of the vectors in the set. For example, the "Standard $2n$ directions" refer to the Euclidean coordinate directions and their negatives, while the "Standard $n{+}1$ basis" refers to the Euclidean coordinates, plus the vector whose elements are all -1. In MADS polling, the directions are selected randomly from among a finite set of directions sufficiently close to the poll center. In the limit, the set of directions from which to select these directions becomes asymptotically dense. In gradient-pruned polling, the gradient is used to determine the non-ascent directions and only poll in those directions until an improvement is found. In the case of the 3^n directions, which are all the vectors whose elements are 1, 0, or -1 (which we would never exhaustively poll), we can find a subset of this set that is also a positive spanning set,and which contains 1 descent direction and n ascent directions. This results in only having to poll in one direction. The L-1, L-2, and L-Inf notation refers to which direction is polled in. L-1 and L-Inf are the actual $\ell_1(n)$ and $\ell_\infty(n)$ gradients, respectively, as they belong to the $\{-1, 0, 1\}^n$ set, while L-2 is simply the direction in $\{-1, 0, 1\}^n$ that makes the smallest angle with the (classical) $\ell_2(n)$ gradient. The "Descent Vectors" option is a combination of the Gradient-pruned 3^n L-2 and $2n$ directions. In bound or linear constraints are ε-active at the current poll center, then these three options are modified slightly to include appropriate tangent cone generating directions which are necessary to guarantee convergence to a solution. Also, slight modifications are made in the case where some, but not all, partial derivatives are available.

Select Poll Center: Select which point to poll around during MADS execution. If there are no nonlinear constraints, the current incumbent (best feasible) solution is used. If the user picks a poll center that is not available, NOMADm defaults to the closest option to the user's choice. For example, if the user wants to poll around the 3rd least infeasible point, but there are only two points in the filter, then the poll is conducted around the 2nd least infeasible point.

Select Polling Order: Select the order in which points in the poll set are evaluated. *Consecutive* order is the natural order that the points are stored in. *Alternating* order consists of polling in each direction and then in its negative; *e.g.*, e_1, $-e_1$, e_2, $-e_2$, ... *Random* order is a random permutation of the natural order. *Dynamic* means that the direction that was successful (or the direction that makes the smallest angle with the one that was successful) in the last successful iteration is polled first, followed by the rest of the directions in Consecutive order. *Dynamic Ranked* means that the directions are placed in order of how small an angle each makes with the last successful direction. *Simplex Gradient* means that directions are ordered according to how small an angle each makes with the negative simplex gradient, which is computed at each iteration if this option is selected. *Surrogate Ranked* means that the poll points are evaluated first by the surrogate function and then ranked in order of their surrogate function value (smallest-to-largest). This option is only available if a surrogate is used in the optimization process. *Custom* means that a user-specified fixed poll order is provided in the Parameter file, using the variable `Param.order`.

Complete Polling: Turn on/off complete polling flags for the *Poll step*, *Discrete Neighbor Poll*, and *Extended Poll step*. Complete polling means that all the points in the specified poll set are evaluated during each poll step, even if an improvement is made. If a poll type is not selected, that poll is performed only until an improved mesh point is found, at which time, the iteration ends and the corresponding point immediately becomes the new incumbent.

Edit Termination Parameters: Edit the MADS termination control parameters that appear in the second panel of the display window. *Convergence Tolerance* is the positive number such that the MADS run is terminated as soon as the mesh size parameter falls below this value. *Maximum Number of Iterations*, *Maximum Number of Function Evaluations*, *Maximum CPU Time*, and *Max Number of Consecutive Poll Failures* are all additional termination criteria that can be applied. Each of these criteria is activated by marking the corresponding checkbox in the gray area of the main figure window. Unmarking a checkbox is equivalent to setting the corresponding criterion to infinity. The MADS run is terminated as soon as the first selected criterion is met. If no checkbox is marked, NOMADm generates an error message and stops running.

Edit Mesh Parameters: Edit the mesh parameters shown in the third panel of the display window. *Initial Mesh Size* refers to initial value of the mesh size parameter, which must be positive. *Maximum Mesh Size* is the maximum value that the mesh size may attain, regardless of other parameter settings. If an iteration is successful in finding an improved mesh point, then the mesh is coarsened (provided that the maximum mesh size is not exceeded) by multiplying the mesh size parameter by the *Mesh Coarsening Factor*, which must have a value greater than or equal to one. If the iteration fails to find an improved mesh point, it is declared to be a mesh local optimizer, and the mesh is refined by multiplying the mesh size parameter by the *Mesh Refining Factor*, a number between 0 and 1. The *Cache Tolerance* is the parameter by which the algorithm distinguishes distinct points. If the distance between a trial point and any point in the Cache is computed to be within this value, the trial point is considered to be the same as the Cache point, and the trial point is not evaluated.

