HOKKAIDO UNIVERSITY

Title	Introduction to MIDACO-SOLVER Software
Author(s)	Schlueter, Martin; Munetomo, Masaharu
Relation	Information Initiative Center, Hokkaido University
Issue Date	2013-03-29
Doc URL	https://hdl.handle.net/2115/52782
Rights(URL)	https://creativecommons.org/licenses/by-nc-sa/2.1/jp/
Type	learning object
File Information	MIDACO_Report.pdf

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

Introduction to MIDACO-SOLVER Software

Martin Schlueter, Masaharu Munetomo

Information Initiative Center,
Hokkaido University
Sapporo, Japan

schlueter@midaco-solver.com
munetomo@iic.hokudai.ac.jp

May 29, 2013

Abstract

MIDACO is a general-purpose software for solving mathematical optimization problems. The software implements an extended ant colony optimization algorithm, which is a heuristic method that stochastically approximates a solution to the mathematical problem. MIDACO can be applied on purely continuous (NLP), purely combinatorial (IP) or mixed integer (MINLP) optimization problems. Problems may be restricted to nonlinear equality and/or inequality constraints. The objective and constraint functions are treated as black box by MIDACO. This user guide provides essential information on understanding the MIDACO screen and solution file output, how to adopt a problem to the MIDACO format and how to solve it with MIDACO (including parameter tuning and parallelization options).

1 Introduction

MIDACO is a heuristic solver for mixed integer nonlinear programming (MINLP) problems. The mathematical formulation of a general MINLP is stated below in (**1**), where $f(x,y)$ denotes the objective function to be minimized and $g_i(x,y)$ represents the vector of equality and inequality constraints. The components of the vector x are the continuous decision variables and the components of the vector y are the discrete decision variables. Furthermore, some box constraints as lower bounds x_l , y_l and upper bounds x_u , y_u on the decision variables x and y are stated in (**1**). MIDACO considers the function $f(x,y)$ and $g(x,y)$ as black box and does not require them to hold any particular property (like convexity, smoothness or differentiability).

$$
\begin{array}{lll}
\text{Minimize} & f(x,y) & (x \in \mathbb{R}^{n_{con}},\ y \in \mathbb{Z}^{n_{int}},\ n_{con},\ n_{int} \in \mathbb{N}) \\
 & & \\
\text{subject to:} & g_i(x,y) \ = \ 0, & i \ = \ 1,...,m_e \in \mathbb{N} \\
 & g_i(x,y) \ \geq \ 0, & i \ = \ m_e+1,...,m \in \mathbb{N} \\
 & x_l \leq \ x \ \leq x_u & (x_l,\ x_u \in \mathbb{R}^{n_{con}}) \\
 & y_l \leq \ y \ \leq y_u & (y_l,\ y_u \in \mathbb{N}^{n_{int}})
\end{array}
\tag{1}
$$

Note that in the MIDACO software the distinction between continuous variables and discrete variables is not indicated by the name (x or y). In the MIDACO software only one vector of decision variables is considered and called X. The first entries of this vector represent the continuous variables, while the last entries represent the discrete (also called integer or combinatorial) variables. See Section **3** Figure **4** for an example on the distinction between variable types in X.

The MIDACO algorithm is based on an evolutionary metaheuristic called Ant Colony Optimization (ACO), which was extended to the mixed integer search domain in [7]. The ACO algorithm in MIDACO is based on a so called multi-kernel gaussian probability density functions (PDF), which generate samples of iterates (called *Ants*). For integer decision variables, a discretized version of the PDF is applied. Figure **1** illustrates a Gauss PDF with three individual kernel PDF's.

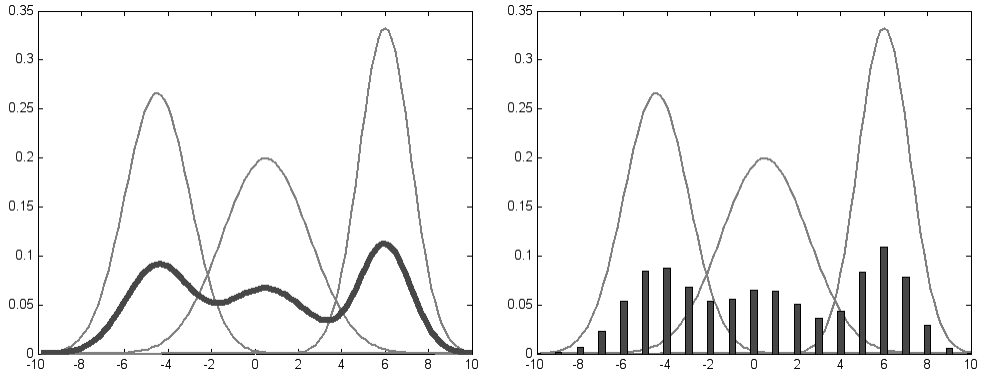

Figure 1: Continuous (left) and discretized (right) multi-kernel Gauss PDF

Constraints are handled within MIDACO by the recently introduced Oracle Penalty Method. This is an advanced method that was developed especially for heuristic search algorithms (like ACO, GA or PSO). This method aims on finding the global optimal solution by using a parameter called *Oracle* (or *Omega* in [9]), which corresponds directly with the objective function value $f(x, y)$. The method is self-adaptive and therefore MIDACO can also be classified as a self-adaptive algorithm. Figure **2** illustrates the shape of the extended oracle penalty function depending on the objective function value $f(x, y)$ and the residual value $res(x, y)$, which represents the constraint violation of $g(x, y)$.

