April 2018

http://www.midaco-solver.com

— User Manual —

Version 6.0

Abstract

MIDACO is a numerical high-performance solver for single- and multi-objective optimization. It is constructed as general-purpose software and can be applied to a wide range of optimization problems. MIDACO is based on a derivative-free, evolutionary hybrid algorithm that treats the objective and constraint functions as black-box which may contain critical function properties like non-linearity, non-convexity, discontinuities or even stochastic noise. Decision variables may be continuous, discrete (e.g. binary) or both (called *mixed integer*). The software can handle problems with thousands of variables and hundreds of objectives. MIDACO offers (massive) parallelization options which is particularly beneficial for CPU-time intensive applications (e.g. numerical simulations), where a single evaluation may take significant time.

Quick Jump Menu

Contents

Overview

The key facts on MIDACO:

- **MIDACO is a solver for global optimization problems**
  - Single- and multi-objective optimization
  - Continuous, discrete/combinatorial and mixed integer variables
  - Constrained and unconstrained problems
- **Evolutionary hybrid algorithm**
  - Fundamentally based on the Ant Colony Optimization (ACO) metaheuristic
  - Internally hybridized with a backtracking line-search for fast local convergence
  - Objective and constraints may be linear or non-linear (differentiability not required)
  - Black-box solver: Objective and constraint functions may be unknown (e.g. simulations)
- **Large scale capability**
  - MIDACO solves problems with up to **100,000** variables
  - MIDACO can handle up to thousands of constraints and hundreds of objectives
- **Parallelization**
  - MIDACO features various parallelization schemes in several programming languages
  - Capable of massive parallelization with thousands of cores/threads (incl. GPGPU)
- **Languages**
  - Excel, VBA, Java, C#, R, Matlab, Octave, Python, Julia, C/C++, Fortran *and more...*
- **Source code**
  - Extensively tested and constantly improved for **over 10 Years**
  - Super lightweight (∼200kb) compressed ANSI-C (*midaco.c*) or Fortran code (*midaco.f*)
  - Completely self-sufficient source code (no third-party dependencies)
  - Compiles and runs on all platforms, incl. Win/Mac/Unix and web-servers
  - **Very easy to use and embed**
- **Record solutions**
  - MIDACO holds several records on interplanetary space trajectory benchmarks
- **Background**
  - Developed in collaboration with the European Space Agency (ESA)
  - Extended with support of the Japanese Space Exploration Agency (JAXA)

Introduction

MIDACO is a software tool for numerical optimization. The MIDACO algorithm is constructed as **general-purpose solver** for **single-** and **multi-objective** optimization problems. A special feature of MIDACO is its capability to handle (constrained) mixed integer nonlinear programming (MINLP) problems. The term *mixed integer* refers here to optimization problems where some decision variables are of continuous type (like 1.23 or 4.56) while others are of discrete type (like 1, 2 or 3). The mathematical formulation of the general multi-objective MINLP considered by MIDACO is stated as follows:

$$\begin{aligned} \text{Minimize} \quad & f_1(x),\ f_2(x), ...,\ f_O(x) \\ \\ \text{subject to:} \quad & g_i(x) = 0, \quad i = 1, ..., m_e \\ & g_i(x) \geq 0, \quad i = m_e + 1, ..., m \\ & x_l \leq x \leq x_u \quad \text{(box constraints)} \end{aligned}$$

In above problem formulation, the vector $f_{1,...,O}(x)$ denotes the objective functions and the vector $g_{1,...,m}(x)$ denotes the constraint functions. Without loss of generality, all objectives are subject to minimization. The first $1, ..., m_e$ values of the constraint vector $g(x)$ represent equality constraints, while the remaining $m_e + 1, ..., m$ values represent in-equality constraints. The vector x of decision variables contains continuous variables as well as discrete variables (also called *integer*, *categorical* or *combinatorial* variables), whereas the continuous ones are stored first and the discrete ones are stored last. Furthermore, some box constraints as lower bounds x_l and upper bounds x_u are assumed on the decision variables x.

MIDACO solves above multi-objective MINLP by combining an extended evolutionary *Ant Colony Optimization* (ACO) [34] algorithm with the *Oracle Penalty Method* [36] for constrained handling. The ACO algorithm within MIDACO is based on so called multi-kernel gaussian probability density functions (PDF's), which generate samples of iterates (also called *ants* or *individuals*). For integer decision variables, a discretized version of the PDF is applied (see [34]). Figure 1 illustrates a Gauss PDF with three individual kernel PDF's for a continuous (left) and integer (right) domain.

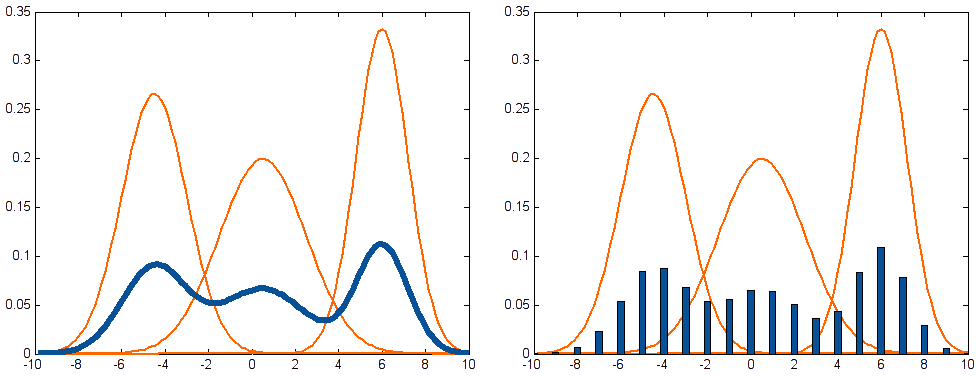


Figure 1: Continuous (left) and discretized (right) multi-kernel Gauss PDF

Constraints are handled within MIDACO by the Oracle Penalty Method which is an advanced method especially developed for metaheuristic 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 [36]), which corresponds directly to the objective function value $f(x)$. 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 $f(x)$ and the *residual* value $res(x)$, which represents the constraint violation of $g(x)$ (measured in the commonly used L_1-norm).

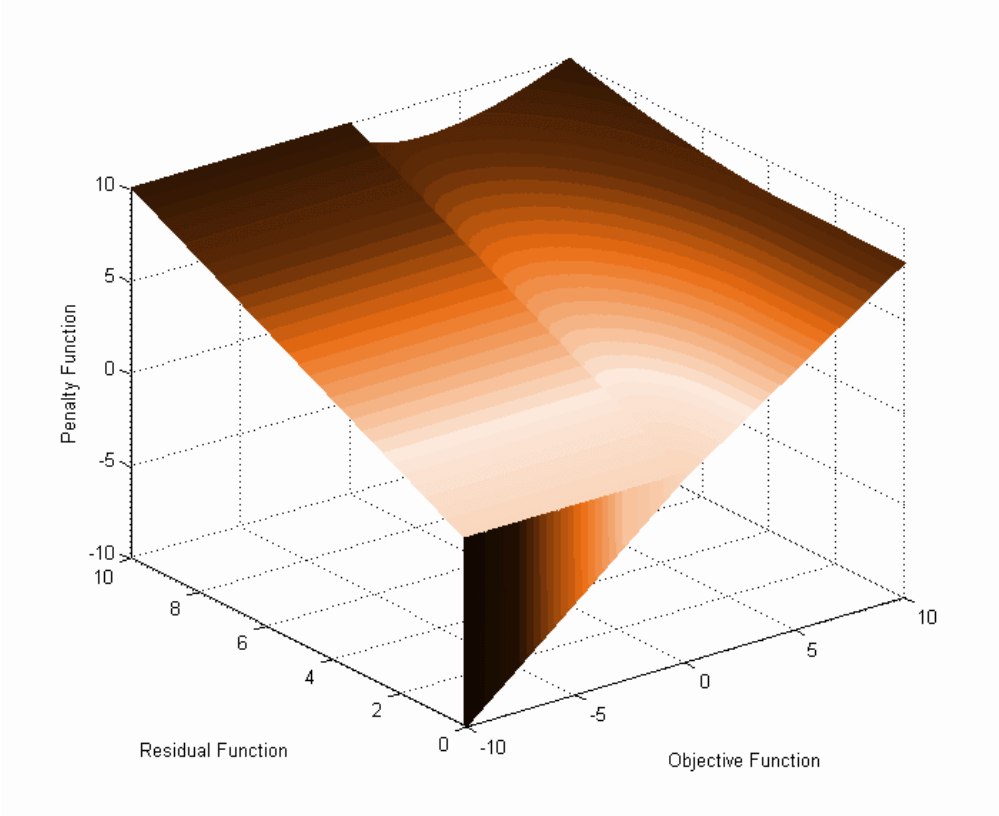


Figure 2: Shape of the extended oracle penalty function

Like the majority of evolutionary optimization algorithms, MIDACO considers the objective $f(x)$ and constraint $g(x)$ functions as **black-box** functions. This means that for some input vector x only the returning objective $f(x)$ and constraint $g(x)$ values are recognized by MIDACO. No particular knowledge on how the objective and constraint function are actually calculated is required by MIDACO. Consequently the objective and constraint functions may exhibit any critical function property, like (high) non-linearity, non-convexity, non-smoothness, non-differentiability, discontinuities and even stochastic noise.

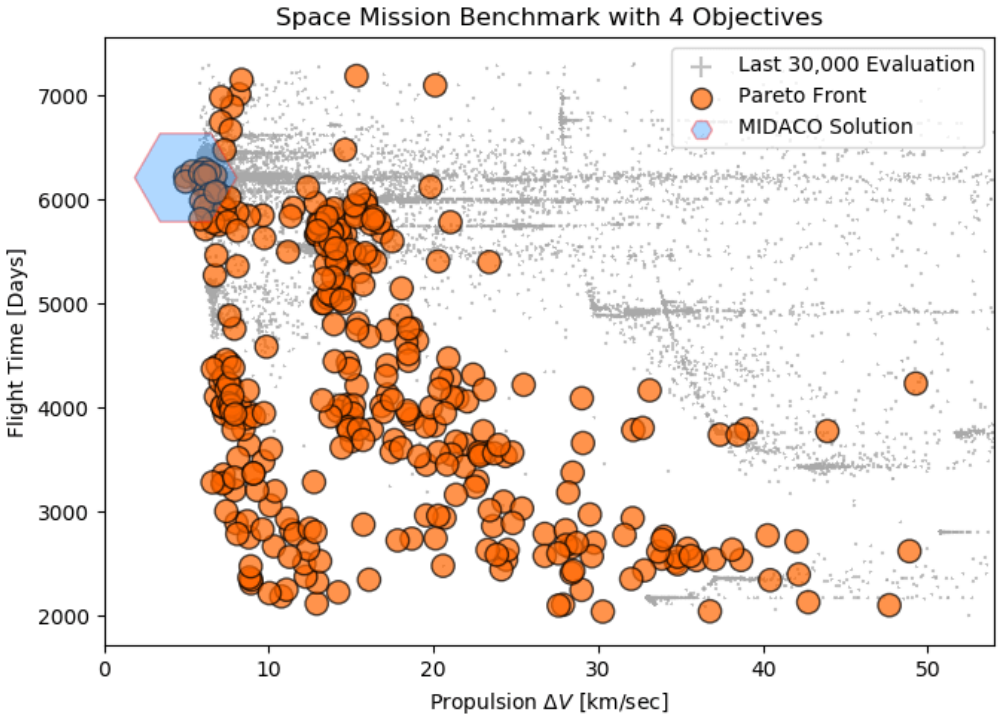
This **black-box** concept gives the user **absolute freedom** to formulate the problem in any desired way. For example, the problem formulation may contain any kind of programming statement (like *if-clauses* or *subroutine* calls) or even call exeternal programs (like Simulink or Text-I/O).

In order to enhance its overall performance, MIDACO implements many heuristics and is internally hybridized with a pseudo-gradient based backtracking line-search for fast local convergence. However, like all heuristic algorithms, MIDACO does not provide a 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 reasonably good solution. Extensive numerical test show (see [37], [36] and especially [44]) that MIDACO is able to obtain global optimal solutions fast and reliable for a large set of MINLP benchmark problems. Numerical comparisons of MIDACO with established deterministic MINLP algorithms can be found for example in [37]. Numerical comparisons of MIDACO with stochastic search algorithms (including genetic algorithms, scatter search, variable neighborhood and covariance matrix based search) can be found for example in [34], [35] or [40]. A collection of general global optimization problems (including well known NLP benchmarks like Rosenbrock, Ackley or Rastrigin) that can be solved by MIDACO are available at the MIDACO benchmark website. Note that the MIDACO run-times and capabilities (e.g. number of objectives, number of variables and number of constraints) considered at the MIDACO benchmark website are at the **state-of-the-art in evolutionary computing**, which is particular true for its MINLP and large-scale performance with thousands of variables.

As a general-purpose solver, MIDACO can be applied to a wide range of optimization problems. On the MIDACO applications website some examples of MIDACO utilizations are presented for interplanetary space trajectory design [18], [38], [41], [42], construction and control of launch vehicles [39], operation of satellite constellations [45], low-thrust orbit transfer optimization [5], cubesat deployment trajectory design [26], structural optimization of aircraft frames [50], aircraft battery optimization [4], passenger and aircraft fleet allocation [30], attitude control of quadrotors [19], chemical plant layout [35], [40], [16], waste water treatment [35], water supply networks [11], optimal camera placement [20], soil parameter optimization [48], meta-material fabric design [14], distance-to-default models in finance [3], [24], sales forecasting [13], wireless network telecommunication [2], [6], [7], [8], [9], [10], [29], network structural vulnerability [17], structural optimization of submarines [51], parameter optimization in bio-technology [32], neuroscience [25], filter design for mitigating harmonics [22], [23], forestry forecasting [12], economic analysis [47], car belt conveyors optimization [28], automotive industry planning [1], [15], electric drives for traction applications [53], optimization of combined cycle power plants [27], CO_2 power cycle design [52], low-carbon energy portfolio optimization [49], oilfield operation [31] or natural gas plant optimization [46].

