# MAPLE Toolbox GANSO
# Global And Non-Smooth Optimization
# Version 1.0
# User Manual

Centre for Informatics and Applied Optimisation
School of Information Technology and Mathematical Sciences
University of Ballarat
Ballarat, Victoria, Australia
www.ballarat.edu.au/ciao

# Contents

# Chapter 1

# Introduction

This manual describes MAPLE toolbox `GANSO`, which stands for `Global And Non-Smooth Optimization`. `GANSO` implements several methods of optimization:

- Derivative-Free Bundle Method (DFBM) of nonsmooth optimization;
- Extended Cutting Angle Method (ECAM) of global Lipschitz optimization;
- Dynamical System – based Optimization (DSO) - a method of global optimization;
- Random Start local optimization;
- Various combinations of the above methods.

A detailed description of these methods is given in a series of papers listed in the bibliography section. This manual is describes Maple user interface, and is a companion of `GANSO` User Manual.

Chapter 2 describes the installation instructions. Chapter 3 gives a brief description of optimization methods used in this library. The description of the programming library `GANSO` is given in Chapter 4. Examples of its usage are provided in Chapter 5.

Note that this distribution applies to Maple, version 9 and 10 running under MS Windows only.

## 1.1 Quick start

Please refer to the detailed description of all the procedures in Chapter 4 and examples in Chapter 5. Suppose you want to find a minimum of $f(x) = x_1^2 + x_2^2 - \cos(18x_1) - \cos(18x_2)$ in the box $[-1, 1]^2$.

1. Copy the *dll* files into Maple's *bin* directory or any other directory on the path. These files can be downloaded using
`http://www.ganso.com.au/libs/gansomapledlls.zip`

2. In your Maple worksheet use the commands

```
 GANSOPATH:="c:/yourdirectory/mwrap_ganso.dll":
 Minecam0 :=define_external('MWRAP_MinimizeECAM_0','MAPLE', LIB=GANSOPATH):
 Mindfbm0 :=define_external('MWRAP_MinimizeDFBM_0','MAPLE', LIB=GANSOPATH):

f := proc( n, x ) #user's objective function
        local s;
        s:= evalf(x[1]^2+x[2]^2-cos(18*x[1])-cos(18*x[2]));
        s  #this last value is what is returned
    end proc;

# defining vectors; large values mean no box constraints
    dimension:=2;
    Lo:=Vector(1..2,datatype=float[8],[-1,-1]):
    Up:=Vector(1..2,datatype=float[8],[1,1]):
    iter:=1000: val:=1.0: # this is needed
    LipConst:=10.0:
    x0:=Vector(1..2,datatype=float[8],[1,1]);
# executing
    Minecam0(dimension,x0,val,f,LipConst,Lo,Up,iter);
    print(x0); #the solution
# try local search from a staring vector(0.5,0.5)
    x0:=Vector(1..2,datatype=float[8],[0.5,0.5]);
    Mindfbm0(dimension,x0,val,f,Lo,Up,iter);
    print(x0); #the solution
```

Notice that the output variable *val* was given an arbitrary float value before calling `GANSO` subroutines. This is to pass to `GANSO` the variable of the right type to obtain the value of the minimum. The same value is returned by Minecam0 and Mindfbm0. Other subroutines and handling constraints are discussed below.

## 1.2 Numerical efficiency

In the definition of the objective function in the example above one can use most Maple commands (such as sums, products, various functions), and conveniently work with the components of the vector $x$. For example one can use the following command to compute the Euclidean norm of $x$

```
f := proc( n, x ) #user's objective function
        local s;
        s:=sqrt( add(x[i]^2, i=1..n) );
    end proc;
```

Of course one has to ensure that the returned value is a floating point number (typically by using `evalf()` function). It is worth checking that the function returns correct values before passing it as an argument to `GANSO` subroutines, as these would call this function many times. For example, use something like `f(2,[1,1])` to test the returned value.

However the price for this convenience is the slowness of evaluation and also memory requirements, as Maple seem to take large chunks of memory during multiple calls to $f$. Therefore we do not recommend using this syntaxis for all but very simple low dimensional problems.

Maple also provides an alternative way of calling user's objective functions from external packages such as `GANSO` , which uses hardware floating point calculations. `GANSO` Toolbox provides an alternative way to call all `GANSO` subroutines (you recognize them by the $HF$ (stands for *hardware float*) after the underscore in subroutine's name), which is much more efficient numerically. The drawback is that one has to declare explicitly all the variables (i.e., one cannot use vector notation) and cannot include more complicated Maple commands (such as `add()` mentioned above). This syntaxis is illustrated below.

```
 GANSOPATH:="c:/yourdirectory/mwrap_ganso.dll":
 MindfbmOHF :=define_external('MWRAP_HFMinimizeDFBM_0','MAPLE', LIB=GANSOPATH):

f := proc( x1,x2 ) #user's objective function
        local s;
        s:= x1^2+x2^2-cos(18*x1)-cos(18*x2));
    end proc;
```

For five variables the function header would be

```
f := proc( x1,x2,x3,x4,x5 )
```