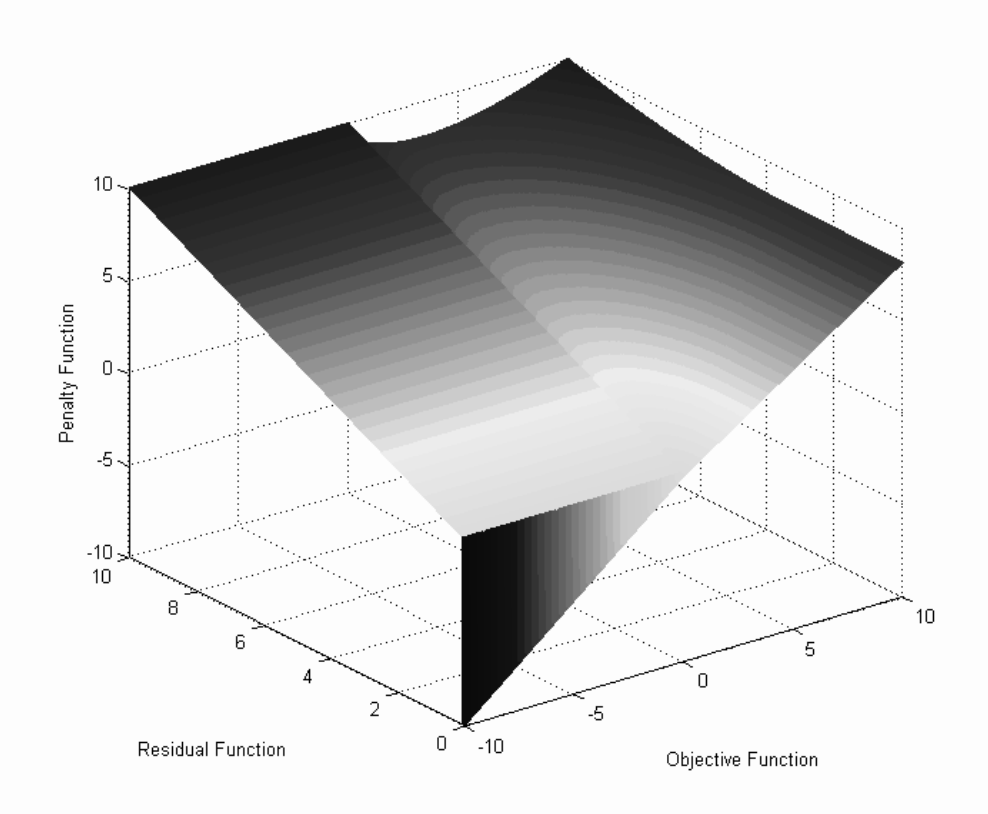

Figure 2: Shape of the extended oracle penalty function

The scope of this user guide is to provide practical information on how to solve an optimization problem with the MIDACO software. Readers who have a deeper interest in the theoretic details of the ACO algorithm within MIDACO can find more information in several publications (e.g. [7] or [8]), whereas [11] provides the most comprehensive and detailed explanation (including a simple step by step example). Detailed information on the development and properties of the oracle penalty method can be found in [9].

As the class of MINLP problems cover purely continuous (NLP) and purely integer (IP) problems, MIDACO can be applied on a wide range of optimization problems. Successful applications of the MIDACO algorithm have been conducted for example on interplanetary space mission planning [12], [3], design and control of launch vehicles [12], satellite constellations [14], chemical plant layout [8], waste water treatment [8], optimal camera placement in robotics [4], distance-to-default models in finance [1], power allocation in wireless networks [2], structural optimization of submarines [15] or parameter optimization in Bio-Technology [5].

MIDACO implements several heuristics to allow the search algorithm to escape from local optima and explore the entire search space in order to find the global optimum. However, like all heuristic algorithms, MIDACO can not provide an absolute guarantee for reaching the global optimal solution. The main motivation behind MIDACO is to provide a robust software tool that can optimize complex real world applications in a reasonable time to a reasonable good solution. Extensive numerical test show (see [10] or [9]), that MIDACO is able to obtain global optimal solutions for easy to medium difficult MINLP benchmark problems mostly within Seconds or Minutes. A collection of general global optimization problems (including well known NLP benchmarks like Rosenbrock or Rastringin) that can be solved by MIDACO are available at the MIDACO benchmark website. Please note that the MIDACO runtimes and capabilities (e.g. number of variables or number of constraints) considered at the MIDACO benchmark website are at the state of the art for evolutionary computing.

For problems that are cpu-time expensive (this means, the evaluation of the objective and/or constraint functions requires a significant amount of time), MIDACO offers an efficient parallelization strategy: MIDACO allows to evaluate the problem functions for several solution candidates in parallel. This strategy can significantly reduce the overall optimization time. The parallelization strategy in MIDACO is implemented by reverse communication which is a very robust and portable concept. Due to this portability, MIDACO can offer its parallelization strategy in several programming languages for several architectures, including C/C++ (openMP, openMPI, GPGPU), Matlab (parfor) and Python (multiprocessing, openMPI). Examples of MIDACO running in parallel mode can be found on the MIDACO parallelization website.

This user guide is structured as follows: In Section **2** all elements of the MIDACO output are explained and information on the PRINTEVAL parameter are given. In Section **3** the problem format as it is assumed by MIDACO is explained in detail. In Section **4** the different available stopping criteria for MIDACO are described. In Section **5** the available parameters to tune the MIDACO performance are discussed. In Section **6** some remarks on the parallelization options for MIDACO are given. In Section **7** a complete list of ILFAG messages is displayed.