For CPU-time expensive problems (this means that a single evaluation of the objective and/or constraint functions requires a significant amount of time), MIDACO offers an efficient parallelization strategy: MIDACO allows to evaluate several solution candidates in **parallel**. This strategy is also known as *co-evaluation* or *fine-grained* parallelization. This strategy can significantly reduce the overall optimization time. The parallelization strategy in MIDACO is implemented by reverse communication which is a robust and portable concept that can scale up to thousands of threads/cores. Due to this concept, MIDACO is able to offer its parallelization strategy in several programming languages for various parallelization schemes, including Fortran and C/C++ (openMP, openMPI, GPGPU), Matlab (parfor), R (dopar), Java (Fork/Join) or Python (multiprocessing, mpi4py, spark). Easy to use example templates for MIDACO running with parallelization can be found on the MIDACO parallelization website.

For **multi-objective optimization** problems, MIDACO applies the recently introduced *Utopia-Nadir-Balance* [43] concept, which was particular developed for *many-objective* problems arising in (aero)space applications. Many-objective optimization problems differ from multi-objective ones in such regard, that they consider four or more objectives. The *Utopia-Nadir-Balance* concept differs from traditional multi-objective approaches in such regard, that it concentrates the algorithmic search effort on a particular area of the pareto front. By default, this is the *central* (or *middle*) part of the pareto front, as this part provides the *best equally balanced* trade-off between all individual objective functions. However, tuning the **BALANCE** parameter, users can freely change the focus to any other part of the pareto front. The main advantage of the *Utopia-Nadir-Balance* is that the most desired part of the pareto front is normally reached faster and explored in more depth as with traditional methods (like *non-dominated sorting*), which treat all parts of the pareto front with equal importance. The *Utopia-Nadir-Balance* concept works on all kinds of pareto fronts, may them be convex, concave, mixed or separated, and has been tested successfully with up to some hundreds of objectives (see MIDACO benchmark website). Below is an example plot of applying MIDACO on ESA's Cassini1 [18] space mission benchmark with four objectives. In below plot the BALANCE has been set on the first objective (Propulsion $\Delta V \rightarrow$ displayed on the x-axis) and it can be observed from the last 30,000 evaluation how MIDACO concentrates its search effort on that part of the pareto front, which contains the solutions with the lowest value in such regard.



The scope of this user manual is to provide practical guidelines on how to setup and solve an optimization problem with the MIDACO software. Readers with a deeper interest in the theoretic details of the ACO algorithm within MIDACO can find more information in several publications (e.g. [34] or [35]). Detailed information on the development and properties of the oracle penalty method can be found in [36] or on the Oracle Penalty Method website.

1 Optimization Problem

This section explains how an optimization problem must be presented to MIDACO. The MIDACO algorithm considers an optimization problem in its most fundamental form: A black-box which returns for some input variables X the corresponding objective function values F(X) and constraint function values G(X). This black-box concept is commonly used in evolutionary algorithms and gives the user complete freedom to define and calculate the objective and constraint values in whatever form is preferred, including subroutines calls and even external subprograms. Due to this black-box concept it is furthermore allowed that the objective and constraint functions have critical properties (like high non-linearity, discontinuity or stochastic noise) or that their actual mathematical formulation is truly unknown (e.g. simulation codes).

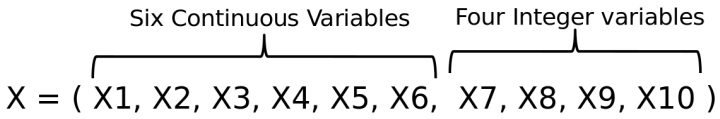
In case of *mixed integer* problems, where continuous and discrete (also called *integer*) variables are present simultaneously, the continuous variables are stored first in the vector of variables X, while the discrete ones are stored last in X. The distinction between equality and inequality constraints in the constraints vector G(X) is handled the same way: The equality constraints are stored first, the inequality constraints are stored last in the constraints vector G. As example, consider a constrained mixed integer problem with the following problem dimensions:

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

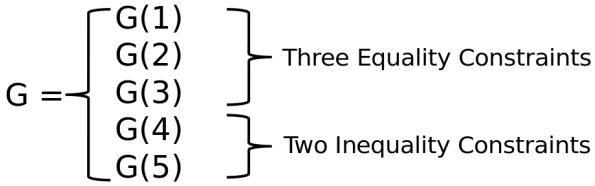
where:

N	:	Number of variables (in total)
NI	:	Number of integer variables
M	:	Number of constraints (in total)
ME	:	Number of equality constraints

then the distinction between continuous and integer variables in X is as follows:



and the distinction between equality and inequality constraints in G is as follows:



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 inbetween the bounds XL and XU. By default, the lower bounds are assumed as starting point in all example problems provided with MIDACO. 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.

In case the starting point X violates its lower or upper bound, MIDACO will raise the error message IFLAG=204 or IFLAG=205 respectively. In case integer variables the corresponding lower and upper bound value must also be a discrete value. In case the starting point X has some variables declared as integer type but those variables have a continuous value (e.g. 0.123), MIDACO will raise error message IFLAG=881. In case the bound value for an integer variable is a continuous value (e.g. 0.123) MIDACO will raise error message IFLAG=882 or IFLAG=883.

Note: **MIDACO will always respect the integer/discrete type of variables.** This means that MIDACO will never try to submit a continuous value for an integer type declared X variable for evaluation to the problem function. This is an important feature that distinguishes the MIDACO algorithm from classical MINLP algorithms (like *Branch & Bound*) which require the so called *relaxation* of integer variables, which is a temporary violation of their integer type into the continuous domain during the optimization process.

1.1 Problem Dimensions, Bounds and Starting Point

This subsection illustrates how to declare the dimensions of the optimization problem, the lower and upper bounds and the starting point for the decision variable vector X. The problem dimensions refer there to the size of the F, G and X arrays. Below Matlab screenshot from the *example_MINLPc.m* illustrates a problem setup with 1 objective function and 4 variables, two of them of integer/discrete type. It furthermore considers 3 constraints, one of them of equality type:

```
% STEP 1.A: Problem dimensions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem.o  = 1; % Number of objectives
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)

% STEP 1.B: Lower and upper bounds 'xl' & 'xu'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem.xl = [ 1, 1, 1, 1];
problem.xu = [ 4, 4, 4, 4];

% STEP 1.C: Starting point 'x'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem.x  = problem.xl; % Here for example: 'x' = lower bounds 'xl'
```

1.2 Problem Function Call

This subsection discusses the software function call to the optimization problem. As explained above, the only three elements that need to be communicated between MIDACO and the optimization problem are the vector of decision variables X, the vector of objective functions F(X) and the vector of constraint values G(X). In the example problems distributed on the MIDACO website, those problem function calls are given by their programming language as follows:

Matlab	:	[f, g] = problem_function(x)
Python	:	problem_function(x) (return f, g)
Julia	:	problem_function(x) (return f, g)
R	:	problem_function <- function(f,g,x)
C/C++	:	problem_function(*double* *f, *double* *g, *double* *x)
Fortran	:	PROBLEM_FUNCTION(F,G,X)
VBA	:	PROBLEM_FUNCTION_VB(X,F,G)
C#	:	blackbox(double[ ] f, double[ ] g, double[ ] x)
Java	:	blackbox(double[ ] f, double[ ] g, double[ ] x)

Below Matlab screenshot from the *example_MINLPc.m* illustrates the problem_function setup, in which for the input vector of decision variables x some calculation is performed to generate a single objective value f(1) and three constraint function values g(1), g(2) and g(3).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%   OPTIMIZATION PROBLEM   %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ f, g ] = problem_function( x )

% Objective functions F(X)
  f(1) = (x(1)-1)^2 ...
       + (x(2)-2)^2 ...
       + (x(3)-3)^2 ...
       + (x(4)-4)^2 ...
       + 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;

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%   END OF FILE   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Note that the vector index i for the F, G and X arrays may start with $i = 1$ in some languages (e.g. Matlab, R, Fortran) while it starts with $i = 0$ in other languages (e.g. C++, Python, Java).

1.3 Passing Additional Input/Output Arguments

The problem function call provided in the examples is reduced to the bare minimum, that is communication only the F,G and X vectors. In case the user wishes to change the name of the problem function and/or pass additional input/output arguments, the problem function call can freely be changed to whatever layout is desired. In case of C/C++ and Fortran such changes can be done directly in the example file source code. In case of other languages (like Matlab or Python) the source code in the language specific gateway file must be changed. Changing the name and/or arguments of the problem function can be done easily with a little programming effort by the user.

Below is a Matlab pseudo-code of how a modified problem function call may look like:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%   OPTIMIZATION PROBLEM   %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ f, g, EXTRA_OUTPUT ] = USER_MODEL_NAME( x, EXTRA_INPUT )

###### Do something with EXTRA_INPUT ######

% Objective functions F(X)
  f(1) = (x(1)-1)^2 ...
       + (x(2)-2)^2 ...
       + (x(3)-3)^2 ...
       + (x(4)-4)^2 ...
       + 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;

###### Prepare EXTRA_OUTPUT ######

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%   END OF FILE   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

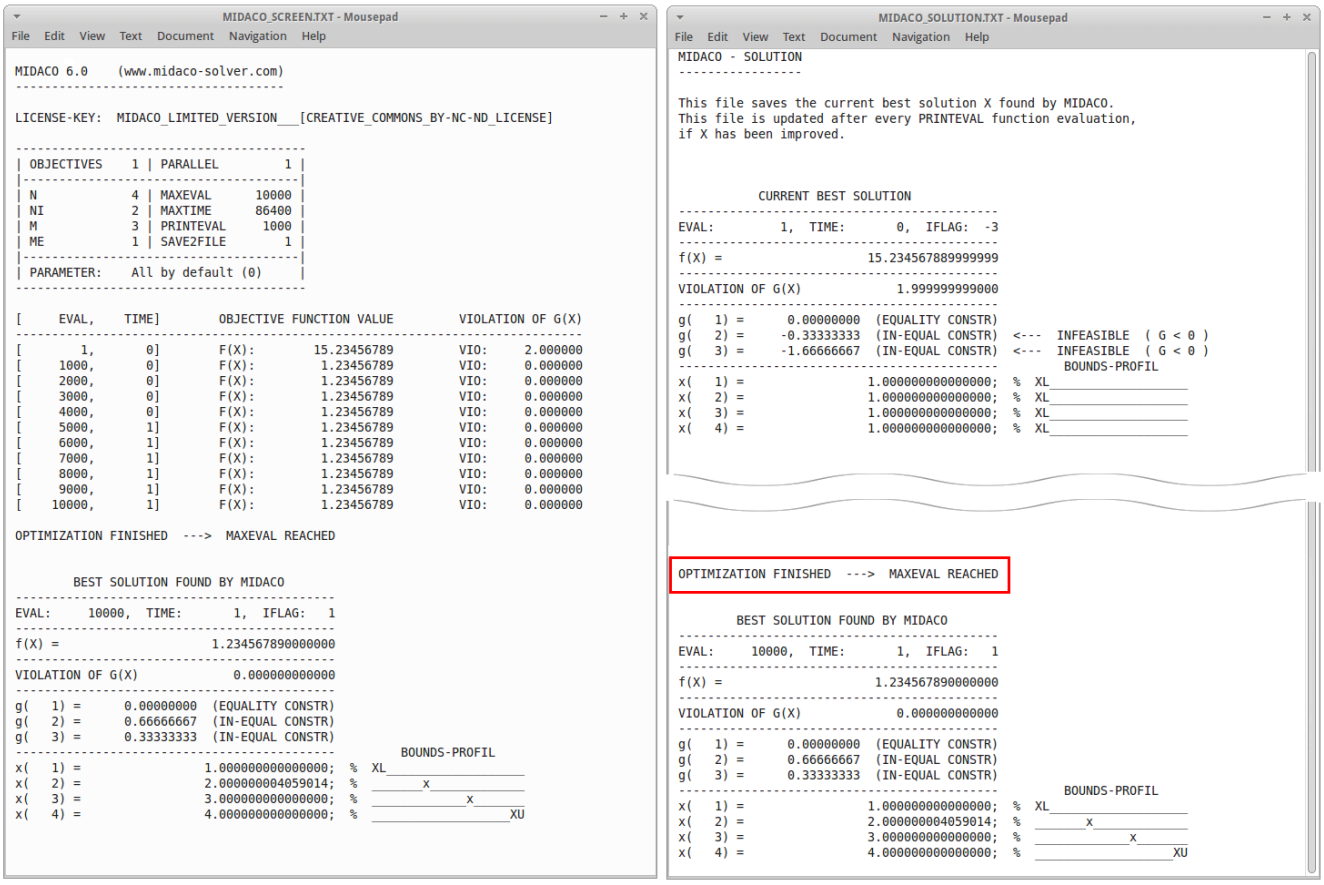
1.4 Verifying a Problem Implementation

If MIDACO (or any other optimizer) should be used to solve a specific problem, it is crucial that the problem is implemented correctly. In order to verify a problem implementation, it is recommended that before the actual optimization begins the user executes a single function evaluation. This can be achieved by setting the option MAXEVAL = 1 (see Section 3). By setting MAXEVAL=1 only the starting point will be evaluated and MIDACO stops immediately, reporting the corresponding objective and constraint values. This gives the user the chance to manually check if all reported objective and constraint values are reasonable.