The rest of the Maple worksheet is unchanged (except that now the call is to `MindfbmOHF()`).

```
# defining vectors; large values mean no box constraints
    dimension:=2;
    Lo:=Vector(1..2,datatype=float[8],[-1,-1]):
    Up:=Vector(1..2,datatype=float[8],[1,1]):
    iter:=1000: val:=1.0: # this is needed
    x0:=Vector(1..2,datatype=float[8],[1,1]);

# try local search from a staring vector(0.5,0.5)
    x0:=Vector(1..2,datatype=float[8],[0.5,0.5]);
    MindfbmOHF(dimension,x0,val,f,Lo,Up,iter);
    print(x0); #the solution
```

Hence `GANSO` toolbox provides two alternative ways of defining user's objective functions: a less efficient but user friendly method, suitable for testing and also for simple low dimensional problems, and a much faster, but less user friendly method, suitable for more complex problems. The gain in efficiency is several orders of magnitude.

# Chapter 2

# Installation instructions and Licensing

## 2.1 Library contents

`GANSO` is distributed under the license from CIAO, described in the sequel, which allows the users unlimited use of the library in their software, as long as all other conditions of the License agreement are met.

The library is distributed in compiled form (as binary library files), example programs in Maple and this user manual. Installation involves copying the contents of the library into a designated directory.

Before using the software and its technical documentation, please take a few moments to read the following legal and contact information.

## 2.2 License agreement

The `GANSO` software product and its entire documentation are developed and maintained by the Centre for Informatics and Applied Optimisation, School of Information Technology and Mathematical Sciences, University of Ballarat, Australia. The developer will be abbreviated below as CIAO.

The `GANSO` library – including all files distributed as part of the product delivery – and its documentation may not be changed or modified in any way, except by CIAO, or following the written permission of CIAO. All proposed changes should be requested by contacting CIAO.

Without a proper license issued to users individually by CIAO, no part of the GANSO documentation and/or of the software may be stored, reproduced, transmitted, transferred, or used in any form or by any means, by any information storage and retrieval system or process.

GANSO is provided to registered users in compiled (static or dynamic link library) form. It is specifically prohibited to apply reverse engineering or any other form of internal program structure analysis to the GANSO software product.

Registered GANSO users are granted permission to use all information summarized by the User Manual, and to apply all GANSO software components and test examples in their own work, without any restriction. A reference to the software and its documentation will be appreciated in published work in which GANSO is used.

If GANSO is used as part of a new software application, then it is requested to maintain all GANSO copyright and licensee information in that application, including all reports generated by the GANSO software.

Regardless of how a copy of GANSO is obtained, it is requested that all users comply with the licensing and registration provisions if they continue to use the software.

**Limited Warranty and Disclaimer**

CIAO guarantees that all shipped GANSO products are free from defects in materials and workmanship, under normal use and service for a period of 90 days.

CIAO reserves the right to revise the GANSO software and its documentation, with no obligations to notify any person or organization of such revision. Information updates will be provided upon request.

The GANSO library is distributed 'as is'. CIAO and library developers specifically disclaim all warranties – express or implied – related to the merchantability and fitness of the GANSO software for a particular purpose or application use.

In no event shall CIAO and its developers be liable for loss of profit, commercial, business related, or any other damage-including, but not limited to special, incidental, consequential, or any other explicit and implied damages–as a consequence of using, or inability to use, the GANSO library.

**Technical Support**

Registered users of GANSO are entitled to limited technical support, provided by CIAO. Please note that e-mail is the preferred way of communication, except in cases of extreme urgency. Consulting fees may be charged for direct assistance via telephone/fax.

Present and prospective `GANSO` users are encouraged to contact CIAO, should they wish to discuss specific applicability issues, and possibilities of research and/or commercial co-operation. All constructive suggestions and comments that could lead to improvements of `GANSO` , its documentation, and its applicability are welcome and appreciated. We are much interested to hear user comments and suggestions; test models and application examples are also welcome.

Thank you for your interest in our software.

## 2.3 Installation: Windows

`GANSO` Maple toolbox is distributed in the form of dynamically linked libraries (DLL), together with the corresponding Maple file and the examples of usage. `GANSO` Maple toolbox requires `GANSO` libraries, which should be downloaded from `http://www.ganso.com.au/gansomapledlls.zip`. The installation involves extracting the contents of this file into a designated directory using a utility program such as `WinZip`. The files from that library are `gansodll.dll`,`mwrap_ganso.dll`, they should be placed into any directory on the $PATH$ or into Maple's *bin* directory.

Installation steps:

1. Download the file `http://www.ganso.com.au/libs/gansomapledlls.zip` and extract its contents into Maple's *bin* directory (or any directory on the $PATH$).

2. Open the `mapletest.mw` worksheet, which comes with this manual, to test the installation and follow the examples.

# Chapter 3

# Optimization methods

GANSO library implements a number of methods of Global and Non-Smooth Optimization methods developed by CIAO and its associates. These methods aim at solving the following generic optimization problem

$$\begin{aligned} &\text{minimize } f(x) \\ &\text{subject to } x \in D \subset R^n. \end{aligned} \tag{3.1}$$

The feasible domain $D$ is specified by a number of linear constraints (equations and inequalities), including box constraints

$$a_i \leq x_i \leq b_i \; i = 1, \ldots, n.$$

In the case of unconstrained minimization, $D = R^n$.