This user guide assumes that the reader has already successfully downloaded and executed one of the small example problems that are distributed together with the limited MIDACO version, available here. Running these examples should be straight forward. However, if you experienced a problem nevertheless, please consult the MIDACO troubleshooting website or contact the authors directly. Answers to specific questions can also be found on the MIDACO FAQ website.

2 MIDACO Screen and Solution File

MIDACO produces two output files:

MIDACO_SCREEN.TXT and MIDACO_SOLUTION.TXT

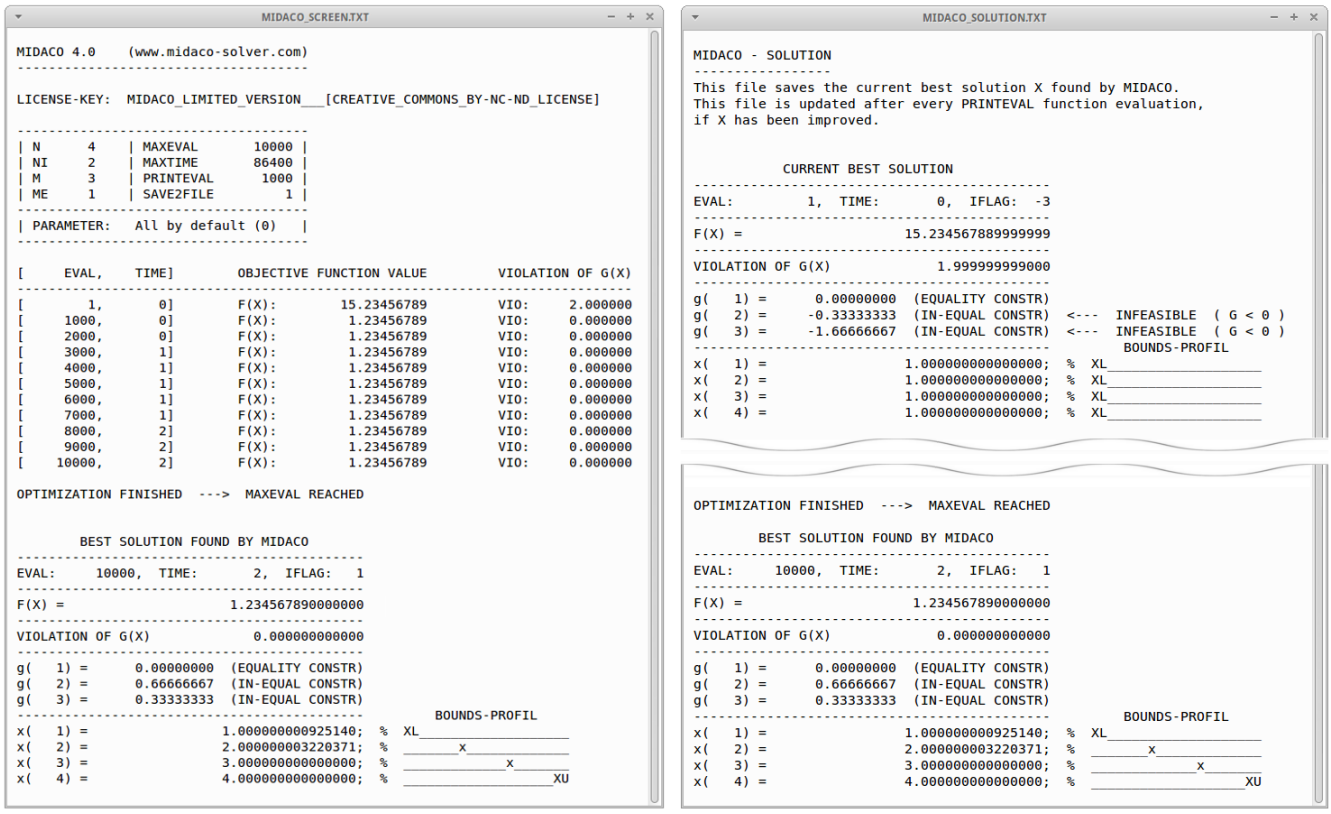

Figure 3: MIDACO Screen and Solution File

Abbreviations used in the MIDACO screen and solution file are explained in Table **1**.

The MIDACO screen file is identical to the output displayed on the console/comand window. The MIDACO screen and solution file layout is identical in all programming languages.

Table 1: Abbreviations used in the MIDACO screen and solution output files (Figure **3**)

N	Number of variables in total
NI	Number of integer variables $0 \leq \text{NI} \leq \text{N}$
M	Number of constraints in total
ME	Number of equality constraints $0 \leq \text{ME} \leq \text{M}$
MAXEVAL	Maximum number of function evaluation (stopping criteria, see Section **4**)
MAXTIME	Maximum cpu-time budget for execution (stopping criteria, see Section **4**)
PRINTEVAL	Print frequency of the current best solution
SAVE2FILE	Create text-file output [0=No, 1=Yes]
PARAM	Parameter for MIDACO tuning (default = 0, see Section **5**)
EVAL	Number of performed function evaluation
TIME	Number of performed cpu-time Seconds
F(X)	Current best objective function value, found after EVAL function evaluation
VIO	Violation of constraints: measured as L1-Norm (Wikipedia) over vector G
IFLAG	Information flag used by MIDACO to indicate final status, warnings or errors
G(i)	Numerical value for individual constraint G_i
X(i)	Numerical value for individual solution variable X_i

The BOUNDS-PROFIL is a graphical (ASCII) illustration of the relative position of X(i) regarding its lower (XL(i)) and upper (XU(i)) bound. If X(i) is closer than 0.1% to the lower or upper bound, the BOUNDS-PROFIL entry will display an upper-case 'XL' or 'XU' respectively, otherwise a lower-case 'x' is displayed.