2 MIDACO Screen and Solution

If printing is enabled, MIDACO produces two output text files:

MIDACO_SCREEN.TXT and MIDACO_SOLUTION.TXT



MIDACO_SCREEN.TXT - Mousepad

```
File  Edit  View  Text  Document  Navigation  Help

MIDACO 6.0    (www.midaco-solver.com)
----------------------------------

LICENSE-KEY:  MIDACO_LIMITED_VERSION___[CREATIVE_COMMONS_BY-NC-ND_LICENSE]

---------------------------------------
| OBJECTIVES    1 | PARALLEL         1 |
|-------------------------------------|
| N             4 | MAXEVAL      10000 |
| NI            2 | MAXTIME      86400 |
| M             3 | PRINTEVAL     1000 |
| ME            1 | SAVE2FILE        1 |
|-------------------------------------|
| PARAMETER:    All by default (0)     |
---------------------------------------

[     EVAL,    TIME]        OBJECTIVE FUNCTION VALUE         VIOLATION OF G(X)
-------------------------------------------------------------------------------
[        1,       0]        F(X):        15.23456789         VIO:     2.000000
[     1000,       0]        F(X):         1.23456789         VIO:     0.000000
[     2000,       0]        F(X):         1.23456789         VIO:     0.000000
[     3000,       0]        F(X):         1.23456789         VIO:     0.000000
[     4000,       0]        F(X):         1.23456789         VIO:     0.000000
[     5000,       1]        F(X):         1.23456789         VIO:     0.000000
[     6000,       1]        F(X):         1.23456789         VIO:     0.000000
[     7000,       1]        F(X):         1.23456789         VIO:     0.000000
[     8000,       1]        F(X):         1.23456789         VIO:     0.000000
[     9000,       1]        F(X):         1.23456789         VIO:     0.000000
[    10000,       1]        F(X):         1.23456789         VIO:     0.000000

OPTIMIZATION FINISHED  --->  MAXEVAL REACHED


        BEST SOLUTION FOUND BY MIDACO
---------------------------------------------
EVAL:     10000,  TIME:       1,  IFLAG:   1
---------------------------------------------
f(X) =                    1.234567890000000
---------------------------------------------
VIOLATION OF G(X)             0.000000000000
---------------------------------------------
g(   1) =      0.00000000  (EQUALITY CONSTR)
g(   2) =      0.66666667  (IN-EQUAL CONSTR)
g(   3) =      0.33333333  (IN-EQUAL CONSTR)
---------------------------------------------                BOUNDS-PROFIL
x(   1) =                 1.000000000000000;  %  XL_________________
x(   2) =                 2.000000004059014;  %  _______x_________
x(   3) =                 3.000000000000000;  %  _____________x___
x(   4) =                 4.000000000000000;  %  _________________XU
```

MIDACO_SOLUTION.TXT - Mousepad

```
File  Edit  View  Text  Document  Navigation  Help
MIDACO - SOLUTION
-----------------

This file saves the current best solution X found by MIDACO.
This file is updated after every PRINTEVAL function evaluation,
if X has been improved.



          CURRENT BEST SOLUTION
---------------------------------------------
EVAL:         1,  TIME:       0,  IFLAG:  -3
---------------------------------------------
f(X) =                    15.234567889999999
---------------------------------------------
VIOLATION OF G(X)             1.999999999000
---------------------------------------------
g(   1) =      0.00000000  (EQUALITY CONSTR)
g(   2) =     -0.33333333  (IN-EQUAL CONSTR)  <---  INFEASIBLE  ( G < 0 )
g(   3) =     -1.66666667  (IN-EQUAL CONSTR)  <---  INFEASIBLE  ( G < 0 )
---------------------------------------------        BOUNDS-PROFIL
x(   1) =                 1.000000000000000;  %  XL_________________
x(   2) =                 1.000000000000000;  %  XL_________________
x(   3) =                 1.000000000000000;  %  XL_________________
x(   4) =                 1.000000000000000;  %  XL_________________

OPTIMIZATION FINISHED  --->  MAXEVAL REACHED

        BEST SOLUTION FOUND BY MIDACO
---------------------------------------------
EVAL:     10000,  TIME:       1,  IFLAG:   1
---------------------------------------------
f(X) =                    1.234567890000000
---------------------------------------------
VIOLATION OF G(X)             0.000000000000
---------------------------------------------
g(   1) =      0.00000000  (EQUALITY CONSTR)
g(   2) =      0.66666667  (IN-EQUAL CONSTR)
g(   3) =      0.33333333  (IN-EQUAL CONSTR)
---------------------------------------------        BOUNDS-PROFIL
x(   1) =                 1.000000000000000;  %  XL_________________
x(   2) =                 2.000000004059014;  %  _______x_________
x(   3) =                 3.000000000000000;  %  _____________x___
x(   4) =                 4.000000000000000;  %  _________________XU
```

The MIDACO screen file is identical to the output displayed on the console/command window (except for Excel). The MIDACO screen and solution file layout is basically identical in all programming languages. Depending different languages, minor differences only apply to the vector index of F,G, and X and the comment symbol used for the bounds profiler. All abbreviations used in the MIDACO screen and solution file are explained in Table 1.

Table 1: Abbreviations used in the MIDACO screen and solution output files (Figure 2)

OBJECTIVES	Number of objective functions
PARALLEL	Number of parallel processed problem function calls (also called *co-evaluation*)
N	Number of variables in total
NI	Number of integer variables $0 \leq$ NI $\leq$ N
M	Number of constraints in total
ME	Number of equality constraints $0 \leq$ ME $\leq$ M
MAXEVAL	Maximum number of function evaluation (stopping criteria, see Section 3)
MAXTIME	Maximum CPU-time budget for execution (stopping criteria, see Section 3)
PRINTEVAL	Print frequency of the current best solution
SAVE2FILE	Create text-file output [0=No, 1=Yes, 2=Yes + create history file]
PARAM	Parameter for MIDACO tuning (default = 0, see Section 4)
EVAL	Number of performed function evaluation
TIME	Number of performed CPU-time Seconds
F(X)	Current best objective function value, found after EVAL evaluation
VIOLATION	Violation of constraints: measured as L1-Norm (Wikipedia) over vector G
IFLAG	Information flag used by MIDACO to indicate final status, warnings or errors
F(i)	Numerical value for individual objective F_i
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. This is an important feature as it gives the user the chance to access the full solution already during runtime and adds security in case an optimization run gets interrupted for some reason (e.g. server reboot or electricity black-out). 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. All objectives F(i) and 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 size enlargement of the solution file.

2.1 PRINTEVAL and SAVE2FILE

PRINTEVAL is the critical parameter to control how often the current best solution is printed on the screen. Note that this parameter is completely independent from any algorithmic iteration within MIDACO. Therefore the user can freely set PRINTEVAL in such a way, that the output frequency is convenient for display. Small values (e.g. 10, 123, 500) for PRINTEVAL will result in a faster output frequency. Large values (e.g. 10000, 100000 or 1000000) 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 CPU-time intensive problems, or for debugging purposes. In general it is recommended to set large values for PRINTEVAL. This way the user gets a better overview on the optimization progress and MIDACO runs a little bit faster (because the printing command needs less often to be executed). For most real-world applications it is sufficient enough to set PRINTEVAL in such way that a new printout line happens only every couple of seconds or minutes.

The creation of the output files is optional. If SAVE2FILE is set to zero, no output file will be created. If no output at all is desired (for example if MIDACO should be silently embedded within a high-level software and only the final solution vector X should be further numerically processed in such high-level software), all visual output can be suppressed by setting PRINTEVAL to zero.

Therefore: Setting **PRINTEVAL=0** and **SAVE2FILE=0** **completely silents MIDACO.**

2.2 Solution History File

Additionally to the screen and solution file, MIDACO can produce a complete history of all evaluated iterates X and their corresponding objective F(X) and constraint G(X) values. If the SAVE2FILE option is given a value greater than one, MIDACO will automatically create a file named "MIDACO_HISTORY.TXT". This file will store up as many solutions as indicated by the value of SAVE2FILE. For example, if SAVE2FILE = 1000 is set, MIDACO will store the latest history of 1000 solutions in this file. In case all solutions should be saved, this can be achieved by setting SAVE2FILE a sufficient large value. For example, if SAVE2FILE = 10000000 is set, MIDACO will store up to 10 million solutions in the history file.

Creating a history file can be useful for applications which are CPU-time intensive and were full access to all available evaluation results is desired. Especially if parallelization is applied, the creation of a history file offers a way to keep track on all processed evaluations.

The solution format which is used in the history file is identical to the format used for the pareto front file "MIDACO_PARETOFRONT.TXT" created for multi-objective problems. Therefore the content of the history file can also be plotted with the PlotTool (see Section 6).

Note: In case of **parallelization**, the SAVE2FILE value will count the number of *blocks* rather than the individual evaluation. For example, if P=30 and SAVE2FILE=10 then up to 300 (=30×10) solution will be stored in the history file, rather than just 10 solutions. Be aware that the file size of the history can become **very large**, if many solutions are stored.

3 MIDACO Stopping Criteria

The stopping criteria for MIDACO can be categorized into two groups: Hard limit criteria and algorithmic criteria. Hard limit criteria are MAXTIME and MAXEVAL which cause MIDACO to stop its optimization process based on a maximal budget of CPU-time or the number of function evaluation. Algorithmic criteria are FSTOP, ALGOSTOP and EVALSTOP which cause MIDACO to stop based on some algorithmic decision. All stopping criteria can be freely combined by the user to fit a specific purpose at hand. The following sub-sections illustrate each criteria in detail and furthermore give some example setup scenarios.

3.1 Hard Limit Criteria

Here stopping criteria are discussed which are based on a hard limit, such as time or evaluation.

3.1.1 MAXTIME

The MAXTIME criteria defines a maximal CPU-time budget measured in seconds. Freely set this stopping criteria to any value. Setting a very large value (like 1000000) practically disables this criteria. Most example problems provided with MIDACO use a dummy value of one day, which is 60*60*24. For quick orientation, below table displays usual time scales measured in seconds.

Minute	:	60	=	60
15 Minutes	:	60*17	$<\approx$	1000
Hour	:	60*60	=	3600
2.5 Hours	:	60*60*2.5	$<\approx$	10000
Day	:	60*60*24	=	86400
27 Hours	:	60*60*27	$<\approx$	100000
Week	:	60*60*24*7	=	604800

3.1.2 MAXEVAL

The MAXEVAL criteria defines a maximal budget of problem function evaluation. It is a distinctive feature of the MIDACO software implementation to be able to stop exactly after any given number of evaluation (e.g. 123456). The user can therefore freely choose any arbitrary integer value for MAXEVAL.

MIDACO can quickly process millions of function evaluation within seconds, if the actual function evaluation is computationally cheap (like for benchmark problems). This means that for fast calculating applications, evaluations limits of 10000000 (ten million) or 100000000 (hundred million) are often reached within minutes. For those fast calculating applications it can be desirable to completely switch of the MAXEVAL stopping criteria. Therefore the MAXEVAL criteria will apply only for values lower than 999999999 (*"nine times nine"*). If the MAXEVAL stopping criteria is assigned any value greater or equal to 999999999, the MAXEVAL criteria is completely disabled.

Note that in contrast to above special scenario the important case of CPU-time expensive applications, where only a few thousands or just hundreds of evaluation can be calculated within reasonable time, is discussed separately in Section 7.

3.2 Algorithmic Criteria

Here stopping criteria are discussed which are based on an algorithmic decision.

3.2.1 FSTOP

The FSTOP parameter is enabled if any value except exactly zero (0.0E+0) is assigned to it. If MIDACO reaches a feasible solution with an objective function value lower or equal to FSTOP, MIDACO will stop. Note that this stopping criteria refers the first objective function in case of multi-objective problems. It is important to note that MIDACO will be strict about the FSTOP value, therefore MIDACO does not add any tolerance to the FSTOP value. If zero is the desired value for FSTOP, some tiny value like 0.000000001 can be used instead as FSTOP value.

3.2.2 ALGOSTOP

The ALGOSTOP parameter is enabled if any positive integer value (e.g. 1,2,3,...) is assigned to it. This criteria will measure the algorithmic improvement between MIDACO internal ACO restarts. The value of ALGOSTOP defines the maximal number of consecutive MIDACO internal ACO restarts without improvement of the (feasible) objective function value. For example: If ALGOSTOP=10 is set, than MIDACO will perform its optimization search until 10 consecutive internal ACO restarts did not further improve the current solution.

The higher the value for ALGOSTOP is set (like 10, 50, 100 or higher) the higher the chance that MIDACO reached the global optimal solution. In such regard this stopping criteria is the most advanced to indicate global optimality. The significant drawback of this stopping criteria is that it might require many (normally thousand or even millions) of function evaluation. It is therefore only suitable for applications which are CPU-time cheap to evaluate. When experimenting with the ALGOSTOP criteria, values such as 1, 5, 10 or 30 might be used at first to get a feeling for the run-time effect on a specific application.

For applications with CPU-time expensive evaluation the EVALSTOP criteria is more appropriate.

3.2.3 EVALSTOP

The EVALSTOP parameter is enabled if any positive integer value (e.g. 1,2,3,...) is assigned to it. It works similar to the ALGOSTOP criteria but with the significant difference that it does not consider complete MIDACO internal ACO restarts but individual function evaluation. For example: If EVALSTOP=999 is set, than MIDACO will perform its optimization search until 999 consecutive function evaluation did not further improve the current solution.