Edit Filter/MVP Parameters: Edit the Filter and MVP parameters shown in the fourth panel of the display window. If the optimization problem has nonlinear constraints, the *Min Filter Constraint Violation* is a small positive number which acts as a feasibility threshold. If an iterate's constraint violation function value is smaller than this number, the iterate is treated as feasible; otherwise, it is infeasible. If an iterate's constraint violation function value exceeds the *Max Filter Constraint Violation*, a positive number that filters out as unsuccessful any iterate whose aggregate constraint violation exceeds this number. If the constraints are normalized, as discussed at the end of Section **??**, then this number represents the percent of constraint violation. For example, a value of 1.0 would indicate a 100% violation.

For mixed variable problems, if a discrete neighbor is feasible and has an objective function value that is greater than, but close to, that of the current best feasible point, then an Extended Poll step is performed. Similarly, if a discrete neighbor is infeasible and has a constraint violation function value that is greater than, but close to, that of the current least infeasible point, then an EXTENDED POLL step is performed. The *MVP Objective Extended Poll Trigger* and the *MVP Constraints Extended Poll Trigger* are, respectively, the tolerances that trigger an EXTENDED POLL. These values are expressed as percentages of the current objective or constraint violation function value. For example, if the objective function f has a value of 5 at the current point x, then an extended poll trigger $\xi_f = 0.01$ means that extended polling would occur around any discrete neighbor point y satisfying $f(x) \leq f(y) < f(x) + \xi_f f(x)$, or $5 \leq f(y) < 5.05$.

Edit Ranking & Selection Parameters: Edit the Ranking and Selection parameters shown in the (sometimes invisible) fifth panel of the display window. Since these parameters are only valid for stochastic problems, the panel is only made visible and the menu choice made available if the appropriate option (described later) for running a stochastic problem is selected. The *R & S Initial Sample Size* is the initial number of points sampled in the ranking and selection process. An initial sample is needed to compute variances. The ranking and selection scheme requires two other parameters, the *alpha* parameter, which controls the significance level of statistical tests, and the indifference zone parameter, which controls how close two function values can be to each other for the R & S procedure to treat them as equal. The *R & S Initial Alpha Parameter* and *R & S Initial Indifference Zone Parameter* are the initial settings for the respective parameters, and *R & S Alpha Decay Factor* and *Indifference Zone Decay Factor* are the rates at which the respective parameters converge to zero (a condition required by the theory to ensure convergence).

5.3 Options

The Options Menu allows the user to specify a few additional variables that are simple on/off switches, except for scaling, which offers a choice of base-2 or base-10 logarithmic scaling. Selecting one of these options toggles on and off a check mark that appear next to the menu text. The choices are described as follows:

Scaling of Mesh Directions: Scale the mesh directions relative to their bounds or the initial iterates. This option has a second popup menu for choosing the type of scaling (none, base-2 logarithmic, or base-10 logarithmic).

Filter for Nonlinear Constraints: Select a filter (none, traditional, or 2-point) for handling nonlinear constraints.

Run as Sensor Placement Problem: Run as a sensor placement problem with 1-D, 2-D, or 3-D sensor locations.

Scheme for Handling Degenerate Linear Constraints: Select a strategy for selecting a subset of constraints from which to compute generators for a relaxation of the tangent cone, in the event that nonredundant ε-active linear constraints are linearly dependent. Current choices are: Proximity to constraint boundary, Random selection, Sequential selection, and Full enumeration. These strategies are fully described in [**?**].

Discard Redundant Linear Constraints: Flag to detect and discard redundant linear constraints before beginning the iteration.

Run as Stochastic Optimization Problem: Flag for running MADS on a stochastic optimization problem; *i.e.*, problems whose objective and nonlinear constraint functions have random noise.

Accelerate Convergence: Accelerate convergence by reducing the mesh refinement factor at every unsuccessful iteration.

Use Relative Termination Tolerance: Scale the Convergence Tolerance by the initial mesh size.