The solution file contains the numerical values of the solution X for every iteration line printed on the screen. This means, the solution file is constantly updated after every PRINTEVAL function evaluation. Additionally, the very first solution (also called starting point, EVAL=1) and the final solution are displayed. The solutions are stored one after another. The BOUNDS-PROFIL is displayed for every solution stored in the solution file. Furthermore, all constraints G(i) are displayed individually. If a constraint is infeasible, it is highlighted by 'INFEASIBLE (G<0)' (for inequality constraints) or 'INFEASIBLE (G NOT=0)' (for equality constraints).

Note that X in the solution file is not updated, if X has not improved between two printing iterations. This is done to avoid unnecessary storage waste and to keep the file compact.

PRINTEVAL is the critical parameter to control how often the current best solution is printed on the screen. Note that this parameter does not correspond with any algorithmic iteration within MIDACO. Therefore the user can freely set PRINTEVAL in such a way, that the output frequency is convenient for him/her. Small values (e.g. 10, 100, 500) for PRINTEVAL will result in a faster output frequency. Large values (e.g. 10000, 100000) will result in a slower output frequency. The fastest possible output frequency is given by PRINTEVAL = 1, which means that after every evaluation the current best solution found by MIDACO is displayed. This option is only useful for very time intensive problems, or for debugging purposes. In general it is recommended, to set large values for PRINTEVAL (see the screenshot on the MIDACO benchmark website). This way

the user gets a better overview on the optimization progress and MIDACO runs a little bit faster (because the printing commands are less often executed).

The creation of the output files is optional. If SAVE2FILE is set to zero, no output file will be generated. If no output at all is desired (for example if MIDACO should be silently embedded within another software), all visual output can be suppressed by setting PRINTEVAL to zero.

3 The MIDACO problem format

This section explains how an optimization problem must be presented to MIDACO. In case of mixed integer problems, where continuous and discrete variables are simultaneously present, the continuous variables are stored first, the discrete ones last in the solution vector X. The distinction between equality and inequality constraints is handled the same way: The equality constraints are stored first, the inequality constraints are stored last in the constraints vector G. As example, considere a constrained mixed integer problem with the following dimensions:

N	=	10	M	=	5
NI	=	4	ME	=	3

The distinction between continuous and integer variables is illustrated in Figure **4**.

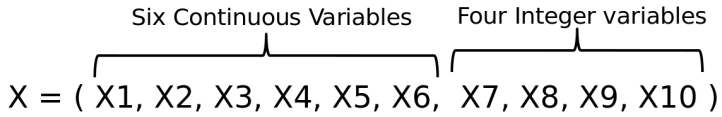

Figure 4: Continuous and integer decision variables stored in X

The distinction between equality and inequality constraints is illustrated in Figure **5**.

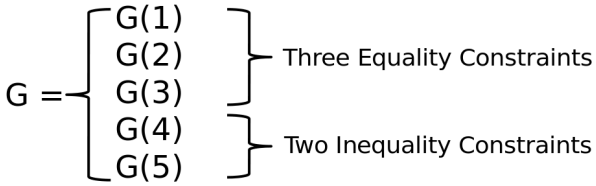

Figure 5: Equality and inequality constraints stored in G

Some lower and upper bounds (XL and XU) for the decision variables X must be provided for any problem. A starting point X (also called initial solution or initial point) must be provided as well, however this can be any point (vector of decision variables X) that lies in between XL and XU. By default, the lower bounds are assumed as starting point in all example problem displayed on the MIDACO website. In general it is recommended, to keep the search space (defined by

XL and XU) as small as possible, as MIDACO will explore the entire search space (Hint: Use the BOUNDS-PROFIL to identify, where a reduction of the search space might be possible). In contrast to this, the starting point is normally not a critical issue for MIDACO.

3.1 Problem Function Call

The user needs to provide a function call to the problem equations (or some black-box library) which evaluates the objective function F and the constraints G for a solution candidate X. In the example problems distributed on the MIDACO website, those problem function calls are given (depending on the language) as:

Matlab	:	[f, g] = problem_function(x)
Python	:	problem_function(x) (return f, g)
C/C++	:	problem_function(*double* *f, *double* *g, *double* *x)
Fortran	:	PROBLEM_FUNCTION(F,G,X)