The lower the value for EVALSTOP is set (like 1000, 100 or lower) the faster MIDACO will stop. In case EVALSTOP=1 is set, MIDACO will stop immediately after any function evaluation which did not improve the current solution. Therefore for very small EVALSTOP values (like 1,2,3,...) MIDACO will stop very fast. Goal of this stopping criteria is to provide an algorithmic stopping criteria that is not as expensive as ALGOSTOP in the number of required function evaluation, but that is still based on an algorithmic measure. When experimenting with the EVALSTOP criteria, values such as 10000, 1000 or 500 might be used at first to get a feeling for the run-time effect on a specific application.

The EVALSTOP criteria can further be fine-tuned by specifying the precision (in relative percentage) applied to measure if a new solution is considered as improvement or not. The default precision for EVALSTOP is 0.001, which is 0.1% in relative percentage. In case a different precision should be used, such precision should be appended as floating point extension to the EVALSTOP value. For example: MIDACO should stop after 333 consecutive function evaluation without improvement of 0.25% relative percentage of the objective function value. Then setting EVALSTOP = 333.0025 will enable such stopping criteria. If no specific floating point extension is given to EVALSTOP, the default precision of 0.001 is applied automatically.

3.3 Example Scenarios

Here some example scenarios are given how to set up a single or several stopping criteria together.

3.3.1 Single Evaluation

If MAXEVAL=1 is set, MIDACO will only perform a single function evaluation. This is by definition the starting point X provided by the user. This scenario is useful to verify a problem implementation, as it give the user the chance to check in detail all objective and constraint function values reported for the starting point X to be reasonable. This option is also useful to re-produce a given solution (thus re-evaluating it).

3.3.2 CPU-Time expensive application

This scenario exemplifies a CPU-time expensive application where only a low number of function evaluation are available (for example a complex machine simulation model). A suitable setup for such application might look something like this:

MAXTIME	=	50000
MAXEVAL	=	999999999 ($\rightarrow$ disabled)
FSTOP	=	0 ($\rightarrow$ disabled)
ALGOSTOP	=	0 ($\rightarrow$ disabled)
EVALSTOP	=	50

Above setup assigns a hard time limit of 50000 seconds (about half a day) and further addresses an EVALSTOP=50 stopping criteria, in the hope that such stopping criteria is reached before the actual time limit is reached. All other criteria are disabled.

3.3.3 CPU-Time cheap application

This scenario exemplifies a CPU-time cheap application where a high number of function evaluation can quickly be calculated (like in academic benchmark problems). A suitable setup for such application might look like something like this:

MAXTIME	=	60*60*24
MAXEVAL	=	10000000
FSTOP	=	0.00000001
ALGOSTOP	=	200
EVALSTOP	=	0 ($\rightarrow$ disabled)

Above setup assigns a hard function evaluation limit of 10000000 (ten million) and further addresses an FSTOP=0.00000001 and ALGOSTOP=500 criteria. Thus, MIDACO will stop if a (feasible) solution with objective lower or equal 0.00000001 is found, or the MIDACO internal ACO performed 200 consecutive restarts without further solution improvement or the evaluation budget is spent. This setup is not concerned with the actual time and thus practically disables the MAXTIME criteria by given it a full day.

3.3.4 Infinite Run

The MIDACO software is constructed in such way that it is able to potentially run forever, except the text file output it will not accumulate any data or memory and no internal algorithmic element will run against some bound and crash.

A setup where MIDACO is practically running forever can be achieved by setting MAXEVAL and MAXTIME to the huge value of 999999999 (*"nine times nine"*) while keeping FSTOP, ALGOSTOP and EVALSTOP by their default value (zero). Such setup might appear absurd at first, but is commonly used in practice. By disabling all automatic stopping criteria for MIDACO the user takes the final decision when to stop the optimization run (e.g. by shutting down the program/computer) in his/her own hand, giving MIDACO the highest chance to find the global optimal solution (or best spread of pareto points).

Note that the MIDACO_SOLUTION.TXT and MIDACO_PARETOFRONT.TXT files are updated with the latest solution(s) at each PRINTEVAL function evaluation. Therefore, even under an infinite run scenario the user has always access to the solutions and can already further process them or plot the pareto front while the actual MIDACO optimization run is still ongoing.

4 MIDACO Parameter

MIDACO offers several parameters to customize its performance and behavior. The individual parameters are explained in the following subsections. The default value for all parameter is zero.

4.1 PARAM(1) : ACCURACY

This parameter defines the accuracy tolerance for the constraint violation. MIDACO considers an equality constraint to be feasible, if $|G(X)| \leq$ PARAM(1). An inequality is considered feasible, 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 many or difficult constraints, it is recommended to start with some test runs using a less precise accuracy (e.g. PARAM(1)=0.1 or PARAM(1)=0.05) 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.

4.2 PARAM(2) : 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).

MIDACO runs are reproducible, if performed with the same seed and executed on the same machine with identical compiler settings. Note that any change in either the hardware (CPU) or software (e.g. compiler version or compile flags) can and will likely change the results. This is due to the highly sensitive nature of the internal random number generator. The main advantage of a user specified random seed is, that promising runs can be reproduced. For example, if a run was unintentionally interrupted and should be restarted again. Another advantage is for debugging purposes.

The impact of the seed normally varies with the complexity of the problem. In general, the more complex the problem, the bigger the influence of the seed can be. For difficult problems it is therefore often a more promising strategy to execute several short runs of MIDACO with different random seeds, rather than performing only one very long run.

4.3 PARAM(3) : FSTOP

This parameter enables a stopping criteria for MIDACO. The FSTOP stopping criteria is based on an objective function value to be reached. Full details on the FSTOP parameter are described in Section 3.2.1. For multi-objective problems the FSTOP values applies for the first objective.

4.4 PARAM(4) : ALGOSTOP

This parameter enables a stopping criteria for MIDACO. The ALGOSTOP stopping criteria is based on the algorithmic process of MIDACO. Full details on the ALGOSTOP parameter are described in Section 3.2.2.

4.5 PARAM(5) : EVALSTOP

This parameter enables a stopping criteria for MIDACO. The EVALSTOP stopping criteria is based on the algorithmic process of MIDACO taking account the number of function evaluation. Full details on the EVALSTOP parameter are described in Section 3.2.3.

4.6 PARAM(6) : FOCUS

This parameter forces MIDACO to focus its search process around the current best solution and thus makes it more *greedy* or *local*. This **is one of the most powerful parameters** and widely applicable. 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. If PARAM(6) is not equal zero, MIDACO will apply an upper bound for the standard deviation of its Gauss PDF's (see Section , Figure 1). The upper bound for the standard deviation for continuous variables is given by (XU(i)-XL(i))/FOCUS, whereas the 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 FOCUS value, the closer MIDACO will focus its search on the 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 disables 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.

4.7 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.

4.8 PARAM(8) : KERNEL

This parameter allows the user to fix the number of kernels within MIDACO's multi-kernel Gauss PDF's (see Section , 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 2 some examples of possible ants/kernel settings are given and explained below.

Table 2: 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.

4.9 PARAM(9) : 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 $USD). 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. More information on the oracle penalty method can be found at the Oracle Penalty Method website.

4.10 PARAM(10) : PARETOMAX

This parameter defines the maximal number of non-dominated solutions (also called *pareto points*) stored by MIDACO. The default value used by MIDACO is 1000 (if PARAM(10)=0 is set). User can freely set any arbitrary large integer for PARETOMAX, for example PARAM(10)=333 or PARAM(10)=100000. Note that larger PARETOMAX values will require more memory and will normally slow slow down the internal calculation time of MIDACO, due to increased pareto-dominance filtering efforts. For many applications a PARETOMAX value <= 1000 is sufficient. The user can also specify smaller values, for example PARAM(10) = 30, which will normally speed up the internal calculation time of MIDACO.

4.11 PARAM(11) : EPSILON

This parameter defines the precision used by MIDACO for its multi-objective pareto-dominance filter. If the default PARAM(11)=0 is set, a value of EPSILON=0.001 is used for problems with two objectives and a value of EPSILON=0.01 is used for problems with three or more objectives. The lower the EPSILON value, the higher the chance that a new solution is introduced into the pareto front. Therefore the EPSILON value can have a great influence on the amount of pareto points stored by MIDACO and also it's internal calculation time.

For most applications, a value of EPSILON larger or equal to 0.001 is sufficient. Smaller values, such as PARAM(11)=0.00001 or PARAM(11)=0.00000001 will normally result in many (!) more pareto points reported. However, those pareto points are only slightly different from each other and might therefore not provide much useful information. A special case in multi-objective optimization are many-objective problems, which consider four or more objective functions. Those problems often easily generate many (thousands) of non-dominated solutions. For many-objective problems it can be useful to assign a larger EPSILON value, such as PARAM(11)=0.01 or PARAM(11)=0.1. Using such high EPSILON value will force MIDACO to store and report only pareto points with a significant difference in at least one of their objectives. Note that using larger EPSILON values will also greatly speed-up MIDACO's internal calculation times.

4.12 PARAM(12) : BALANCE

The BALANCE parameter is relevant for multi-objective problems and has a great impact. It defines on what part/area of the pareto front MIDACO should focus its main search effort. By default, MIDACO will focus most of its search effort on that part of the pareto front, which offers the best equally balanced trade-off between all objectives. Using the BALANCE parameter, this focus can be shifted to any part of the pareto-front. See Section 5.2 for a full explanation of this parameter with detailed examples.

4.13 PARAM(13) : CHARACTER

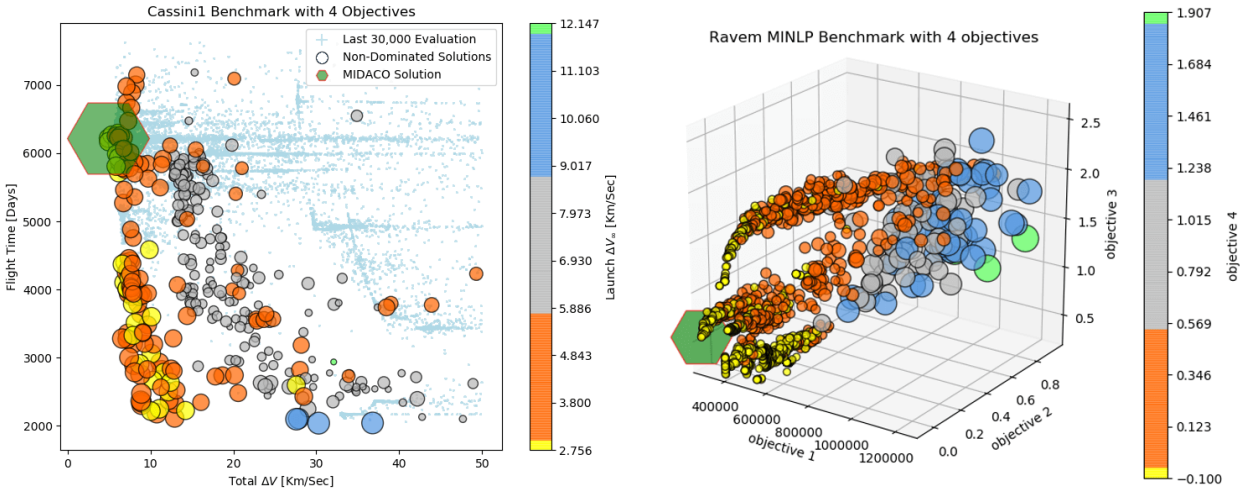
The character parameter allows to activate MIDACO internal parameter settings. MIDACO offers the following three pre-defined characters:

CHARACTER = 1	:	MIDACO internal parameters for continuous problem types
CHARACTER = 2	:	MIDACO internal parameters for combinatorial problem types
CHARACTER = 3	:	MIDACO internal parameters for *All-Different* problem types

If PARAM(13)=0 is set, MIDACO will decide by itself if the internal parameters for the continuous or combinatorial problem is chosen. The internal parameters for continuous problem types will enable a more fine-grained search process, while the internal parameters for combinatorial problem types will enable a more coarse-grained search process. A special case are *All-Different* problem types. Those problems require that all integer variables must contain a different value. A famous example for *All-Different* problems is the traveling salesman problem (TSP). In case of *All-Different* problems the CHARACTER=3 should be enabled. If CHARACTER=3 is set, MIDACO will generate only solutions which automatically satisfy the *All-Different* constraint. This means the *All-Different* constraint does not need to be explicitly formulated and provided by the vector of constrains G(X). MIDACO will take care of it automatically. When using the *All-Different* character it is to note that the staring point X must already satisfy the *All-Different* constraint, otherwise an IFLAG=402 error is raised.

Note that MIDACO's *All-Different* character can also be used for mixed integer problems. In such case the all-different constraint affects all integer variables, but does not affect any of the continuous variables.