The class of objective functions $f$, dealt with in GANSO , is very broad. We do not assume differentiability of $f$, and only require its Lipschitz continuity (local or global). The Lipschitz continuity is expressed as

$$|f(x) - f(y)| \leq Ld(x, y),$$

where $x, y$ are two points in $D$, $d(x, y)$ is the distance between these points (e.g., Euclidean distance) and $L$ is some positive number. The inequality should hold for all $x, y \in D$.

Under such a general condition, the optimization problem is extremely difficult. The objective function may have many local extrema, and locating its absolute minimum (global minimum) is very challenging. Computation of the direction of descent at those points where $f$ is not differentiable is also tricky. Familiar methods, such as quasi-Newton, or conjugate gradient, will simply not work on this type of problems.

There is substantial literature on the subject of global and non-smooth optimization. The references section lists a few popular books and overviews, e.g., [BGLS00, HT93, HPT00, HP95, MW93, Pin96, SS00]. The user is encouraged to familiarize him/herself with some of these books, to appreciate the difficulty of the problem and the unavoidable limitations of any generic method for its numerical solution.

`GANSO` implements four different approaches to global non-smooth optimization, and also their combinations. For unconstrained local non-smooth optimization we use the Derivative Free Bundle Method (DFBM), based on finite difference approximation to the subgradient [Bag03, Bag02]. For global optimization (necessarily on some compact domain $D$) we use simple Random start method, the Extended Cutting Angle method (ECAM) [Rub00, BR01, BB02, Bel04, Bel05], and a method based on trajectories of Dynamical Systems (DSO). Some combinations of these methods are also implemented. The idea is that methods that use different approaches may enhance each other in some way.

It should be noted that none of the methods alone (or in combination) cannot guarantee the best optimal solution in practical setting (global convergence can be proved in theory, but it requires astronomical computing time). Therefore the user should experiment with different methods in order to choose the one most suitable for her particular problem.

## 3.1 Non-Smooth Local Optimization

There are many practically relevant mathematical models that involve non-smooth functions, i.e. continuous functions that have a discontinuous gradient. Within the broad class of non-smooth functions, the set of locally Lipschitz-continuous functions, and in particular the class of convex functions, is of special interest. The notion of subdifferential (see [Roc70]) is a generalization of the gradient for non-smooth convex functions. Different approaches to the generalization of this notion have been proposed subsequently: the Clarke subdifferential (see [Cla83]) and the quasidifferential (see [DR86, DR95]) are the most important among these from the numerical point of view.

In the optimization methods based on descent, an essential step is estimating the direction of descent using some information about the subdifferential. In the DFBM method, implemented in `GANSO` , we use a special finite difference approximation to the subdifferential, called the discrete gradient [Bag02].

The DFBM method is derivative-free, which means that it only uses the values of the objective function, but not its derivative (or its generalization) in explicit form. In essence the implemented algorithm iterates between two steps: calculation of the descent direction from the approximation to the subdifferential, and a line search along this direction.

While the DFBM is a local method (i.e., it converges to a locally optimal solution, from any starting point $x$), the fact that it uses an approximation to the subdifferential, allows it to converge to a sufficiently "deep" local minimum in multiextremal problems, i.e., "skip" through many annoying shallow local minima. This is an advantage of this method over other competing approaches that converge to the nearest local minimum.

## 3.2 Global Optimization

In many practical problems the objective function $f(x)$ possesses many (sometimes myriads of) local minima. The goal is to locate the global minimum (see Fig.3.1).

### 3.2.1 Random start

This is a very simple approach which involves using any local optimization method from a large number of "starting" points. The starting points are chosen in a random way, so that they cover the whole feasible domain. The smallest local minimum is selected as a substitute for the global minimum. There is no guarantee on the quality of the solution, but in many applications this approach delivers good results.

In `GANSO` we use the Sobol quasirandom sequence of starting points. We apply the DFBM from each of these starting points. The algorithm returns the best local minimizer found in this way.

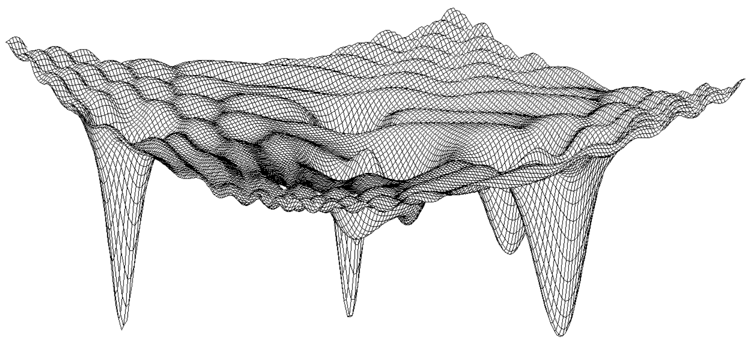

Figure 3.1: Graph of a typical multiextremal function, with a large number of shallow local minima.