Figure **6** illustrates the problem function call of the example_MINLPc in Matlab/Octave.

```
%%%%%%%%%%%%%%%% Problem Dimensions of example_MINLPc %%%%%%%%%%%%%%%%%%%

problem.n  = 4; % Number of variables (in total)
problem.ni = 2; % Number of integer variables (0 <= nint <= n)
problem.m  = 3; % Number of constraints (in total)
problem.me = 1; % Number of equality constraints (0 <= me <= m)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ f, g ] = problem_function( x )

% Objective function value F(X) (denoted here as 'f')
  f = (x(1)-1)^2 ...   % x(1) is a continuous variable
    + (x(2)-2)^2 ...   % x(2) is a continuous variable
    + (x(3)-3)^2 ...   % x(3) is a discrete/integer variable
    + (x(4)-4)^2 ...   % x(4) is a discrete/integer variable
    + 1.23456789;

% Equality constraints G(X) = 0 MUST COME FIRST in g(1:me)
  g(1) = x(1) - 1;
% Inequality constraints G(X) >= 0 MUST COME SECOND in g(me+1:m)
  g(2) = x(2) - 1.333333333;
  g(3) = x(3) - 2.666666666;

return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 6: Problem dimensions and function call in Matlab/Octave

Note that the vector indexes for X(i) and G(i) start with i=1 in Matlab and Fortran, while they start with i=0 in C/C++ and Python.

It is possible (and relatively easy) to change the name of the problem function call, e.g.

[f, g] = UserFunctionName(x)

It is also possible to pass additional input/output arguments to the problem function call, e.g.

[f, g, UserOutputData] = UserFunctionName(x, UserInputData)

Please visit the MIDACO FAQ website on how to implement those issues in particular.

3.2 Verifying a problem implementation

If MIDACO (or any other optimizer) should be used to solve a specific problem, it is crucial that the problem is implemented correctly (otherwise a GIGO scenario might occur). In order to verify a problem implementation, it is recommended that the user executes a single evaluation of the problem (for some starting point X) and manually checks the output (objective and constraint function values) to be reasonable. In Section **4.5** an example of a MIDACO setup for only one single function evaluation is given.

4 The MIDACO Stopping Criteria

MIDACO does provide four different stopping criteria: MAXTIME, MAXEVAL, AUTOSTOP and FSTOP. MAXTIME specifies a maximum cpu-time budget (e.g. 60 Seconds) during which MIDACO is allowed to performs its search process. Analogue to MAXTIME, the MAXEVAL criteria specifies a maximum number of function evaluation (e.g. 1000 or 1000000). If AUTOSTOP is active (see Section **5.4**), MIDACO will stop automatically by itself. FSTOP defines a specific value for the objective function F(X) to be reached. If FSTOP is active and MIDACO succeeds in finding a (feasible) solution with an objective value lower or equal to FSTOP, MIDACO will stop. Note that those four stopping criteria can be freely combined with each other.

In general, the recommended stopping criteria is to use MAXTIME in combination with AUTOSTOP. The stopping criteria by MAXEVAL and FSTOP are of rather academic interest only. Obviously it depends on the user, how much time can be spent for an optimization run. Nevertheless, in Table **2** some example scenarios for possible stopping criteria setups are illustrated. The first three scenarios are considered as general reasonable choices for practitioners, while the last three scenarios exemplify rather specific setups for MIDACO's stopping criteria. The individual scenarios are explained below in subsections.

Note: The MAXEVAL stopping criteria disables itself if a value greater or equal to 100 Million is set. This is done to avoid numerical problems (integer flip) for very long runs on cpu-time cheap problem functions.

Table 2: Example scenarios for stopping criteria settings

Scenario 1		Scenario 2		Scenario 3	
MAXTIME	60	MAXTIME	60*60	MAXTIME	60*60*24
MAXEVAL	999999999	MAXEVAL	999999999	MAXEVAL	999999999
AUTOSTOP	0 (*inactive*)	AUTOSTOP	50	AUTOSTOP	500
FSTOP	0 (*inactive*)	FSTOP	0 (*inactive*)	FSTOP	0 (*inactive*)
Scenario 4		Scenario 5		Scenario 6	
MAXTIME	999999999	MAXTIME	999999999	MAXTIME	999999999
MAXEVAL	1000000	MAXEVAL	1	MAXEVAL	999999999
AUTOSTOP	0 (*inactive*)	AUTOSTOP	0 (*inactive*)	AUTOSTOP	5000
FSTOP	1.23456789	FSTOP	0 (*inactive*)	FSTOP	0 (*inactive*)

4.1 Scenario 1

In this scenario MIDACO will stop after exactly 60 Seconds. MAXEVAL, AUTOSTOP and FSTOP are inactive. This scenario is recommended for a first test run. Based on the outcome of such first run, the user can then assign shorter or longer runtimes for a second run.

4.2 Scenario 2

In this scenario, MAXTIME is used in combination with AUTOSTOP. MAXEVAL and FSTOP are inactive. MIDACO will run for a maximal time limit of 1 Hour (60*60 Seconds), but will (most probably) stop much earlier by itself due to the AUTOSTOP (50) criteria. This scenarios is probably a good choice for users with medium difficult problems.

4.3 Scenario 3

In this scenario, MAXTIME is used in combination with AUTOSTOP. MAXEVAL and FSTOP are inactive. MIDACO will run for a maximal time limit of 1 Day (60*60*24 Seconds), but will (most probably) stop much earlier by itself due to the AUTOSTOP (500) criteria. This scenario is probably a good choice for users with hard problems or for users, who want to gain further confidence that the global optimal solution is reached.

4.4 Scenario 4

In this scenario, MAXEVAL is used in combination with FSTOP. MAXTIME and AUTOSTOP are (practically) inactive. This scenario is of purely academic interest. MIDACO will perform up

to 1000000 function evaluation, but will eventually stop earlier, if a (feasible) solution with F(X) $\leq$ 1.23456789 is reached.

4.5 Scenario 5

In this scenario, no optimization at all is performed, because only a single evaluation is allowed by setting MAXEVAL=1. This means only the starting point is evaluated once and MIDACO stops immediately. This scenario is useful to verify (check), if the implementation of a problem works bug-free and returns the expected values for the starting point.