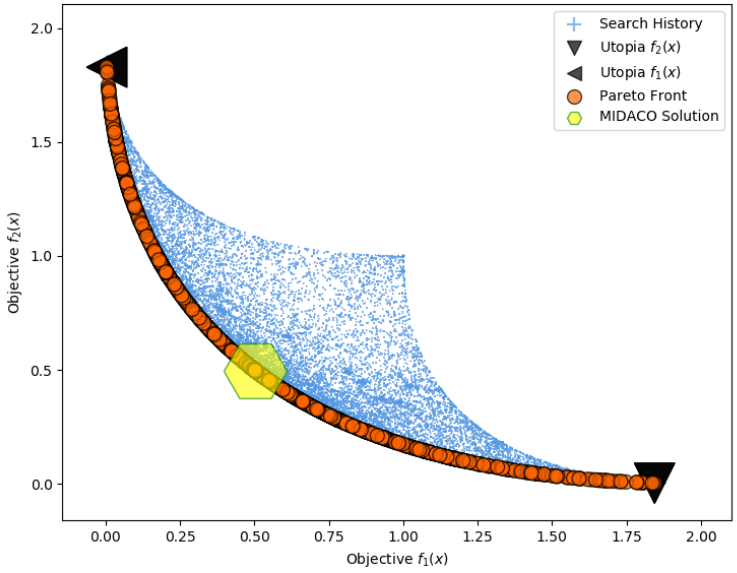
5 Multi-Objective Optimization



Multi-objective optimization considers several objective functions simultaneously. In contrast to single-objective optimization, where (normally) a single solution exists as global optimum, in multi-objective optimization there is (normally) not a single solution that simultaneously optimizes each objective. Instead, a set of *non-dominated*, also called *pareto-optimal*, solutions exists, that represents a **trade-off curve** (also called pareto front) between the individual objectives. As example, consider the following multi-objective toy problem with two objectives $f_1(x)$ and $f_2(x)$:

$$\text{Minimize} \quad \begin{cases} f_1(x) = (x_1 - 0)^2 + (x_2 - 0)^2 \\ \\ f_2(x) = (x_1 - 1)^2 + (x_2 - 1)^2 \end{cases} \qquad \text{with} \qquad x_1, x_2 \in [0, 1]$$

Then the pareto front to this toy problem looks as follows:



Multi-objective optimization with MIDACO is straight forward. The user only needs to indicate the number of objectives via the corresponding MIDACO input parameter in the problem dimension declaration (*STEP 1.A* in all example files). For previous toy problem, this looks as follows:

```
% STEP 1.A: Problem dimensions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem.o  = 2; % Number of objectives
problem.n  = 2; % Number of variables (in total)
problem.ni = 0; % Number of integer variables (0 <= ni <= n)
problem.m  = 0; % Number of constraints (in total)
problem.me = 0; % Number of equality constraints (0 <= me <= m)
```

MIDACO will then **fully automatically** solve the multi-objective problem by delivering the entire pareto-front and it will particularly highlight a single point of the pareto front as *MIDACO Solution*. The MIDACO console screen for solving above toy problem looks as follows:

Current number of non-domiated solutions, also called ***pareto-points***

```
----------------------------------------
| OBJECTIVES    2 | PARALLEL         1 |
|--------------------------------------|
| N             2 | MAXEVAL     100000 |
| NI            0 | MAXTIME      86400 |
| M             0 | PRINTEVAL    10000 |
| ME            0 | SAVE2FILE        1 |
|--------------------------------------|
| PARAMETER:    All by default (0)     |
----------------------------------------

[     EVAL,    TIME]  MULTI-OBJECTIVE PROGRESS  VIOLATION OF G(X)
-------------------------------------------------------------------   [PARETO]
[        1,       0]   PRO:          2.00000000   VIO:     0.000000   [     1]
[    10000,       0]   PRO:        -13.52562586   VIO:     0.000000   [   286]
[    20000,       0]   PRO:        -16.00900246   VIO:     0.000000   [   327]
[    30000,       0]   PRO:        -16.78937764   VIO:     0.000000   [   361]
[    40000,       0]   PRO:        -16.78937764   VIO:     0.000000   [   398]
[    50000,       1]   PRO:        -20.30755765   VIO:     0.000000   [   472]
[    60000,       1]   PRO:        -20.30755765   VIO:     0.000000   [   529]
[    70000,       1]   PRO:        -22.59627173   VIO:     0.000000   [   551]
[    80000,       1]   PRO:        -25.16010495   VIO:     0.000000   [   574]
[    90000,       1]   PRO:        -25.16010495   VIO:     0.000000   [   591]
[   100000,       1]   PRO:        -26.78532572   VIO:     0.000000   [   597]

OPTIMIZATION FINISHED  --->  MAXEVAL REACHED


        BEST SOLUTION FOUND BY MIDACO
-------------------------------------------
EVAL:    100000,  TIME:    1.00,  IFLAG:   1
-------------------------------------------
PROGRESS                -26.785325716864353
-------------------------------------------
NUMBER OF PARETO POINTS                 597
-------------------------------------------
f[   0] =                0.495602836738631
f[   1] =                0.504550856899625
-------------------------------------------
x[   0] =                0.491982059221132;  # __________x__________
x[   1] =                0.503543930698371;  # __________x__________
```

5.1 The Multi-Objective Progress (PRO) Function

When solving multi-objective problems, MIDACO displays a *Multi-Objective Progress (PRO)* function value in the main column of its screen output. This function is a **unique feature** of MIDACO and acts as a measurement to monitor the overall multi-objective optimization process. It's display and behavior is intentionally similar to those of a single-objective optimization function, which makes the transition from a single-objective problem to a multi-objective as convenient as possible.

Without loss of generality, MIDACO considers all objectives to be minimized. A lower value in the multi-objective progress function therefore represents a positive progress. When using the BALANCE parameter default value (zero), or a value smaller than one, the exact meaning of the value of the multi-objective progress function is only relevant for internal purposes of MIDACO. Therefore it has no direct connection to the objective function values of the problem. Instead, the purpose of displaying the PRO value is to give the user a feedback on the progress: An improvement ($\rightarrow$ lower value) in the PRO value indicates that some kind of improvement on the entire pareto front has been made. Normally this will mean that more or better pareto points have been discovered. It also indicates if a direct improvement on the MIDACO solution (which is a single point of the pareto front) has been made. Such improvement may either be a lower function value in some objective and/or a re-location of the MIDACO solution among the pareto front.

The exact calculation of the PRO function is complex and based on the *Utopia-Nadir-Balance* concept introduced in [43] introduced particular for **many-objective** optimization problems.

5.1.1 Set BALANCE exclusively to one objective

A special case is given if the BALANCE parameter (see below Section 5.2) is set exclusively to one out of the multiple objectives. In such case, the displayed PRO function value is **identical** to the objective function value to which the BALANCE parameter has been assigned to. This way the user can directly monitor the progress on the selected objective function via the PRO value. **This strategy is often promising**, if one of the several objectives is significantly harder to solve than the others, or if one objective is of much great importance than the others.

5.2 The BALANCE parameter

By default, MIDACO will particularly focus its search effort on that point of the pareto front which represents the **best equally balanced** solution among all objectives. In case the search effort should be focused on a different part of the pareto front, this can be achieved by the BALANCE parameter (see also Section 4.12).

In case the search effort should focus exclusively on one out of the multiple objectives, this can easily be achieved by setting the BALANCE parameter equal to the index number of the desired objective. For example, if BALANCE = 1.0 is set, MIDACO will focus its search effort exclusively on the first objective (see above Sec 5.1.1). If Balance = 2.0 is set, MIDACO will focus its search effort exclusively on the second objective. And so on for the third, fourth, fifth,... objective.

In case the search effort should be **fine-tuned** on a particular part of the pareto front which represents some non-equal priority between objectives, this can be achieved by passing that information via the **individual decimal digits** of the BALANCE parameter. In such case, each individual objective can be assigned an *importance* (or *priority*) value of 0 to 9 and this value has to be placed on the corresponding decimal position of the BALANCE parameter. This may sound complicated at first, but is in fact very easy. For example, on a two objective problem the second objective should be given **twice as much importance** as the first one. Then the following BALANCE parameter settings will achieve this: BALANCE = 0.12 or 0.24 or 0.36 or 0.48. In all those cases the BALANCE parameter is set in such way that the second decimal digit (corresponding to the second objective) is as twice as high as the first digit (which corresponds to the first objective).

As another example consider that the first objective should be given **four times more importance** than the second objective. Then the following BALANCE parameter settings will achieve this: BALANCE = 0.41 or 0.82, because the first digit ($\rightarrow$ importance of first objective) is four times larger than the second digit ($\rightarrow$ importance of second objective).

As a last example consider that the second objective should be given **nine times more importance** as the first objective. Then the following BALANCE parameter settings will achieve this: BALANCE = 0.19, because the second digit ($\rightarrow$ importance of second objective) is nine times higher than the first digit ($\rightarrow$ importance of first objective). Figure 5.2 graphically illustrates the impact of varying BALANCE parameters for the position of the MIDACO solution among the pareto front of previously considered toy problem.

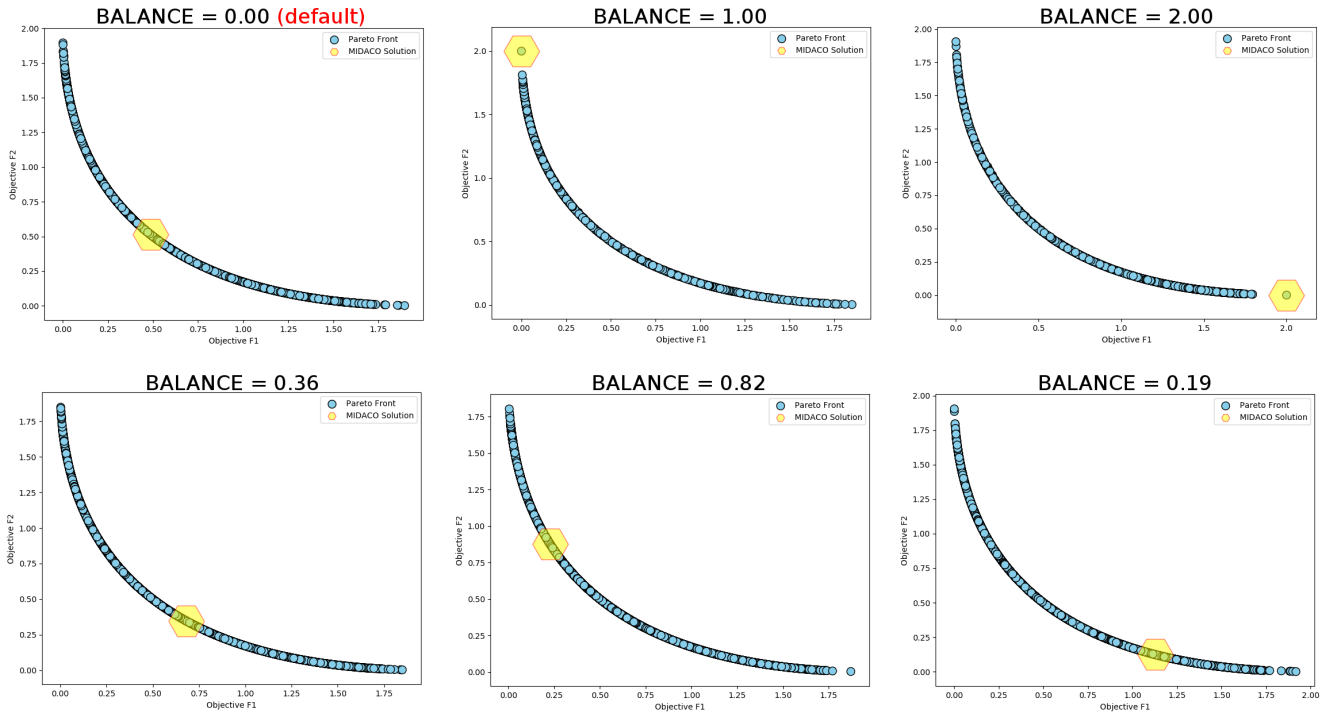
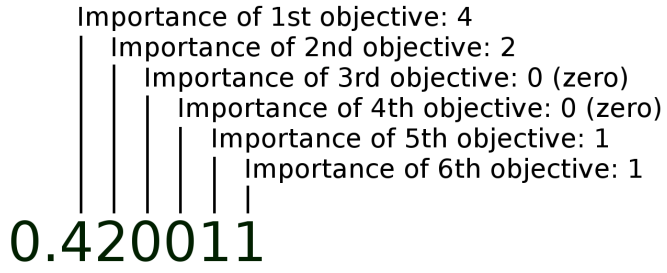


Figure 3: Impact of the BALANCE on the MIDACO solution position on the pareto front

Note that for numerical reasons, fine-tuning of the BALANCE parameter works only on the first eight objectives. If a fine-tuned BALANCE parameter is submitted to MIDACO, each digit below the eight's position will automatically be assigned a **zero importance**. Further note that it is possible to assign a zero importance to any objective. This can be useful in case of many objective problems, where some objectives should only be monitored (but be excluded from the search effort). For example, consider a problem with 6 objectives, where the first objective should have the highest importance, the second should have half that importance, the third and forth objective half zero importance and the fifth and sixth objective should have half of the importance as the second objective. Then all this information can be passed to MIDACO as a single number by setting BALANCE = 0.420011, because:



Note that fine-tuning the BALANCE parameter is especially useful for **many-objective** problems, which are notoriously difficult to solve and where focusing on a particular part of the pareto front can be more effective than trying to obtain optimal convergence on the entire pareto front.

Important note: For numerical reasons, it is recommend to add *trailing zeros* to the BALANCE parameter value, in case fine-tuning is applied. For example, instead of passing the BALANCE value "0.15" as is to MIDACO, the value "0.15000000" should be used. The reason is that in some programming languages (e.g. C++ and Fortran), the last digit might otherwise be replaced by its dual-representation. For above example "0.15" this would be "0.14999999" and would therefore pass a wrong information to MIDACO, which can lead to significantly different results.

5.2.1 Disable full front search capability

A further fine-tuning of the BALANCE parameter is possible by adding a negative "-" flag to the numerical value. For example, setting BALANCE = -0.12000000 instead of 0.12000000. The effect of the negative "-" flag is that MIDACO will disable all its internal full front search heuristics and therefore focus even a little stronger on the particular desired part of the pareto front.

Setting the negative "-" flag is also possible when focusing exclusively on one out of the multiple objectives. For example, setting BALANCE = -1.0 instead of 1.0, when focusing only on the first objective. However, the effect of the setting a negative "-" flag to the BALANCE parameter is generally weak and will normally only be visible if many thousands of function evaluations are performed.