### 3.2.2 Extended Cutting Angle Method (ECAM)

Under Lipschitz continuity assumption, it is possible to estimate the smallest possible minimum of the objective function, from its recorded values at various points. It follows from the Lipschitz condition that

$$f(x) \geq \max_{k=1,\dots,K} f(x^k) - Ld(x^k, x) =: H(x), \tag{3.2}$$

where $x^k$ are the points with the recorded values of $f(x^k)$, and $L$ is the Lipschitz constant of $f$. The expression on the right is called the (saw-tooth) underestimate of $f$. By using a large number of points $x^k$, it is possible to approximate $f$ closely by its underestimate $H$, and then use the global minimum of the underestimate to approximate that of $f$.

It turns out that minimizing the underestimate $H$ is a structured optimization problem, and all its local (and hence global) minimizers can be found explicitly. This is the basis of the ECAM [Rub00, BB02, Bel04, Bel05]. It uses a computationally efficient representation of local minimizers of $H$ in a tree data structure, and computes the global minimum of $f$ from this information. The method guarantees the globally optimal solution.

It should be noted that this method requires a very large number of function values even in problems with moderate dimension ($n \leq 5$). The issue here is not the computational algorithm, but the very fact that the points in $n$-dimensional space are very sparse. To build an accurate underestimate of $f$, the points should cover the feasible domain densely. What ECAM does however, is choosing the points $x^k$ adaptively, only in the "promising" regions, and thus improving the accuracy in the regions near deep local minimizers. ECAM employs the technique of fathoming, similar to branch-and-bound algorithms.

At the end of its execution, ECAM algorithm calls local search (DFBM) a specified number of times to improve the solution. DFBM uses the starting points supplied by ECAM.

### 3.2.3 Dynamical Systems - Based Optimization

The idea of this method (DSO) is to build a dynamical system using a number of values of the objective function, and associating with these data certain "forces". The evolution of such a system yields a globalized descent trajectory, which leads to lower values of the objective function. The DSO method typically starts with a box domain, samples the objective function within this search domain, and chooses a number of these values to define the system evolution rules. The algorithm continues sampling the domain until it converges to a stationary point [Mam04, MO05, MRY05].

## 3.3 Combination of methods

**DFBM + ECAM**

This is a combination of the DFBM with a version of ECAM, designed to improve line search used in DFBM, as well as to facilitate leaving shallow local minima. If at some stage DFBM algorithm does not find a direction of descent, then it chooses the specified number of promising ascent directions, and spans a linear subspace of $R^n$, in which global search is performed. Using the boundaries of the feasible domain, and its estimate of the Lipschitz constant of $f$, DFBM algorithm sets a search domain for ECAM in this subspace. ECAM performs a specified number of iterations, and if successful, finds a point with a smaller value of the objective function.

The dimension of the subspace is typically smaller than $n$, and therefore there is no guarantee that global search will escape the current local minimum. There is a trade-off between the effectiveness of this method and its numerical efficiency (as global search is numerically expensive).

### ECAM + DFBM

This is a combination of ECAM with local search. We use the global algorithm ECAM at the outer level, and the DFBM at the inner level. That is, at every iteration of ECAM, instead of calculating the value of the objective function $f(x)$, the algorithm executes the DFBM local optimization routine, thus taking a local minimum to which the DFBM converges taking $x$ as the starting point. From the point of view of DFBM, ECAM serves as a driver to generate appropriate starting points for local optimization.

While from the point of view of ECAM the objective function is no longer continuous, the underestimate $H$ in (3.2) will remain an underestimate, and the algorithm will converge to the global minimum of $f$.

### ECAM + DSO

In this combination ECAM performs global search and builds a crude model of the objective function. It then calls DSO a specified number of times, and provides it with box domains, chosen according to the model. DSO performs its search within the supplied box domains.

### Iterative DSO

This combination consists of using DSO method in a number of stages, each time reducing the search domain by some factor, using the optimal solution found in a previous stage. The natural search domain for DSO method is a box, which initially comprises the whole feasible domain. Once the first stage of DSO has converged to some solution, a smaller box around this point is chosen as the search domain for the next stage. This way DSO algorithm tries to improve its results by concentrating search in a smaller neighbourhood of the current optimal solution.

An approximation to a globally optimal solution provided by a global method is usually improved by a local search (DFBM) at the final stage.

## 3.4 Constraints

`GANSO` allows one to specify linear equality and inequality constraints on the feasible domain $D$, as

$$D = \{x \in R^n : Ax = b, Cx \leq d\},$$

where $A$, $C$ are matrices with $n$ columns and $m_A$ and $m_C$ rows, and $b$ and $d$ are vectors with $m_A$ and $m_C$ elements respectively. It is assumed that the rows of matrices $A$ and $C$ are linearly independent.

`GANSO` performs preprocessing of the constraints. The inequality constraints (except box constraints) are transformed into equality constraints with the help of $m_C$ artificial non-negative variables (slack variables). Then the feasible domain is expressed as

$$D = \{y \in R^{n+m_C} : \tilde{A}y = \tilde{b}\}.$$

Following, $n + m_C - m_A$ independent variables are identified (basic variables $y_B$). The rest of the variables are expressed as an affine combination of the basic variables $y_{NB} = Gy_B + g$, where the matrix $G$ and vector $g$ are obtained from the original parameters $A, b, C, d$ using linear algebra operations. The choice of the basic variables is governed by numerical stability of the matrix inversion operation when computing $G$ and $g$ (we use LU factorization with pivoting).