Print Debugging Messages to Screen: Print status messages to the screen to help with debugging. Current choices of *Off*, *Level 1*, *Level 2*, and *Level 3* are in increasing order of output.

Save History to Text File: Saves the performance history (objective function value and iterate coordinates) to file for post-processing.

Plot History: Submenu for plotting objective function value versus cumulative number of function evaluations. Allows plotting up to 10 runs. Current choices are *Off*, *On* (one plot at the end of each run), and *Real-Time* (plot updated at each iteration).

Plot Filter (Real-Time): Real-time plot of the filter – objective function vs. an aggregate constraint violation function.

The scaling of mesh directions is done by taking each direction, one at a time, and multiplying each element of the direction vector by the corresponding element of a vector of scale factors. The vector of scale factors is simply a vector whose elements each represent "typical" values for the continuous variables. To compute such a value for an individual variable, NOMADm first looks for an adequate lower and/or upper bound on the variable. If at least one exists and is nonzero, then the "typical" value is computed by taking the base-2 or base-10 logarithms of the bounds, averaging them, and then exponentiating the result using 2 or 10, respectively, as the base. For example, if a variable has bounds of 10^{-4} and 10^{-2}, then the base-10 scale factor would be 10^{-3}. If the variable bounds do not exist or are zero, then the scale factor is chosen as the initial value of that variable, the assumption being that the initial iterate is representative of a "typical" value of the variable. If the initial value is zero, then no scaling is used because the there is no frame of reference from which a "typical" value can be ascertained.

For mixed variable problems with changing linear constraints, similar rules apply, except that the scale factors must be updated each time the linear constraints change. If the new bounds don't exist or are zero, then the scale factors are determined by searching the Cache for the first iterate having the same set of discrete variable values as the current iterate. If no such Cache point is found, the current iterate is used as typical values.

5.4 Session

The Session menu allows the user to save, retrieve, and reset all of the NOMADm options. The choices are described as follows:

Load Session Options: Load user options from a pre-existing file.

Save Session Options: Save currently selected user options to file.

Reset Session Options to Defaults: Restore the default values for all MADS options.

Delete Session File: Delete the file that stores the user options.

5.5 Cache

The Cache menu allows the user to manage the Cache, a database that contains data on all previously computed iterates. This is beneficial if function evaluations are computationally expensive, in that it prevents re-evaluation of previously computed iterates. Choices are as follows:

Use Existing Cache: Load a pre-existing Cache file prior to executing a run.

Count Cache as Function Calls: Include Cache iterates when counting the number of function evaluations.

Save Problem Cache: Save to file the Cache for the most recent run for possible later use.

Delete Pre-existing Cache File: Deletes the Cache file associated with the current optimization problem.

5.6 Run

The Run menu allows the user to actually run the optimizer on the problem that has been specified and under the options that have been set. The choices for the various Run functions are described as follows:

Execute Next Run: Executes the next MADS run, given all the user-specified options.

Resume a Stopped Run: Restarts the current run at the current best solution with the same cache and parameter settings.

Restart Run from Current Point: Restarts the current run at the current best solution, but with a new empty Cache.

Run One Iteration at a Time: Runs one iteration of MADS at a time, requiring the user to resume the run between iterations. Parameters may be changed between iterations.

Run Only Until Feasible: Runs MADS, but stops as soon as a feasible point is found.

Clear Previous Runs: Clear the screen and reset all data (done automatically when a new problem is selected).

5.7 Results

The Results menu is not visible until after at least one run has been completed. Then the menu choices are simply to view results from a sequentially numbered run. If the real-time history plot is displayed, the text in the menu is colored the same as the corresponding curve in the plot.

When a particular run is selected, a solution screen is displayed with two small panels at the top and one large one at the bottom. The two at the top show the objective function value and constraint violation function value for both the best feasible solution and the least infeasible solution, if they exist, along with a "View Solutions" push button. If no infeasible points are generated, then the right panel will be blank, and if no feasible points are generated, the left hand-side will display only a message saying so. Pressing a "View Solutions" push button launches a Solutions display window, in which case, continuous and categorical variables (if appropriate) for the chosen solution are displayed by index number, as stored in the original problem.

The third panel displays run statistics; specifically, the final poll size, number of iterations required, the number of ¶steps executed, number of consecutive poll failures, number of function and gradient evaluations performed, CPU time required, number of Cache hits, a Yes/No flag indicating whether or not the run was interrupted by the user (via the Stop Run button), and the R & S indifference zone and significance parameters.