5.3 Pareto Front Data

When solving multi-objective problems and setting the SAVE2FILE parameter >= 1, MIDACO will automatically create a text file named **MIDACO_PARETOFRONT.TXT**. This file contains the entire pareto front approximation, and is created at each PRINTEVAL event (see Section 2.1). The PlotTool (see Section 6) can be used to graphically illustrate the data of the MIDACO_PARETOFRONT.TXT file.

Alternatively to the text file, user can gain access to the pareto front data via the "pf" array, which is an input/output argument to the MIDACO source code routine (or library file in case of higher languages). The "pf" array stores the entire pareto front information and is in used to create the above mentioned text file. The very first element of the "pf" array stores the number of pareto points. For programming languages starting with a zero-index (e.g. Python), this information is given by pf[0]. For programming languages starting with a one-index (e.g. R), this information is given by pf[1]. Once the number of stored pareto points (called *psize*) is known, the individual solutions can be accessed as follows (pseudo code, starting with one-index):

```
psize = pf[1]  # number of pareto points stored in array pf[]

pfmax = 1000   # default value of maximal number of pareto points

for k=1:psize

      for i=1:o   # objectives

            f[i]  =  pf[ 2 + o*(k-1)+i-1 ]

      for i=1:m   # constraints

            g[i]  =  pf[ 2 + o*pfmax + m*(k-1)+i-1 ]

      for i=1:n   # variables

            x[i]  =  pf[ 2 + o*pfmax + m*pfmax + n*(k-1)+i-1 ]
```

User with an interest to access the pareto front data directly via the "pf" array may also consult the "print_paretofront" subroutine code given in their respective programming language or MIDACO gateway code. The "print_paretofront" subroutine executes above pseudo-code to create the pareto front text file and can be copied and modified for further purposes.

5.4 Number of Pareto Points

The number of pareto points can be influenced via the PARETOMAX and EPSILON parameters.

By default, MIDACO will collect up to 1000 pareto points. This maximal limit can be changed by setting the PARETOMAX parameter to any number. For example, if PARETOMAX = 5000 is set, MIDACO will store up to 5000 points. Or if PARETOMAX = 60 is set, MIDACO will store only up to 60 points.

The EPSILON parameter (see Section 4.11) defines a tolerance precision that influences if a solution is included into the pareto front or not. The smaller the EPSILON value, the more likely a solutions is included. A smaller EPSILON value will therefore normally result in (many) more pareto points being collected. However, those solutions might only slightly differ from each other.

In order to collect as much pareto points as possible, a large PAERTOMAX value should be combined with a small EPSILON value. For example PARETOMAX = 10000 and EPSILON = 0.00001 will normally result in many collected pareto points. One drawback of collecting large amounts of pareto points is that the internal MIDACO runtime may significantly increase. Another drawback is that many of those pareto points might only slightly differ and thus not offer much insight.

In order to speed up MIDACO's internal runtime on multi-objective problems, this can be achieved by combining a smaller PARETOMAX value with a higher EPSILON value. For example setting PARETOMAX = 100 and EPSILON = 0.005 will likely significantly speed-up the internal runtime of MIDACO while still delivering sufficiently well distributed pareto points. Speeding up MIDACO's internal runtime on multi-objective problems can be especially beneficial on problems with many objectives.

5.4.1 Reduced filtering for many-objective optimization

In case of many-objective problems where the BALANCE parameter is fine-tuned with some objectives having assigned a zero importance (see Section 5.2), those objectives with zero importance can be excluded from the pareto-dominance filtering process. This can be achieved by adding a negative "-" flag to the numerical value of the PARETOMAX parameter. For example, if PARETOMAX = -500 is set instead of 500, MIDACO will collect up to 500 pareto points, whereas the pareto-dominance filtering criteria is exclusively applied to those objectives having a positive importance indicated by the BALANCE parameter.

Using this feature will **reduce the number of pareto points to the relevant set** of solutions that are pareto-optimal only in regard to those objectives, that have been assigned a positive importance. This feature is therefore useful for problems with many objectives, where the number of pareto points can quickly become very large and unmanageable. It can further help to speed-up MIDACO's internal runtime.

6 PlotTool

The PlotTool is a **Windows** executable program that is based on the Matplotlib [21] graphic library. The PlotTool can generally be used to graphically illustrate pareto front data from various sources (including other optimization algorithms than MIDACO). In particular, it can be used to plot the **MIDACO_PARETOFRONT.TXT** and the MIDACO_HISTORY.TXT file. The PlotTool.exe can also be executed in **Linux** and **Mac** using Wine.

> Note that some Antivirus software may raise a ***false positive*** on the PlotTool.exe

It is recommended (but not necessary) to store the PlotTool.exe in the same folder as where the source files are located. Double-clicking the executable will launch the program. Note that launching might take some time due to Windows security checking's. The main functionalities of the PlotTool should be self explanatory. In the following some advanced features are discussed.

6.1 Values, Colors, Colormaps and LaTeX support

All numerical values given as drop-down menu choice can freely be changed. For example, it is possible to enter *Marker Size*=123 or *Transparency*=0.456. This works also for large objective and constraint/variable indexes. For example *objective 33*, *constraint 44* or *variable 55*. Besides the drop-down choice of single colors, any named color (e.g. silver, gold, aqua, lightpink, darkgreen) or HEX color code (e.g. #ff5733) can be entered. For a complete list of named colors see:

https://matplotlib.org/examples/color/named_colors.html

Any Matplotlib supported colormap can be entered, if the original colormap name is entered with an @-symbol in front. For example: @viridis, @Reds, @winter, @summer, @Pastel1. Note that names are lower and upper case sensitive. Any colormap can be reversed by appending a "_r" at the end of its name (e.g. "rainbow_r"). For a complete list of named colors see:

https://matplotlib.org/examples/color/colormaps_reference.html

Title and legend texts support most common LaTeX syntax commands. For example the title `"$\alpha-\beta-\gamma$ Design"` will be displayed as "$\alpha - \beta - \gamma$ Design".

6.2 Additional Data Files and Background position

Up to three additional source files can be used in addition to the main source file. Each file can either contain several solutions (e.g. history file), or just a single one. The plot hierarchy will put the main source file in the foreground and the last add file in the background. Optionally the main source file data can be place in the *Background Position*, if that option box is ticked.

6.3 Solution export and re-import

An **important feature** of the PlotTool is it's capability to export individual solutions from the source files. On a 2D plot, place the mouse cursor on the desired solution point and press the **right-click** mouse button. A dialog will appear, asking for confirmation to export the solution with respective x-axis and y-axis coordinates. If confirmed, the PlotTool will search among all source files (main and additional ones) that specific solution, which is closest to the position of the mouse cursor. If successful, a new dialog window will pop-up, asking to store the solution into a text file. This text file will then contain the entire solution information (F,G and X) and present it in several common programming language formats. From that text file, the solution can then be investigated and further processed by copy-and-paste, for example as starting point for a refinement run.

This feature is particular useful for source files with many (thousands) of pareto-optimal solutions. Note that the export search process can take some seconds on large source files.

This feature is also useful to **highlight a specific pareto point** in the plot. Once a solution was exported into a text file, it can easily be **re-imported** into the plot using the "ADD FILE" option. As additional source file, the point can be given a different size, color or marker to distinguish it.

6.4 Save, load and reset

The entire PlotTool user settings can be saved and loaded. The default name of the file which stores the settings data is PlotTool.set. This file can also be manually changed and includes additional options, like GUI positioning and changing the number of colorbar marks. Changes take effect after restarting the PlotTool with the modified PlotTool.set file located in the same folder.

All settings can be reset to their default values, using "LOAD" > "**Reset All to Default**".

6.5 Live mode

The LIVE command will cause the PlotTool to constantly check for updates in the source files and display them in real-time in the plot. The LIVE command is more fault-tolerant then the PLOT command, for example the LIVE command will also execute, even if no source file is present yet. Note that the LIVE mode requires some CPU power and might therefore (slightly) reduce the MIDACO optimization execution speed.

Warning: In some cases (e.g. VBA and MS-Visual) the LIVE mode can cause the MIDACO optimization run to crash. This is due to a file access conflict, when both, the PlotTool and MIDACO try to gain access to the source file. Caution is advised.

6.6 Zooming

Using the mouse wheel will zoom in and out on a 2D plot. On a 3D plot zooming can be achieved by keeping the right mouse button pressed and moving the mouse forward or backward. Note that solution export is only available in 2D plots.

6.7 Customized MIDACO colormap

Besides the regular Matplotlib colormaps, the PlotTool offers a customizable MIDACO colormap that is particular suited for optimization purposes. This colormap consists of five separated colors (yellow, orange, blue, green) and white. It's specific feature is that the color range gets exponentially smaller towards the edge. As the edge represents the minimum (or maximum) and is therefore of greater importance, this colormap offers a more detailed display of this area of interest. The MIDACO colormap offers two parameters: The percentage (%) along the entire colormap until which the colors are displayed and the exponential factor (ex) according to which the colors change. For example, the colormap "midaco_50%_ex2.5" will leave 50% of the entire colormap range white and change the colors according an exponential factor of 2.5. The exponential factor should normally be selected as floating number somewhere between 2.0 and 4.0. Below Figure displays four different customized midaco colormaps on the Fonesca benchmark. In below Figure the colormap has further been reversed, adding the "_r" to its name. Further note that the number of colormap marks has been increased from 10 to to 30 (see Section 6.4) for more detailed analysis.

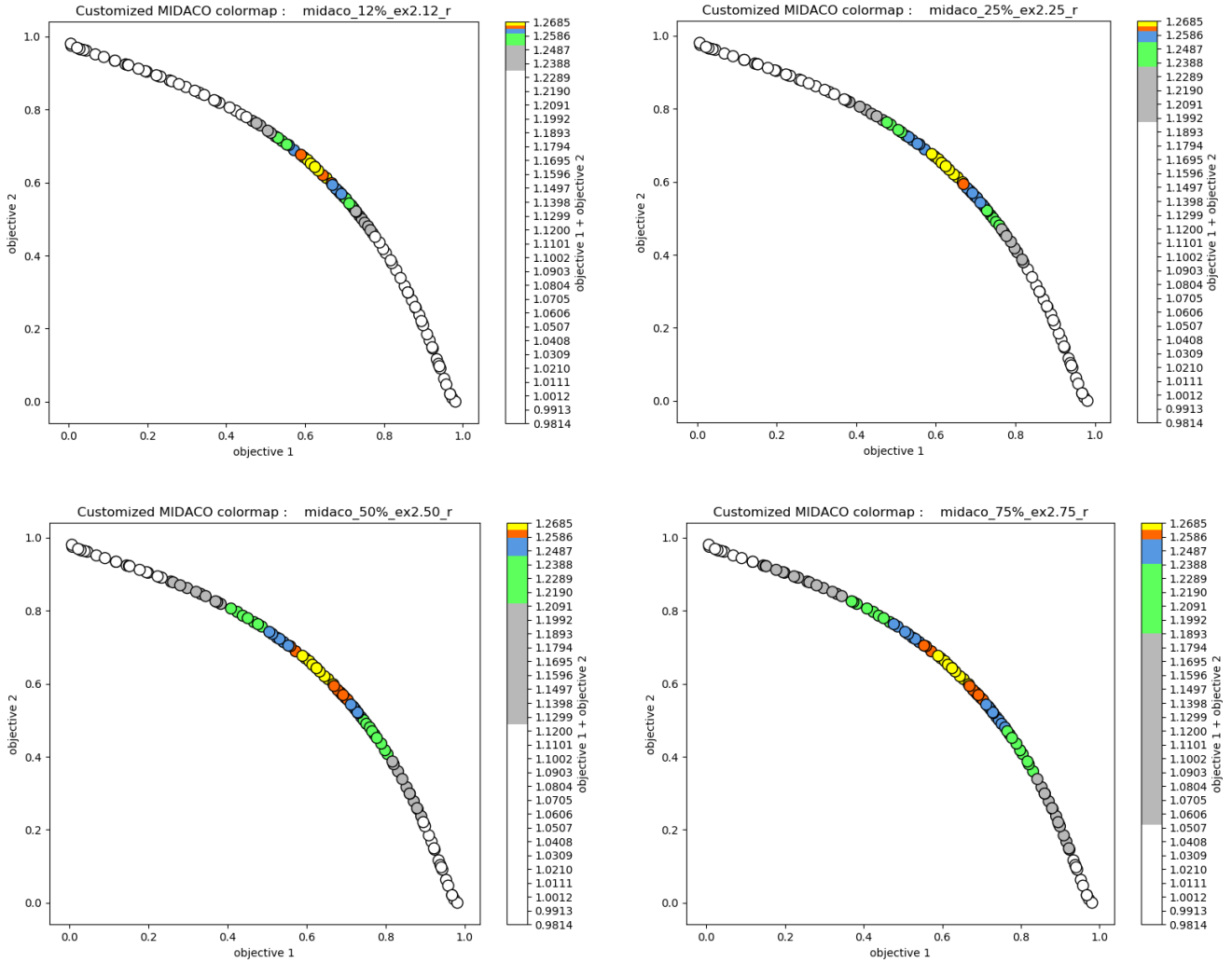


Figure 4: Four examples of customized MIDACO colormaps on the Fonesca benchmark

7 Parallelization

MIDACO offers the possibility to evaluate multiple solution candidates in parallel. In the context evolutionary algorithms such feature is also known as *co-evaluation* or *fine-grained* parallelization. Figure 5 illustrates how a *block* of **P** solution candidates (x1, x2, x3, ... xP) is submitted for parallel evaluation and the corresponding objective and constraint values ([f1,g1],...,[fP.gP]) are returned to MIDACO.