Box constraints on the basic variables are dealt with by the algorithms directly (they are natural global optimization, the DFBM adapts its approximation to the subdifferential), while box constraints on the non-basic variables are dealt with using an exact penalty function, calculated internally).

All these procedures are transparent to the user, who only needs to provide the arrays containing the elements of $A, b, C, d$ to the algorithm.

Nonlinear equality and inequality constraints are generally dealt with using exact penalty functions. The penalty parameters depend very much on the type of the constraints, and should be implemented by the user. It involves modifying the value of the objective function by using an additive penalty term, such as

$$objf = f(x) + P||c(x)_+||,$$

where $P$ is the penalty parameter, $c(x)$ denotes the vector of constraint violations, and $c_+$ denotes its positive part (i.e., its $i$-th component is not zero only when the $i$-th constraint is violated. The norm is arbitrary, see [BGLS00].

The nonlinear constraints should be implemented by the user, and supplied to the algorithms through the values of the modified objective function. The `GANSO` algorithms will treat the nonlinearly constrained and unconstrained problems equally, and it is the users responsibility to choose the right penalty parameters and to check the feasibility of the solution returned by `GANSO` .

# Chapter 4

# Maple interface

## 4.1 Interface

The described methods of optimization are implemented in the programming library `GANSO` in `C++` language. The algorithms can be accessed via Maple wrapper library `mwrap_ganso.dll`, which passes the parameters to `gansodll.dll` and executes the required subroutines.

The user has to perform the following steps.

- Declare `GANSO` subroutines which will be called by Maple using `define_external` Maple command (see examples below).
- Define the objective function, as Maple procedure, which takes two arguments (the size of vector $x$, and $x$ itself) and returns a real value.
- Define the vectors of bound constraints and the vector to hold the solution (with the initial approximation for DFBM).
- Define matrices of linear equality and inequality constraints if required.
- Execute `GANSO` subroutines.

Please note that Maple executes the commands much slower than C or Fortran, hence the bottleneck of computations will be executing the user's objective function. We mentioned in Chapter 1, that there are two alternative ways of calling the user's function, a slow but user friendly method, which allows to use vector notation, loops and most Maple commands, and a less user friendly but much more efficient method, which uses hardware floats.

It is advisable not to use the first method for computations involving a significant number of function evaluations (for complicated functions in more than 4-5 variables), and to use the second method, or `GANSO` C++ library instead.

Below is an example of Maple worksheet (First method)

```
Mindfbm0 := define_external('MWRAP_MinimizeDFBM_0','MAPLE',
    LIB="c:/work/maple/mwrap_ganso.dll"):

ff := proc( n, xx ) #user's objective function
        local s;
        s:= evalf(xx[1]^2+(xx[2]-1)^2)+1.;
        s  #this last value is what is returned
    end proc;

# defining vectors; large values mean no box constraints
    dimension:=3;
    Lo:=Vector(1..3,datatype=float[8],[-10e30,-10e30,-1e30]):
    Up:=Vector(1..3,datatype=float[8],[2e30,2e30,2e30]):
    iter:=100: val:=1.0:
    x0:=Vector(1..3,datatype=float[8],[1,1,1]);

# executing
    Mindfbm0(dimension,x0,val,ff,Lo,Up,iter);
    print(x0);
```

Please note that vectors must be declared using

```
Vector(1..dimension,datatype=float[8]);
```

declaration. This ensures that Maple passes correct data type to C-language wrapper library. Also note that even though *val* is an output parameter, it is assigned a (real) value before calling `Mindfbm0`. This is to ensure we pass the right data type to the subroutine to receive the computed value.

Here is the same example (Second method using hardware floats).

```
MindfbmOHF := define_external('MWRAP_HFMinimizeDFBM_0','MAPLE',
    LIB="c:/work/maple/mwrap_ganso.dll"):

ff := proc( xx1,xx2,xx3 ) #user's objective function
        local s;
        s:= xx1^2+(xx2-1)^2+1.;
    end proc;

# defining vectors; large values mean no box constraints
    dimension:=3;
    Lo:=Vector(1..3,datatype=float[8],[-10e30,-10e30,-1e30]):
    Up:=Vector(1..3,datatype=float[8],[2e30,2e30,2e30]):
    iter:=100: val:=1.0:
    x0:=Vector(1..3,datatype=float[8],[1,1,1]);

# executing
    MindfbmOHF(dimension,x0,val,ff,Lo,Up,iter);
    print(x0);
```

The difference in syntaxis is:
a) Using $HF$ in `define_external` in the name of `GANSO` subroutine.
b) Declaring all (scalar) arguments explicitly in `ff()`.