4.6 Scenario 6

In this scenario, only AUTOSTOP (5000) is the active stopping criteria. MIDACO will stop only, if 5000 internal algorithmic restarts did not succeed in improving the current best solution. This scenario is for users how have no time restriction and want to make as sure as possible to reach the global optimal solution. (Note that "example_MINLPc.cpp" will stop in less than 1 Minute when executed with this scenario on a regular PC)

5 MIDACO Parameter

MIDACO offers nine parameters to customize the optimization performance. The individual parameters are explained in the following subsections. The default value for all parameter is zero.

5.1 PARAM(1): ACCURACY

This parameter defines the accuracy tolerance for the constraints G(X). An equality constraint is considered feasible by MIDACO, if |G(X)| $\leq$ PARAM(1). An inequality is considered feasible by MIDACO, if G(X) $\geq$ -PARAM(1). If the user sets PARAM(1) = 0, MIDACO uses a default accuracy of 0.001. This parameter has strong influence on the MIDACO performance on constraint problems. For problems with difficult constraints, it is recommended to start with some test runs using a less precise accuracy (e.g. PARAM(1)=0.05 or even PARAM(1)=0.1) and to apply some refinement runs with a higher precision afterwards (e.g. PARAM(1)=0.0001 or PARAM(1)=0.0000001).

Note that the displayed "VIOLATION OF G(X)" (see MIDACO screen) expresses the L1-Norm over the vector G in respect to PARAM(1). In case all constraints are feasible to to accuracy defined by PARAM(1), the "VIOLATION OF G(X)" is displayed as zero.

5.2 PARAM(2): RANDOM SEED

This parameter defines the initial seed for MIDACO's internal pseudo random number generator. The seed determines the sequence of pseudo random numbers sampled by the generator. Therefore

changing this value will lead to different results by MIDACO. The seed must be an integer greater or equal to zero (e.g. PARAM(2) = 0,1,2,3,...1000).

Note that MIDACO runs are 100% reproducible, if performed with the same seed (and executed on the same computer with identical compiler settings). The advantage of a user specified random seed is, that promising runs can easily be reproduced. This is in esp. useful, if a run was unintentionally interrupted (e.g. power outage) and should be restarted again.

For difficult problems it can be useful to execute several runs of MIDACO with different random seed, rather than performing only one very long run.

5.3 PARAM(3): FSTOP

This parameter enables a stopping criteria for MIDACO. If PARAM(3) is not equal to zero, MIDACO will stop if a (feasible) solution is found with F(X) $\leq$ FSTOP. If PARAM(3) is equal to zero, this stopping criteria is inactive. In case the user wishes to use zero as FSTOP value, a dummy value (e.g. PARAM(3)=0.00000001) should be used.

Note that MIDACO does not apply any tolerance on the FSTOP value. For problems where the precision of F(X) is a critical issue, the user might want to add some tolerance to FSTOP (e.g. PARAM(3) = 1.0 + 0.0001, where 0.0001 is a tolerance).

5.4 PARAM(4): AUTOSTOP

This parameter enables a stopping criteria for MIDACO. If PARAM(4) is an integer greater than zero, MIDACO will activate its automatic stopping criteria. The automatic stopping criteria is based on the number of internal algorithmic restarts by MIDACO, which did not succeed in further improving the current best solution. For example: If PARAM(4) = 1, MIDACO will stop if any internal algorithmic restart of MIDACO did not improve the current best solution. If PARAM(4) = 50, MIDACO will stop if 50 (successive) internal restarts did not further improve the current best solution. Small values of PARAM(4) will cause MIDACO to stop earlier, but will lower MIDACO's chance of reaching the global optimal solution. Larger values of PARAM(4) will imply longer runtimes, but will give MIDACO a higher chance of reaching the global optimum. Tabel **3** illustrates some examples of AUTOSTOP values. Note that these examples are intended only as very rough illustration of the possible impact of the AUTOSTOP parameter.

Table 3: Examples of AUTOSTOP values and their possible impact on MIDACO

AUTOSTOP	Impact on MIDACO runtime and chance of global optimality
1	Fastest runtime but very low chance of global optimality
5	Fast runtime but low chance of global optimality
50	Medium long runtime and medium chance of global optimality
500	Long runtime but high chance of global optimality
5000	Very long runtime but also very high chance of global optimality

5.5 PARAM(5): ORACLE

This parameter specifies a user given oracle parameter to the penalty function within MIDACO. This parameter is only relevant for constrained problems. If PARAM(5) is not equal to zero, MIDACO will use PARAM(5) as initial oracle (otherwise MIDACO will use 10^9 as initial oracle). This option can be especially useful for constrained problems where some background knowledge on the problem exists. For example: It is known that a given application has a feasible solution X corresponding to F(X)=1000 (e.g. plant operating cost in Dollar). It might be therefore reasonable to submit an oracle value of 800 or 600 to MIDACO, as this cost region might hold a new feasible solution (to operate the plant at this cost value). Whereas an oracle value of more than 1000 would be uninteresting to the user, while a too low value (e.g. 200) would be unreasonable. Extensive information on the oracle penalty method can be found in [9].

5.6 PARAM(6): FOCUS