If an optimization problem is *CPU-time expensive*, that means a single evaluation of the objectives and constraints requires a significant amount of time, parallelization is highly beneficial in reducing the overall time required to solve the problem. In a recent study (Schlueter & Munetomo [44]) it was numerically demonstrated on 200 benchmark problems that MIDACO's potential speed up by parallelization exhibits a nearly linear scale-up, which means it is most effective. For a parallelization factor of $P = 10$ the potential speed up was around 10 times, while for a parallelization factor of $P = 100$ the potential speed up was still around 70 times (see Figure 4 in [44]). Note that due to the parallelization overhead such speed ups are only expected if the time to calculate the objectives and constraints is *CPU-time expensive*. Sub-section 7.2 discusses this issue in more detail.

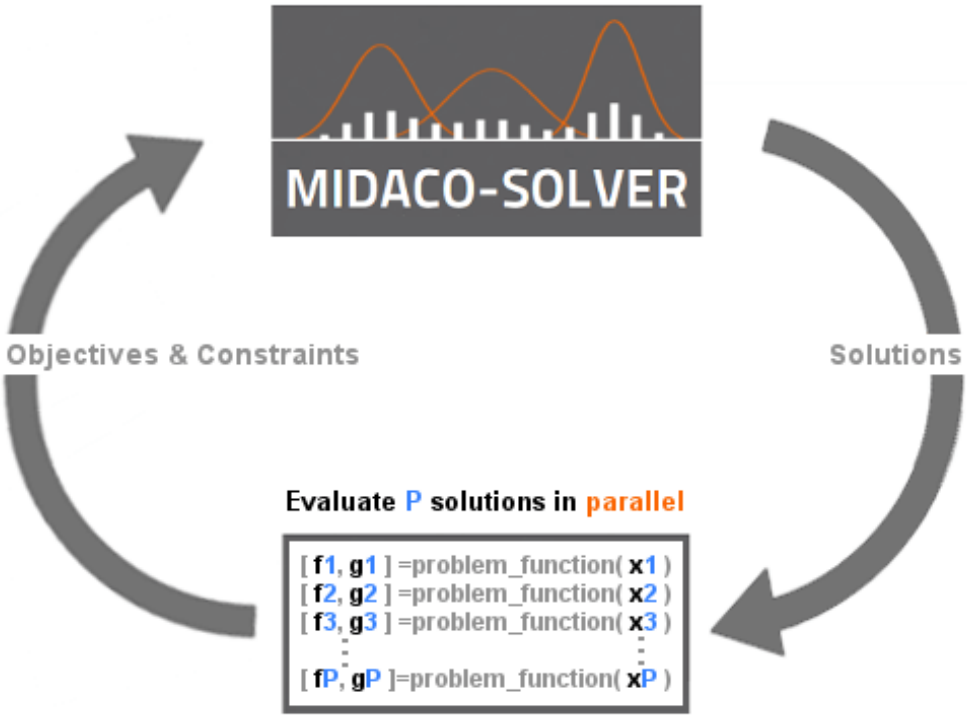


Figure 5: MIDACO evaluating a block of **P** solutions in **parallel**.

7.1 Running MIDACO in parallel

Running MIDACO in parallel mode is easy. Fully functional examples are given online for various languages (Matlab, Python, C++, R, Java, C#, Fortran) at:

http://www.midaco-solver.com/index.php/more/parallelization

Note that MIDACO's parallelization feature is not limited to those languages and approaches (like openMP or MPI) and can further be used in other languages and other parallelization approaches (like GPGPU or Hadoop/Spark). Based on MIDACO's reverse communication concept its parallelization feature can be enabled with virtually any language/approach.

7.2 Parallelization overhead

Because parallel computing will introduce some computational overhead, the question if running MIDACO in parallel mode is effective or not depends mainly on two aspects: The programming language/approach and the actual CPU-time to evaluate the objectives and constraints. As the computational overhead can be significantly large in some languages (particular in Matlab), it may happen that parallelization actually increases the overall optimization time, instead of reducing it. This is normally the case if problem function evaluation is not time intensive to compute, for example a single evaluation takes less than 0.00001 seconds.

Table 3 gives some rough guideline on the minimal evaluation cost for which parallelization is recommended, depending on various languages. As those times are subject to the actual cpu specification and also the number of available parallel threads, the actual times for a user may be both: larger or smaller. Users are advised to experiment, if parallelization in a given case is beneficial or not.

Table 3: Minimal evaluation cost for which parallelization is promising

Language	Minimial evaluation cost
Matlab	1.0 Second
Python	0.01 Second
R	0.01 Second
Java	0.001 Second
C/C++	0.001 Second
Fortran	0.001 Second

8 Tips & Tricks

This section describes some advanced hints in common optimization scenarios.

8.1 Constraint Handling

Constraint handling is an important and challenging area in optimization. The MIDACO software is capable to solve problems with up to hundreds and even thousands of constraints (e.g. see MIDACO benchmark website or Schlueter and Munetomo [44]). This includes equality as well as in-equality constraints, whereas in-equality constraints are normally easier to solve by MIDACO. Due to MIDACO's black-box concept there is no restriction on the function properties of the constraints, therefore they might be linear or non-linear and do not need to be smooth or differentiable.

In order to efficiently solve problems with many and/or difficult constraints with MIDACO, a *cascading* approach with multiple runs is recommended. The most critical parameter in solving problems with many constraints is the accuracy (PARAM(1), see Section 4.1) that measures the constraint violation. For problems with many constraints the default accuracy of 0.001 might be too difficult or time intensive to reach with a solution from scratch. Therefore the accuracy should be increased for a first run. Depending on the given application, a suitable accuracy value for a first run might be for example 0.1, 0.5 or even 1.0. With such high PARAM(1) value MIDACO will normally find feasible solutions much quicker and will also proceed faster in minimizing the objective function value, once a feasible area is found.

After above described first run with moderate constraint accuracy, the refinement of the solution accuracy can begin. Above implied feasible solution with moderate constraint accuracy can be used as starting point for further runs with a more precise accuracy, such as 0.01, 0.001 or lower. For such refinement runs it is advisable to activate the FOCUS parameter (see Section 4.6). How many refinement runs and which particular ACC and FOCUS settins are promising is subject to a given application. Table 4 gives a rough example on possible ACC and FOCUS settings for solving a difficult constrained problem in several runs. Note that in Table 4 it is assumed that the final solution satisfies a constraint violation equal or below 0.00001.

Table 4: Potential settings for multiple runs

Run	ACC	FOCUS	Starting point
1st	0.5	0.0 (default)	from scratch
2nd	0.1	-10.0	previous solution
3rd	0.01	-100.0	previous solution
4th	0.001	-1000.0	previous solution
5th	0.00001	-100000.0	previous solution

With the hypothetical multiple run setup in Table 4 it requires five runs in total to solve the constrained problem to a solution with the desired constraint accuracy of 0.00001. Note that the FOCUS parameter is used with its "-" flag, which forces MIDACO to stay with the current solution and disable complete internal restarts. This is done as in above scenarios the 2nd till 5th run are considered as refinement runs only.

8.2 Highly nonlinear problems

Similar to solving problems with many and/or difficult constraints, it may be useful to solve highly nonlinear problems (and in generally very difficult to solve problems) with a setup of several runs. In such scenario the first run act as *from scratch* run to deliver a decent first solution which is then further refined in following runs. The critical parameter for such refinement runs is FOCUS (see Section 4.6). Table 5 gives an example scenario of three runs where the 2nd and 3rd run are refinement runs.

Table 5: Potential settings for multiple runs

Run	FOCUS	Starting point
1st	0.0 (default)	from scratch
2nd	100.0	previous solution
3rd	-10000.0	previous solution

Note that in Table 5 the 2nd run assumes the FOCUS parameter without the "-" flag. This is done to enable MIDACO to still explore further areas in the 2nd run, even though it is a refinement run. In contrast to that, the 3rd run assumes FOCUS with its "-" flag, as the 3rd run is intended as the final refinement for high precision.

8.3 Large-Scale Problems

Performance on problems with hundreds and thousands of variables generally benefit from tuning the following parameter: FOCUS, ANTS, KERNEL. For the FOCUS parameter values such as 10, 50 or 100 might work in a first run from scratch. The reason for this is that in large-scale problems the FOCUS parameter effect can be stronger as in small-scale problems because a search space reduction is of greater impact in large-scale problems. For the ANTS and KERNEL parameters, settings such as [ANTS=2,KERNEL=2], [ANTS=5,KERNEL=20] or [ANTS=10,KERNEL=50] might improve the performance. Low values for the ANTS and KERNEL parameters will also reduce the search space exploration and thus lead to faster convergence.

On the contrary, above tuning examples might lead to sub-optimal convergence to a local solution. As large-scale problems are generally difficult to solve, such local solution might however be acceptable in such scenarios where the gained reduction in run-time is of greater value than the solution quality.

8.4 CPU-Time expensive applications

The performance on solving CPU-time expensive applications with MIDACO can be greatly improved by using parallelization. For applications where a single objective and constraint evaluation is expensive (for example takes more than few seconds), parallelization will on average always (drastically) reduce the total time required to solve the problem. Furthermore, the higher the level of parallelization the better. In Schlueter and Munetomo [44] it was demonstrated the for a parallelization factor of 100 the number of sequential steps required by MIDACO could be reduced by around 70 times. For a CPU-time expensive application this means that if a cluster with 100

threads is available, the MIDACO runtime can be increased around 70 times. This is for example less than a day instead of 2 month ($\approx$ 60 days).

In addition to parallelization, the same recommendation (ANTS, KERNEL, FOCUS) as given for large-scale problems (see Section 8.3) can be applied to CPU-time expensive applications. This is because in both scenarios, large-scale and CPU-time expensive applications, a reduction of the search space (and therefore of the number of function evaluation) has great impact. As mentioned in (see Section 8.3) the recommended settings might lead to faster convergence but a sub-optimal solution. For CPU-time expensive application this might be acceptable when a sub-optimal but good solution reached in reasonable time is preferred over a global solution which search effort would require unreasonable time.

8.5 Solving non-linear equation systems

Due to MIDACO's capability to handle problems with hundreds and even thousands of constraints, MIDACO can be used to solve systems of nonlinear equations (in which there are typically no particular objective functions). Instead of formulating all constraints as constraints and leaving the objective function blank (for example a constant value), it is recommended that the most difficult constraint is formulated as objective. Such way MIDACO will more easily find solutions that satisfy the majority of constraints and future refinement runs can focus on satisfying the most difficult constraint (given as objective).

8.6 Multi-modal optimization

Multi-modal optimization seeks to locate not only a single global solution, but the set of all local solutions. Because MIDACO is based on an evolutionary algorithm it explores a vast area of the search space and consequently enters a lot of local optima (in case of multi-modal problem landscapes). Enabling the creation of a history file (see Section 2.2) gives the user an easy option to keep track on all evaluated solutions and consequently to further investigate all local solutions found during the search process.

8.7 Submitting several starting points

When running MIDACO with **parallelization**, it is possible to submit several starting points. This option is useful for very cpu-time intensive application (which require e.g. hours for a single evaluation). By default, MIDACO example templates expect only a single starting point, which is then duplicated into the XXX array which stores P solution vectors X one after another. The corresponding source code in Matlab can be found in the gateway "midaco.m" around line 100:

```
103  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
104  xxx = zeros(1,P*n);
105  for c = 1:P
106      for i = 1:n
107          xxx((c-1)*n+i) = x0(i);
108      end
109  end
110  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

In case several starting points should be stored in the XXX array, those must be placed manually by the user by modifying the XXX fill up command. For example, consider P=3 and three different starting points, then the modified "midaco.m" gateway might look like following:

```
103  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
104  xxx = zeros(1,P*n);
105  % Fill up XXX array with 3 different starting points
106  for i = 1:n
107      xxx( 0*n + i ) = starting_point_1_(i);
108      xxx( 1*n + i ) = starting_point_2_(i);
109      xxx( 2*n + i ) = starting_point_3_(i);
110  end
111  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The source code commands for the XXX array fill up might look slightly different depending on the language, but is essentially always fulfilling the same purpose. Note that it is also possible (and can be useful) to submit different random solutions as starting points.

8.8 Parallel-Overclocking with MIDACO

In case of CPU-time expensive applications where the evaluation time drastically varies (for example some evaluation may take seconds while other may take minutes or hours) and parallelization with a sufficient large number of threads/cores is performed, it can be efficient to *overclock* MIDACO's parallelization factor. The term *overclocking* means here that a larger parallelization factor P is assigned than actual physical threads/cores are available. This approach is also known as **oversubscription** in high-performance computing HPC. Because the parallelization factor P can be freely chosen, it can be set to any integer value larger (or smaller) than the actual number of threads/cores available.

For example: Consider a cluster of 32 CPU's (each with one core) which act as function evaluator for some master node which runs MIDACO (such setup can for example be established with Spark). Assigning a parallelization factor of P=64 or P=128 to MIDACO will exceed the actual number of number of available cores but might lead to overall faster processing. The reason is that the pool of cores in the cluster can be used more effectively by utilizing cores which otherwise (in case of P=32) would be idle for some time after a evaluating a fast calculating solution.

Note: In some cases overclocking might also be effective on a single machine. However, overclocking is not recommended on applications where all solution evaluation require equal or quite similar CPU-time.

9 IFLAG Messages

This section describes the list of IFLAG values used by MIDACO as *information flag*. MIDACO reports various IFLAG values to indicate final solution information, warnings or input errors. In case of final solution messages an IFLAG value between 1 and 7 is stated, indicating the reason for terminating and if the solution is feasible or not. It is quite common in the process of setting up a new optimization problem that some IFLAG error messages (like IFLAG=204 → bound error) are experienced. Those are normally easy to fix and not of greater concern. The is a list of all relevant IFLAG values. Note that MIDACO uses negative IFLAG values for internal communication only.