The following commands should be used to declare GANSO subroutines (when using slow calling method). Note that the path to mwrap_ganso.dll will be specific to the user's PC. Also note that gansodll.dll should be installed in a subdirectory where Maple can find it (e.g., specified with *PATH* environment variable, or in Maple's *bin* directory).

```
PathGanso:="c:/work/maple/mwrap_ganso.dll";
Mindfbm0 :=define_external('MWRAP_MinimizeDFBM_0','MAPLE', LIB=PathGanso):
Mindfbm:=define_external('MWRAP_MinimizeDFBM','MAPLE',LIB=PathGanso):
Minecam0 := define_external('MWRAP_MinimizeECAM_0','MAPLE',LIB=PathGanso):
Minecam := define_external('MWRAP_MinimizeECAM','MAPLE',LIB=PathGanso):
Minecamdfbm0 := define_external('MWRAP_MinimizeECAMDFBM_0','MAPLE',
                           LIB=PathGanso):
Minecamdfbm := define_external('MWRAP_MinimizeECAMDFBM','MAPLE',
                           LIB=PathGanso):
Mindfbmecam0 := define_external('MWRAP_MinimizeDFBMECAM_0','MAPLE',
                           LIB=PathGanso):
Mindfbmecam := define_external('MWRAP_MinimizeDFBMECAM','MAPLE',LIB=PathGanso):
Mindso0 := define_external('MWRAP_MinimizeDSO_0','MAPLE',LIB=PathGanso):
Mindso := define_external('MWRAP_MinimizeDSO','MAPLE',LIB=PathGanso):
Minecamdso0 := define_external('MWRAP_MinimizeECAMDSO_0','MAPLE',
                           LIB=PathGanso):
Minecamdso := define_external('MWRAP_MinimizeECAMDSO','MAPLE',LIB=PathGanso):
Miniterativedso0:=define_external('MWRAP_MinimizeIterativeDSO_0','MAPLE',
                           LIB=PathGanso):
Miniterativedso:=define_external('MWRAP_MinimizeIterativeDSO','MAPLE',
                           LIB=PathGanso):
Minrandomstart0:=define_external('MWRAP_MinimizeRandomStart_0','MAPLE',
                           LIB=PathGanso):
Minrandomstart:=define_external('MWRAP_MinimizeRandomStart','MAPLE',
                           LIB=PathGanso):
```

These commands are provided in the example worksheet. Of course, the user does not need to declare all these subroutines, only the ones that she intends to use.

This is the list of alternative subroutines, which use a faster execution method. They differ by the letters $HF$ after the underscore in the subroutine's name.

```
PathGanso:="c:/work/maple/mwrap_ganso.dll";
Mindfbm0HF :=define_external('MWRAP_HFMinimizeDFBM_0','MAPLE', LIB=PathGanso):
MindfbmHF:=define_external('MWRAP_HFMinimizeDFBM','MAPLE',LIB=PathGanso):
Minecam0HF := define_external('MWRAP_HFMinimizeECAM_0','MAPLE',LIB=PathGanso):
MinecamHF := define_external('MWRAP_HFMinimizeECAM','MAPLE',LIB=PathGanso):
Minecamdfbm0HF := define_external('MWRAP_HFMinimizeECAMDFBM_0','MAPLE',
                           LIB=PathGanso):
MinecamdfbmHF := define_external('MWRAP_HFMinimizeECAMDFBM','MAPLE',
                           LIB=PathGanso):
Mindfbmecam0HF := define_external('MWRAP_HFMinimizeDFBMECAM_0','MAPLE',
                           LIB=PathGanso):
MindfbmecamHF := define_external('MWRAP_HFMinimizeDFBMECAM','MAPLE',LIB=PathGanso):
Mindso0HF := define_external('MWRAP_HFMinimizeDSO_0','MAPLE',LIB=PathGanso):
MindsoHF := define_external('MWRAP_HFMinimizeDSO','MAPLE',LIB=PathGanso):
Minecamdso0HF := define_external('MWRAP_HFMinimizeECAMDSO_0','MAPLE',
                           LIB=PathGanso):
MinecamdsoHF := define_external('MWRAP_HFMinimizeECAMDSO','MAPLE',LIB=PathGanso):
Miniterativedso0HF:=define_external('MWRAP_HFMinimizeIterativeDSO_0','MAPLE',
                           LIB=PathGanso):
MiniterativedsoHF:=define_external('MWRAP_HFMinimizeIterativeDSO','MAPLE',
                           LIB=PathGanso):
Minrandomstart0HF:=define_external('MWRAP_HFMinimizeRandomStart_0','MAPLE',
                           LIB=PathGanso):
MinrandomstartHF:=define_external('MWRAP_HFMinimizeRandomStart','MAPLE',
                           LIB=PathGanso):
```

Next we describe the usage of each of the listed subroutines. Since the syntaxis of both types of subroutines (those that do and no not use hardware floats) is exactly the same, we only describe the first type.

We start with the subroutines which have "0" at the end, meaning that they do not use constraints (except box constraints).

`float Mindfbm0(dim, x0, val, f, Xlo, Xup, maxiter)`

Performs local nonsmooth optimization using DFBM. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),
$float\ x0[dim]$ - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
*float val* - is the value of the objective function at the final point, the output parameter,

*f* - is the reference to the user supplied function,
*float Xlo*[*dim*] - is the lower bound on $x$,
*float Xup*[*dim*] - is the lower bound on $x$,
*integer maxiter* - is the number of iterations. If 0, then the default value of 10000 will be used.
This function returns the value of the minimum (same as *val*).