This parameter forces MIDACO to focus its search process around the current best solution. This parameter is probably the most powerful and widely applicable one. For many problems, tuning this parameter is useful and will result in a faster convergence speed (in esp. for convex and semi-convex problems). This parameter is also in especially useful for refining solutions (e.g. to improve the precision of their objective function value or constraint violation). If PARAM(6) is not equal zero, MIDACO will apply an upper bound for the standard deviation of its Gauss PDF's (see Section **1**, Figure **1**). The upper bound for the standard deviation for continuous variables is given by (XU(i)-XL(i))/FOCUS, whereas he upper bound for the standard deviation for integer variables is given by MAX((XU(i)-XL(i))/FOCUS,1/SQRT(FOCUS)).

In other words: The larger the value of FOCUS, the closer MIDACO will concentrate its search around its current best solution.

The value for PARAM(6) must be an integer. Smaller values for FOCUS (e.g. 10 or 100) are recommend for first test runs (without a specific starting point). Larger values for FOCUS (e.g. 10000 or 100000) are normally only useful for refinement runs (where a specific solution is used as starting point).

Furthermore it is possible to submit negative values for FOCUS (e.g. -1000 or -10000). In such case, the minus ("-") is not treated numerically; instead, MIDACO will interpret the minus ("-") as

an information flag. While for positive FOCUS values MIDACO will also explore other regions of the search space by independent restarts, a negative FOCUS value disable the independent restart option within MIDACO. In other words: For a negative FOCUS value MIDACO is focused entirely on the starting point. Therefore negative FOCUS values should be used only for refinement runs, where the user has high confidence in the quality of the specific solution used as starting point.

5.7 PARAM(7): ANTS

This parameter allows the user to fix the number of *ants* (iterates) which MIDACO generates within one generation (major iteration of the evolutionary ACO algorithm). This parameter must be used in combination with PARAM(8). Using the ANTS and KERNEL parameters can be promising for some problems (in esp. large scale problems or cpu-time intensive applications). However, tuning these parameters might also significantly reduce the MIDACO performance. If PARAM(7) is equal to zero, MIDACO will dynamically change the number of ants per generation. See PARAM(8) for more information on handling this parameter.

5.8 PARAM(8): KERNEL

This parameter allows the user to fix the number of kernels within MIDACO's multi-kernel Gauss PDF's (see Section **1**, Figure **1**). The kernel size corresponds also to the number of solutions stored in MIDACO's solution archive. On rather convex problems it can be observed, that a lower kernel number will result in faster convergence while a larger kernel number will result in lower convergence. On the contrary, a lower kernel number will increase the risk of MIDACO getting stuck in a local optimum, while a larger kernel number increases the chance of reaching the global optimum. The kernel parameter must be used in combination with the ants parameter. In Table **4** some examples of possible ants/kernel settings are given and explained below.

Table 4: Example settings for ANTS/KERNEL combinations

Setting 1	Setting 2	Setting 3	Setting 4
ANTS 2	ANTS 30	ANTS 500	ANTS 100
KERNEL 2	KERNEL 5	KERNEL 10	KERNEL 50

The 1st setting is the smallest possible one. This setting might be useful for very cpu-time expensive problems where only some hundreds of function evaluation are possible or for problems with a specific structure (e.g. convexity). The 2nd setting might also be used for cpu-time expensive problems, as a relatively low number of ANTS is considered. The 3rd and 4th setting would only be promising for problems, with a fast evaluation time. As tuning the the ants and kernel parameters is highly problem depended, the user needs to experiment with those values.

Note that the maximum kernel number for MIDACO is fixed to 100.

5.9 PARAM(9) = CHARACTER

This character allows to activate MIDACO internal parameter settings, which can be customized to a specific user problem. Using this parameter is currently only available as a service from the authors upon request.

6 Running MIDACO in Parallel Mode

For problems with cpu-time expensive objective and/or constraint functions, MIDACO offers an effective parallelization strategy. MIDACO allows the parallel execution of the problem function evaluation calls. This way significant speed ups can be gained, if problems are time expensive and sufficient parallelization threads (cores) are available.

Running MIDACO in parallel is recommended, if a single function evaluation takes more than a specific time. The specific time depends on the programming language (due to the efficiency of the parallelization overhead in each language). Below is a language depended list of minimal cpu-time cost, for which running MIDACO in parallel mode is recommended:

Language	Minimal cost for which parallelization is promising
Matlab	0.1 Second
Python	0.01 Second
C/C++	0.001 Second
Fortran	0.001 Second

The parallelization option for MIDACO is available for a wide range of platforms and approaches and is very easy to use. The user only needs to specify the parallelization factor (called "P" or "option.parallel" in Matlab/Octave), which specifies the number of parallel executed problem function calls. Normally the parallelization factor is equal to the number of cores (or threads) available on a machine. For a duo-core CPU, P would be 2. For a quad-core CPU, P would be 4.

MIDACO examples running in parallel mode are freely available at the MIDACO parallelization website and can be executed with the limited version of MIDACO.

A real-world application, where running MIDACO in parallel mode are the ESA/ACT GTOP benchmark problems displayed at the MIDACO benchmark website. Those problems are even cheaper than 0.001 Second (C/C++), but running MIDACO in parallel does already give a speed up of about 4 times compared to the serial execution. Another example of a real-world application solved by MIDACO in parallel mode can be found in [13].