5.8 Help

The Help menu simply displays in a web browser documentation for this program, the DACE Toolbox, NW Toolbox, and the CMA-ES Genetic algorithm code.

NOMADm Help: Display this file.

DACE Toolbox Help: Display DACE Toolbox documentation.

NW Toolbox Help: Display NW Toolbox documentation.

CMA-ES Toolbox Help: Display CMA-ES Toolbox documentation.

View List of Version Changes: Display the list of changes in recent versions of NOMADm.

View GNU Public License: Display the GNU General Public License.

About: Display basic information about this program.

6 Custom Searches and Surrogates

This section describes the details of setting up custom searches and surrogate functions, so that the functions are compatible with the NOMADm software. In both cases, the user will have to look at the `mads` source code to fully understand the input parameters. Any search or surrogate function file used in this process must be placed in a directory that is on the MATLAB path, such as the one containing the optimization problem files, or the NOMADm directory.

6.1 Custom Searches

For a custom search file called `mySearch`, the first line of the file (function statement) must be in the form,

```
S = mySearch(Problem,Options,delta,pcenter);
```

where

`S`	=	set of iterates to be evaluated,
`Problem`	=	structure containing the names of the optimization problem files,
`Options`	=	structure containing the selected MADS parameter values,
`delta`	=	current mesh size,
`pcenter`	=	coordinates of the current poll center.

6.2 Custom Surrogates

Custom surrogates can be used in two different ways. In the first case, the user already has MATLAB functions for building (recalibrating) and evaluating the surrogate. The user then only needs to provide a simple interface file. For an interface file called `mySurrogate`, the first line must be in the form,

```
[surrogate,param] = mySurrogate(n);
```

where

`surrogate`	=	structure containing information for the surrogate functions,
`param`	=	cell array containing all parameters needed by the surrogate.
`n`	=	number of continuous variables

The variable `surrogate` is a structure that must contain at least two fields, `recalibrator` and `evaluator`, which are strings containing the filenames to build and evaluate, respectively, the surrogate. The variable `param` is a cell array in which all other parameters that are needed by the surrogate are stored.

For example, for the DACE package, which is already a separate option, the function `dacefit` is used to build the surrogate, and the function `predictor` is used to evaluate the surrogate at a point (or set of points). The function `dacefit` also uses three parameters, a vector called `theta` and two object handles for regression and correlation functions. For the regression function, `regploy0` and correlation function, `correxp`, the interface file might look like this:

```
[surrogate,param] = mySurrogate(n)
surrogate.recalibrator = 'dacefit';
surrogate.evaluator = 'predictor';
reg = str2func('regpoly0');
corr = str2func('correxp');
theta = ones(n,1);
param = {reg,corr,theta};
return
```

Once this file is called in NOMADm, and the `surrogate` variable is established, the NOMADm function `recalSurrogate` builds the surrogate, adding fields to the `surrogate` structure, as appropriate. Then the `optimizeSurrogate` function is called, in which one of three optimizers is applied to the surrogate problem; namely, fmincon (from the MATLAB Optimization Toolbox), mads (the NOMADm optimizer), and cmaes (the CMA-ES Toolbox). Each of these packages will input the surrogate model as the function to be optimized and will return a specified number of "optimal" points, as specified in the appropriate *Number of Points* field in the Search Input Screen. These points are returned in the variable `S`, a vector of `iterates` to be evaluated by the original problem functions. Currently, the recursive use of `mads` as the surrogate optimizer is remarkably slow (an area to be addressed in future versions), and `fmincon` only returns one point, regardless of how many points are specified in the Search input screen.

In the second case, the user provides everything, including the optimizer. In order to use this option, "Custom" must be must selected from the *Surrogate Optimizer* popup window in the Search Input Screen. For a custom surrogate file called `mySurrogate`, the first lines of these file must be in the form,

```
S = mySurrogate(Problem,Cache,Search,surrogate,optimizer,tol);
```

where

`S`	=	set of iterates to be evaluated,
`Problem`	=	structure containing the names of the optimization problem files,
`Cache`	=	vector containing all previously processed iterates,
`Search`	=	structure containing current Search information,
`surrogate`	=	structure containing information for the surrogate functions,
`optimizer`	=	string containing the name of the optimizer executable file,
`tol`	=	tolerance for how accurate the optimizer will solve the surrogate problem.