9.1 Solution Messages (IFLAG = 1 ∼ 9)

Table 6 describes IFLAG messages which are reported along with a solution reported by MIDACO.

Table 6: 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 ALGOSTOP
4	Infeasible solution, MIDACO stopped automatically by ALGOSTOP
5	Feasible solution, MIDACO stopped automatically by EVALSTOP
6	Infeasible solution, MIDACO stopped automatically by EVALSTOP
7	Feasible solution, MIDACO stopped automatically by FSTOP

9.2 Warning Messages (IFLAG = 10 ∼ 99)

Table 7 describes IFLAG messages which are reported as warning at the beginning of the optimization process. Those warning can be ignored, but are sometimes an indicator that the problem setup is flawed.

Table 7: MIDACO warning messages indicated by IFLAG

IFLAG	
51	Some X(i) is greater/lower than +/- 10^{16} (try to avoid huge values)
52	Some XL(i) is greater/lower than +/- 10^{16} (try to avoid huge values)
53	Some XU(i) is greater/lower than +/- 10^{16} (try to avoid huge values)
71	Some XL(i) = XU(i) (fixed variable)
81	F(1) 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^{16}
92	ORACLE is greater/lower than +/- 10^{16}

9.3 Error Messages (IFLAG = 100 ∼ 999)

Table 8 and Table 9 describe IFLAG messages which are reported as error at the beginning of an optimization run. MIDACO will reject to perform any optimization if an error message is raised.

Table 8: MIDACO error messages indicated by IFLAG

IFLAG	Message Description
100	P <= 0 or P > 10^{99}
101	O <= 0 or O > 10^{9}
102	N <= 0 or N > 10^{99}
103	NI < 0
104	NI > N
105	M < 0 or M > 10^{99}
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)
	Note: The array count index of PARAM(i) is starting with one here, not zero.
301	PARAM(1) < 0 or PARAM(1) > 10^{99}
302	PARAM(2) < 0 or PARAM(2) > 10^{99}
303	PARAM(3) greater/lower than +/- 10^{99}
304	PARAM(4) < 0 or PARAM(4) > 10^{99}
305	PARAM(5) greater/lower than +/- 10^{99}
306	PARAM(6) not discrete or PARAM(6) > 10^{99}
307	PARAM(7) < 0 or PARAM(7) > 10^{99}
308	PARAM(8) < 0 or PARAM(8) > 100
309	PARAM(7) < PARAM(8)
310	PARAM(7) > 0 but PARAM(8) = 0 (ANTS and KERNEL must be used together)
311	PARAM(8) > 0 but PARAM(7) = 0 (ANTS and KERNEL must be used together)
312	PARAM(9) greater/lower than +/- 10^{99}
321	PARAM(10) >= 10^{99}
322	PARAM(10) not discrete
331	PARAM(11) < 0 or > 0.5
344	Pareto front (PF) workspace **LPF is too small**. LPF must be at least of size (O+M+N)*PARETOMAX + 1, where PARETOMAX=1000 is default (Sec 4.10)
347	PARAM(5) > 0 but PARAM(5) < 1
348	PARAM(5): Optional EVALSTOP precision appendix > 0.5
350	PARAM(12) < -1 or PARAM(12) > 1 but not a discrete value
351	PARAM(13) < 0 or PARAM(13) > 3
352	PARAM(13) not a discrete value
399	Some PARAM(i) has type NaN

Table 9: MIDACO error messages indicated by IFLAG (continued)

401	ISTOP $<$ 0 or ISTOP $>$ 1
402	Starting point does not satisfy all-different constraint
IFLAG	Message Description
501	Double precision work space size **LRW is too small**. Increase size of RW array. RW must be at least of size LRW = 120*N+20*M+20*O+20*P+P*(M+2*O)+O*O+5000
502	Internal LRW check error (please contact support)
601	Integer work space size **LIW is too small**. Increase size of IW array. IW must be at least of size LIW = 3*N+P+1000
602	Internal LIW check error (please contact support)
701	Input check failed! MIDACO must be called initially with IFLAG = 0
881	Integer part of X contains continues (non discrete) values
882	Integer part of XL contains continues (non discrete) values
883	Integer part of XU contains continues (non discrete) values
900	Invalid or corrupted LICENSE-KEY
999	N $>$ 4. The free version is limited up to 4 variables.

References

[1] Abolhassani, A., .Harner, J., Jaridi, M., Gopalakrishnan, B.: *Productivity enhancement strategies in North American automotive industry.* Int. J. Prod. Res., 8(3), pp.1–18 (2017)

[2] Alghamdi W.Y., Wu H., Zheng W., Kanhere S.S.: *Constructing A Shortest Path Overhearing Tree With Maximum Lifetime In WSNs.* Hawaii International Conference on System Sciences (HICSS-49) (2016)

[3] 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)

[4] Askin, T., Pornet, P.C., Vratny, M., Schmidt, M.: *Optimization of Commercial Aircraft Utilizing Battery based Voltaic-Joule/Brayton Propulsion.* Journal of Aircraft 54(1), pp. 246–261 (2016)

[5] Astos Solutions GmbH: *Low-Thrust Orbit Transfer Trajectory Optimization Software (LOTOS)*. Stuttgart, Germany (2016)

[6] Bahbahani M.S., Baidas M.W., Alsusa E.A.: *A Distributed Political Coalition Formation Framework for Multi-Relay Selection in Cooperative Wireless Networks.* IEEE Transactions on Wireless Communications, Volume 14 , Issue: 12, pp. 6869 - 6882 (2016)

[7] Bahbahani M.S., Alsusa E.A.: *Relay Selection for Energy Harvesting Relay Networks using a Repeated Game.* IEEE Wireless Communications and Networking Conference (WCNC), At Doha, Qatar (2016)

[8] 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)

[9] Baidas M.W. and Alsusa E.A.: *Power allocation, relay selection and energy cooperation strategies in energy harvesting cooperative wireless networks.* Wirel. Commun. Mob. Comput., DOI: 10.1002/wcm.2668 (2016)

[10] Baidas M.W. and Masud M.: *Energy-efficient partner selection in cooperative wireless networks: a matching-theoretic approach.* International Journal of Communication Systems, Volume 29, Issue 8, pp 1451-1470 (2016)

[11] Chagwiza G., Musekwa S., Jones B., Mtisi S.: *Impact of new water sources on the overall water network: an optimisation approach.* International Journal of Mathematics and Statistics Research, Vol.1, No.1, pp. 32-41 (2014)

[12] Chakraborty, D., Wang, T., Monika, A., Manfred, J., Christoph, L., Lambert, M., Schueler, W.: *Adapting Douglas-fir forestry in Central Europe: evaluation, application, and uncertainty analysis of a genetically based model.* European Journal of Forest Research, 135(5), pp. 919–936 (2016)

[13] Comas M.: *Application of sales forecasting for new products.* Technical report, Escola tecnica superior d'enginyeria industrial de Barcelona, Spain (2012)

[14] Dell I., Lekszyck T., Pawlikowski M., Grygoruk R., Greco L. : *Designing a light fabric metamaterial being highly macroscopically tough under directional extension: rst experimental evidence.* Zeitschrift fur angewandte Mathematik und Physik (ZAMP) Vol 66 (6), pp. 3473-3498 (2015)

[15] Duquenne, B.: *Optimization tool dedicated to the validation process for the automotive industry.* MSc Thesis, University of Liege, Belgium (2017)

[16] Esche E.: *MINLP optimization under uncertainty of a mini plant for the oxidative coupling of methane.* PhD-Thesis, Fakultaet III, Prozesswissenschaften, Technical University of Berlin (2015)

[17] Faramondi, L., Oliva, G., Panzieri, S., Pascucci, F., Schlueter, M., Munetomo, M., Setola, R.: *Network Structural Vulnerability: A Multiobjective Attacker Perspective.* IEEE Transactions on Systems, Man, and Cybernetics: Systems, 99, pp.–1-14 (2018)

[18] European Space Agency (ESA) and Advanced Concepts Team (ACT): GTOP database - global optimisation trajectory problems and solutions. (2016)

[19] Grujic I., Nilsson R.: Model-based development and evaluation of control for complex multi-domain systems: attitude control for a quadrotor UAV. Technical report ECE-TR-23, Aarhus University, Denmark (2016)

[20] 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)

[21] Hunter, J. D.: *Matplotlib: A 2D graphics environment.* Computing In Science & Engineering, 9(3), pp.90–95 (2007)

[22] Kahar, N.H.B.A., Zobaa, A.F. : *Optimal single tuned damped filter for mitigating harmonics using MIDACO.* IEEE Industrial and Commercial Power Systems Europe, DOI: 10.1109/EEEIC.2017.7977541, (2017)

[23] Kahar, N.H.B.A., Zobaa, A.F. : *Application of mixed integer distributed ant colony optimization to the design of undamped single-tuned passive filters based harmonics mitigation.* Swarm and Evolutionary Computation, https://doi.org/10.1016/j.swevo.2018.03.004, *in press* (2018)

[24] Lou X.: *Acceleration of Distance-to-Default with GPU.* Master-Thesis, School of Information & Communication Technology Royal Institute of Technology Stockholm, Sweden (2012)

[25] Mahajan N.R., Mysore S.P.: *Combinatorial neural inhibition for stimulus selection across space.* biorxiv, doi.org/10.1101/243279 (2018)

[26] Minguijon Pallas, P.: *Cubesat Deployment Trajectories for the Asteroid Impact Mission.* MSc Thesis, Delft University of Technology, Netherlands (2017)

[27] Mohammed, S. M.: *Exergoeconomic analysis and optimization of combined cycle power plants with complex configuration.* PhD Thesis, Univ. of Belgrade, Fac. Mech. Eng., Serbia (2015)

[28] Mukalu, M.S., Lijun, Z., Xiaohua, X.: *A Comparative Study on the Cost-effective Belt Conveyors for Bulk Material Handling.* Energy Procedia, Volume 142, pp. 2754–2760 (2017)

[29] Nie C., Wu H., Zheng W.: *Lifetime-Aware Data Collection Using A Mobile Sink in WSNs with Unreachable Regions.* MSWiM17, November 21-25, 2017, Miami, FL, USA 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems Pages 143-152, DOI:10.1145/3127540.3127544 (2017)

[30] Perez R.E., Jansen P.W.: *Effect of Passenger Preferences on the Integrated Design and Optimization of Aircraft Families and Air Transport Network.* 17th AIAA Aviation Technology, Integration, and Operations Conference, DOI: 10.2514/6.2016-3748 (2017)

[31] Redutskiy Y.: *Oilfield development and operations planning under geophysical uncertainty.* Engineering Management in Production and Services, Volume 9, Issue 3, Pages 10-27, doi.org/10.1515/emj-2017-0022 (2018)

[32] 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)

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

[34] 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)

[35] 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)

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

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

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

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

[40] Schlueter M. and Munetomo M.: *Parallelization Strategies for Evolutionary Algorithms for MINLP*, Proc. Congress on Evolutionary Computation (IEEE-CEC), pp.635-641 (2013)

[41] Schlueter M. and Munetomo M.: *Parallelization for Space Trajectory Optimization*, Proc. World Congress on Computational Intelligence (IEEE-WCCI), pp. 832 - 839 (2014)

[42] Schlueter M.: *MIDACO Software Performance on Interplanetary Trajectory Benchmarks*, Advances in Space Research (Elsevier), Vol 54, Issue 4, Pages 744 - 754 (2014)

[43] Schlueter M., Yam C.H., Watanabe T., Oyama A.: *Parallelization Impact on Many-Objective Optimization for Space Trajectory Design*, *Int. J. of Machine Learning and Computing 6.1: 9-14.* (2016)

[44] Schlueter M. and Munetomo M.: *Numerical Assessment of the Parallelization Scalability on 200 MINLP Benchmarks*, *Proc. IEEE-WCCI, Vancouver, Canada* (2016)

[45] 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)

[46] Teichgraeber, H., Brodrick, P., Brandt, A.: *Optimal design and operations of a flexible oxyfuel natural gas plant.* Energy 141, DOI: 10.1016/j.energy.2017.09.087 (2017)

[47] Tilly, J., Niedermayer, K.: *Employment and Welfare Effects of Short-Time Work.* German Economic Association, Annual Conference: Demographic Change, Augsburg (2016)

[48] Ukritchon B., Boonyatee T.: *Soil Parameter Optimization of the NGI-ADP Constitutive Model for Bangkok Soft Clay.* Geo. Eng. J. SEAGS & AGSSEA, Vol. 46(1), pp. 28-36 (2015)

[49] Wang K., Mao Y., Chen J., Yu S.: *The optimal research and development portfolio of low-carbon energy technologies: A study of China.* J. Clean. Prod., Vol 176, pp. 1065–1077 (2018)

[50] Weiss L., Koeke H., Schlueter M., Huehne C.: *Structural optimisation of a composite aircraft frame for a characteristic response curve*. Proc. Euro. Conf. Comp. Mat. (ECCM17), (2016)

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

[52] Zhao Q., Neveux T., Jaubert J.N., Mecheri M., Privat R.: *Design of SC-CO2 Brayton cycles using MINLP optimization within a commercial simulator.* 6th Inter. Supercritical CO2 Power Cycles Symposium, March 27-29, 2018, Pittsburgh, USA (2018)

[53] Zarko D., Kovacic M., Stipetic S., Vuljaj D.: *Optimization of electric drives for traction applications.* 19th Int. Conf. on Elec. Drives and Power Electronics (EDPE), Dubrovnik, pp. 15–32 (2017)