Notes: 1. Memory should be allocated to vectors $x0, Xlo, Xup$ in the Maple worksheet as
`x0:=Vector(1..3,datatype=float[8],[-1,-1,-1]):`
and the values should be initialized.
2. To ensure correct type of the output variable *val* assign to it a real value (like $val := 1.0$ :) before calling `Mindfbm0`.
3. For unconstrained problems set all components of $Xlo$ to $-10^{20}$ or smaller and components of $Xup$ to $10^{20}$ or larger.

`float Minecam0(dim, x0, val, f, LipConst, Xlo, Xup, maxiter, iterlocal)`

Performs global optimization using ECAM. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),
*float x*0[*dim*] - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
*float val* - is the value of the objective function at the final point, the output parameter,
*f* - is the reference to the user supplied function,
*float LipConst* - an estimate of the Lipschitz constant of $f$,
*float Xlo*[*dim*] - is the lower bound on $x$,
*float Xup*[*dim*] - is the lower bound on $x$,
*integer maxiter* - is the number of iterations. If 0, then the default value of 1000 will be used,
*integer iterlocal* - is the number of runs of local descent method DFBM to improve results of global search, from the starting points chosen by ECAM. If 0, DFBM will run only once from the best solution found so far.

Notes: 1. ECAM algorithm does not require a staring point $x0$, but the memory for $x0$ still has to be allocated in Maple as
`x0:=Vector(1..dimension,datatype=float[8]):`
2.Setting *maxiter* to a *negative* value will perform $|maxiter|$ iterations, but will not run DFBM at the end.

`float Minrandomstart0 (dim, x0, val, f, Xlo, Xup, maxiter)`

Performs global optimization using multistart local search. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),

$float\ x0[dim]$ - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
$float\ val$ - is the value of the objective function at the final point, the output parameter,
$f$ - is the reference to the user supplied function,
$float\ Xlo[dim]$ - is the lower bound on $x$,
$float\ Xup[dim]$ - is the lower bound on $x$,
*integer maxiter* - is the number of runs of the local optimization algorithm (DFBM). The number of iterations of DFBM local search is limited to 1000. The final optimization is performed with the limit of 10000 iterations.

`float Mindso0 (dim, x0, val, f, Xlo, Xup, speed, precision)`

Performs global optimization using DSO heuristic. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),
$float\ x0[dim]$ - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
$float\ val$ - is the value of the objective function at the final point, the output parameter,
$f$ - is the reference to the user supplied function,
$float\ Xlo[dim]$ - is the lower bound on $x$,
$float\ Xup[dim]$ - is the lower bound on $x$,
*integer speed* - is the speed factor, controlling how exhaustively the box domain is explored, it should take values from 1 to 4, 1 is the fastest but less accurate, 4 is slower but has a better chance to find the global minimum,
*integer precision* - is the desired precision, related to number of steps in the evolution of the dynamical system. This parameter can have values from 1 to 9 (9 means better accuracy but it is slower. 3 is the default).

`float Minecamdfbm0(dim, x0, val, f, LipConst, Xlo, Xup, maxiter, iterlocal)`

Performs global optimization using ECAM, but also runs DFBM at each iteration of ECAM. Every function value is a local minimum found by DFBM. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),
$float\ x0[dim]$ - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
$float\ val$ - is the value of the objective function at the final point, the output parameter,
$f$ - is the reference to the user supplied function,
$float\ LipConst$ - an estimate of the Lipschitz constant of $f$,
$float\ Xlo[dim]$ - is the lower bound on $x$,
$float\ Xup[dim]$ - is the lower bound on $x$,

*integer maxiter* - is the number of iterations. If 0, then the default value of 1000 will be used,
*integer iterlocal* - is the number of runs of local descent method DFBM to improve results of global search, from the starting points chosen by ECAM. If 0, DFBM will run only once from the best solution found so far.

`float Mindfbmecam0(dim, x0, val, f, Xlo, Xup, maxiter, iterECAM, dimECAM)`
Performs local nonsmooth optimization using DFBM, in combination with ECAM to improve line search and escape local minima. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),
*float* $x0[dim]$ - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
*float val* - is the value of the objective function at the final point, the output parameter,
$f$ - is the reference to the user supplied function,
*float* $Xlo[dim]$ - is the lower bound on $x$,
*float* $Xup[dim]$ - is the lower bound on $x$,
*integer maxiter* - is the number of iterations. If 0, then the default value of 10000 will be used,
*integer* $iterECAM$ is the number of iterations of ECAM,
*integer* $dimECAM$ is the dimension of the subspace on which ECAM performs global search, should be smaller than *dim* and typically smaller than 20.

`float Minecamdso0 (dim, x0, val, f, Xlo, Xup, maxiter, iterDSO)`
Performs global Lipschitz optimization in the domain specified by the box constraints $Xlo$, $Xup$. Every function value is a local minimum found by DSO algorithm. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),
*float* $x0[dim]$ - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
*float val* - is the value of the objective function at the final point, the output parameter,
$f$ - is the reference to the user supplied function,
*float* $Xlo[dim]$ - is the lower bound on $x$,
*float* $Xup[dim]$ - is the lower bound on $x$,
*integer maxiter* - is the number of iterations of ECAM.
*integer iterDSO* refers to the number of times DSO is called in the box regions supplied by ECAM.