Note: If the cpu-time evaluation cost of a problem is significantly cheaper than those reported in list above, running MIDACO in parallel mode is not recommended due to the algorithmic overhead.

7 Warning and Error Messages

MIDACO returns specific warning and error messages via an information flag value called IFLAG. In case of warnings, the IFLAG value is printed on the screen and MIDACO will proceed with the optimization nevertheless. In case of errors, MIDACO will stop immediately, displaying the IFLAG value on the screen. In the regular case (no warning or error) MIDACO will provide a final IFLAG message along with the solution, indicating the stopping criteria and the feasibility.

Table 5: MIDACO solution messages indicated by IFLAG

IFLAG	
1	Feasible solution found, MIDACO was stopped by MAXEVAL or MAXTIME
2	Infeasible solution found, MIDACO was stopped by MAXEVAL or MAXTIME
3	Feasible solution, MIDACO stopped automatically by AUTOSTOP
4	Infeasible solution, MIDACO stopped automatically by AUTOSTOP
5	Feasible solution, MIDACO stopped automatically by FSTOP

Table 6: MIDACO warning messages indicated by IFLAG

IFLAG	
51	Some X(i) is greater/lower than +/- 10^8 (try to avoid huge values!)
52	Some XL(i) is greater/lower than +/- 10^8 (try to avoid huge values!)
53	Some XU(i) is greater/lower than +/- 10^8 (try to avoid huge values!)
61	Some X(i) should be discrete (e.g. 1.0), but is continuous (e.g. 1.234)
62	Some XL(i) should be discrete (e.g. 1.0), but is continuous (e.g. 1.234)
63	Some XU(i) should be discrete (e.g. 1.0), but is continuous (e.g. 1.234)
71	Some XL(i) = XU(i) (fixed variable)
81	F(X) has value NaN for starting point X
82	Some G(X) has value NaN for starting point X
91	FSTOP is greater/lower than +/- 10^8
92	ORACLE is greater/lower than +/- 10^8

Table 7: MIDACO error messages indicated by IFLAG

IFLAG	
101	P <= 0 or P > 10^6
102	N <= 0 or N > 10^6
103	NI < 0
104	NI > N
105	M < 0 or M > 10^6
106	ME < 0
107	ME > M
201	Some X(i) has type NaN
202	Some XL(i) has type NaN
203	Some XU(i) has type NaN
204	Some X(i) < XL(i)
205	Some X(i) > XU(i)
206	Some XL(i) > XU(i)
301	PARAM(1) < 0 or PARAM(1) > 10^6
302	PARAM(2) < 0 or PARAM(2) > 10^{12}
303	PARAM(3) greater/lower than +/- 10^{12}
304	PARAM(4) < 0 or PARAM(4) > 10^6
305	PARAM(5) greater/lower than +/- 10^{12}
306	|PARAM(6)| < 1 or PARAM(6) > 10^{12}
307	PARAM(7) < 0 or PARAM(7) > 10^8
308	PARAM(8) < 0 or PARAM(8) > 100
309	PARAM(7) < PARAM(8)
310	PARAM(7) > 0 but PARAM(8) = 0
311	PARAM(8) > 0 but PARAM(7) = 0
312	PARAM(9) < 0 or PARAM(9) > 1000
313	Some PARAM(i) has type NaN
401	ISTOP < 0 or ISTOP > 1
501	Double precision work space size LRW is too small. RW must be at least of size LRW = 200*N+2*M+1000
601	Integer work space size LIW is too small. IW must be at least of size LIW = 2*N+P+1000
701	Input check failed! MIDACO must be called initially with IFLAG = 0
801	P > PMAX (user must increase PMAX in the MIDACO source code)
802	P*M+1 > PXM (user must increase PXM in the MIDACO source code)
900	Invalid or corrupted LICENSE-KEY
999	N > 4. The free test version is limited up to 4 variables.

References

[1] Allugundu I., Puranik P., Lo Y.P. and Kumar A.: *Acceleration of distance-to-default with hardware-software co-design.* 22nd International Conference on Field Programmable Logic and Applications (FPL) (2012)

[2] Baidas M.W. and MacKenzie A.B.: *On the Impact of Power Allocation on Coalition Formation in Cooperative Wireless Networks.* IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (2012)

[3] European Space Agency (ESA) and Advanced Concepts Team (ACT): Gtop database - global optimisation trajectory problems and solutions. (2013)

[4] Haenel M., Kuhn S., Henrich D., Gruene L. and Pannek J.: *Optimal camera placement to measure distances regarding static and dynamic obstacles.* Int. J. of Sensor Networks, 12(1), pp.25–36 (2012)

[5] Rehberg M., Ritter J.B, Genzela Y., Flockerzi D. and Reichl U.: *The relation between growth phases, cell volume changes and metabolism of adherent cells during cultivation.* J. Biotechnol., 164(4), pp. 489–499 (2013)

[6] Schittkowski K.: *NLPQLP - A Fortran Implementation of a Sequential Quadratic Programming Algorithm with distributed and non-monotone Line Search (User Guide)*, Report, Department of Computer Science, University of Bayreuth (2009)

[7] Schlueter M., Egea J.A. and Banga J.R.: *Extended Ant Colony Optimization for non-convex Mixed Integer Nonlinear Programming*, Comput. Oper. Res. 36(7), pp. 2217–2229 (2009)

[8] Schlueter M., Egea J.A., Antelo L.T., Alonso A.A. and Banga J.R.: *An extended Ant Colony Optimization algorithm for integrated Process and Control System Design*, Ind. Eng. Chem. 48(14), pp. 6723–6738 (2009)

[9] Schlueter M. and Gerdts M.: *The Oracle Penalty Method*, J. Global Optim. 47(2), pp. 293–325 (2010)

[10] Schlueter M., Rueckmann J.J and Gerdts M.: *A Numerical Study of MIDACO on 100 MINLP Benchmarks*, Optimization, 61(7), pp. 873–900 (2012)

[11] Schlueter M.: *Nonlinear mixed integer based Optimisation Technique for Space Applications*, Ph.D. Thesis, School of Mathematics, University of Birmingham (UK) (2012)

[12] Schlueter M., Erb S., Gerdts M., Kemble S. and Rueckmann J.J.: *MIDACO on MINLP Space Applications*, Optimization, 51(7), pp.1116–1131 (2013)

[13] Schlueter M. and Munetomo M.: *Parallelization Strategies for Evolutionary Algorithms for MINLP*, *submitted* (2013)

[14] Takano A.T. and Marchand B.G.: *Optimal Constellation Design for Space Based Situational Awareness Applications* AAS/AIAA Astrodynamics Specialists Conference (Paper No. AAS11-543) (2011)

[15] Wong S.I.: *On Lightweight Design of Submarine Pressure Hulls.* MSc Thesis, Delft University of Technology (2012)