In this case, the entire surrogate process, including construction, calibration, and optimization is managed by the user, and the `mySurrogate` function returns a set S of `iterates` to be evaluated by the true problem functions. Inside the `mySurrogate` function, the user may or may not wish to use the variables passed as input arguments. Some of these are fairly complex structures. The user should consult the documentation within the source code to get a description of the various fields these structures use.

7 Interfacing with a FORTRAN or C Functions File

In many optimization problems, the objective and constraint functions may use or call a function that is written in FORTRAN or C. NOMADm can be used to solve such problems as long as the user provides the FORTRAN or C file in a certain format. Specifically, the first line of the FORTRAN function (called `MYPROBLEM`) and the C function (called `MyProblem`) must be in the following forms, respectively:

```
SUBROUTINE MYPROBLEM(NP1,NP2,NX,P1,P2,X,NC,F,C,DF,DC)
void Function MyProblem(np1,np2,p1,p2,x,nc,f,c,df,dc)
```

where

`NP1`	=	number of categorical variables stored as integers,
`NP2`	=	number of categorical variables stored as character,
`NX`	=	number of continuous variables,
`P1`	=	current iterate's integer categorical variable values,
`P2`	=	current iterate's character categorical variable values,
`X`	=	current iterate's continuous variable values,
`NC`	=	number of nonlinear constraints,
`F`	=	objective function value,
`C`	=	row vector of constraint function values,
`DF`	=	column vector of partial derivatives of the objective function,
`DC`	=	matrix of constraint function gradients,

Here, the lower case variables in the C statement above match the upper case variables in the FORTRAN statement, so that the variable descriptions can remain the same.

To select this problem in NOMADm, first choose the *New Optimization Problem* option from the *Problem* Menu. When the resulting window appears, choose *All Files (*.*)* from the *Files of Type* popup menu. This will make the FORTRAN or C source file visible in the window, so that it can be selected. Editing of this file can be done exactly like a MATLAB file – by clicking on the *Edit Functions File* option.

In order to run MADS on a problem with a FORTRAN or C functions file, NOMADm (actually MATLAB) requires a compiled version of the functions file. This file is platform dependent and must be compiled by the MATLAB mex utility. Details on how to do this can be found in downloadable documents from the MathWorks web site. However, if the user has a FORTRAN or C compiler that is compatible with the MATLAB mex utility, NOMADm will automatically do all the required processing so that the MADS run can proceed. Specifically, NOMADm first looks to see if the proper compiled functions file exists. If not, it generates the required mex-FORTRAN gateway source code (called `MYPROBLEMg.for`), or the similar code in C into the proper format, and executes the mex command to produce a MATLAB-compatible, platform-dependent executable file.

Note: Before using a particular compiler within MATLAB, the mex setup routine must be run. This is a simple script that is invoked by typing "`mex -setup`" at the MATLAB prompt, and a simple set of choices follows. The desired compiler can be selected from a list. What actually happens is that the mex setup routine simply copies a pre-scripted file associated with a particular compiler into a file, named `mexopts.bat` (or similar name for non-Microsoft Windows platforms) that lies in a working directory. More details can be obtained by typing "help mex" at the MATLAB prompt. The mex setup routine can be rerun at any time to change compilers. Unfortunately, if the user has many different files written in different languages, the mex setup routine is still the only reasonable way to change compilers without making the code platform-specific.

8 Additional Resources for Help

In addition to the specific references cited in this paper, additional papers and resources for help in understanding the algorithms described here can be found on the following web sites:

NOMADm: `http://en.afit.edu/ENC/Faculty/MAbramson/nomadm.html`.

NW Toolbox `http://en.afit.edu/ENC/Faculty/MAbramson/nomadm.html`.

RBF Toolbox `http://en.afit.edu/ENC/Faculty/MAbramson/nomadm.html`.

DACE Toolbox: `http://www.imm.dtu.dk/ hbn/dace/`.

CMA-ES software: `http://www.bionik.tu-berlin.de/user/niko/cmaes_inmatlab.html`.

Acknowledgments

The author wishes to thank John Dennis, Charles Audet, and Gilles Couture for many helpful suggestions and ideas over the past few years, and Keith Berrier and Thierry Dalon for teaching me so much about MATLABand for countless wonderful suggestions for improving the code. Also thanks to Olga Brezhneva, John Dunlap, Aran Garcia-Lekue, Alison Marsden, Juan Meza, Rachael, Pingel, Jacob Sondergaard, and Rakesh Kumar (Mathworks, Inc.) for various contributions with respect to ideas and code.