`float Miniterativedso0 (dim, x0, val, f, Xlo, Xup, speed, precision, rounds)`
Performs global optimization using DSO heuristic. Parameters:
*integer dim* - is the number of variables (the dimension of the problem),

*float* $x0[dim]$ - is the starting point (initial approximation) on entry, and the final locally optimal solution on exit,
*float val* - is the value of the objective function at the final point, the output parameter,
$f$ - is the reference to the user supplied function,
*float* $Xlo[dim]$ - is the lower bound on $x$,
*float* $Xup[dim]$ - is the lower bound on $x$,
*integer speed* - is the speed factor, controlling how exhaustively the box domain is explored, it should take values from 1 to 4, 1 is the fastest but less accurate, 4 is slower but has a better chance to find the global minimum,
*integer precision* - is the desired precision, related to number of steps in the evolution of the dynamical system. This parameter can have values from 1 to 9 (9 means better accuracy but it is slower. 3 is the default),
*integer rounds* - is the number of rounds in the iterative process, can range from 1 to 10.

The following subroutines have very similar parameters to their counterparts with "0" at the end, but allow for linear equality and inequality constraints. The constraints are specified in the parameters

*integer lineq* - is the number of linear equality constraints,
*integer linineq* - is the number of linear inequality constraints.
*float*[] $AE$ is the array of size $lineq \times dim$ containing the entries of the matrix of linear equality constraints,
*float*[] $AI$ is the array of size $linineq \times dim$ containing the entries of the matrix of linear inequality constraints,
*float*[*lineq*] $RHSE$ is vector of size *lineq* of the right hand size of the system of linear equality constraints,
*float*[*linineq*] $RHSI$ is vector of size *linineq* of the right hand size of the system of linear inequality constraints,
*integer*[*dim*] *basic* is an optional vector of integers of size *dim*, specifying the indices of the desired basic variables, if the components of this vector are 0, then the basic variables are chosen automatically.

```
float Mindfbm(dim, x0, val, f, lineq, linineq, AE, AI, RHSE, RHSI, Xlo, Xup,
      basic, maxiter)
float Minecam(dim, x0, val, f, lineq, linineq, LipConst, AE, AI, RHSE, RHSI,
      Xlo, Xup, basic, maxiter, iterlocal)
float Minrandomstart (dim, x0, val, f, lineq, linineq, AE, AI, RHSE, RHSI,
      Xlo, Xup, basic, maxiter)
```

```
float Mindso(dim, x0, val, f, lineq, linineq, AE, AI, RHSE, RHSI, Xlo, Xup,
      basic, penalty, speed, precision)
float Miniterativedso(dim, x0, val, f, lineq, linineq, AE, AI, RHSE, RHSI,
      Xlo, Xup, basic, penalty, speed, precision, rounds)
```

*float penalty* - penalty parameter when inequality constraints are specified. Used to penalize constraints violations.

```
float Minecamdfbm(dim, x0, val, f, lineq, linineq, LipConst, AE, AI, RHSE,
      RHSI, Xlo, Xup, basic, maxiter, iterlocal)
float Mindfbmecam(dim, x0, val, f, lineq, linineq, AE, AI, RHSE, RHSI, Xlo,
      Xup,basic, maxiter, iterECAM, dimECAM)
float Minecamdso(dim, x0, val, f, lineq, linineq, LipConst, AE, AI, RHSE, RHSI,
      Xlo, Xup, basic, maxiter, iterDSO)
```

## 4.2 Constraints

Matrices of constraints have the following standard structure.

```
AE:=Matrix(1..lineq,1..dim,datatype=float[8],[[1,-1,0],[1,0,-1]]);
AI:=Matrix(1..linineq,1..dim,datatype=float[8],[[1,-1,0],[1,0,-1]]);
```

The values can be initialized as above (by specifying *lineq* or *linineq* rows in square brackets), or by direct assignment like `AE[1,1]:=1;`, etc.

The vectors of right hand sides should also be specified as

```
RHSE:=Vector(1..lineq,datatype=float[8],[1,1]);
RHSI:=Vector(1..linineq,datatype=float[8],[0,0]);
```

The inequality constraints are interpreted as $AIx \leq RHSI$.

## 4.3 Objective function

The objective function, whose reference is passed to `GANSO` subroutines, which do not use hardware floats, should have this standard format:

```
ff := proc( n, x ) #user's objective function
        local s;
        s:= evalf(x[1]^2+(x[2]-1)^2)+1.;
        s  #this last value is what is returned
    end proc;
```

It takes two parameters, integer $n$ – number of variables, real array $x$ of size $n$ – the input vector. The function returns a real value of $f(x)$.

Note the `evalf()` operator, which ensures that a real value (rather than an expression) is returned. The contents of the procedure is otherwise unrestricted. The vector $x$ is 1-based (as in Maple), i.e., we have $x[1], x[2], \dots, x[n]$. Ensure $n$ and $x$ do not conflict with variables with the same name defined elsewhere in Maple worksheet.

The second class of `GANSO` subroutines that use hardware floats (subroutines with $HF$ in their name) use the following format for the objective function

```
ff := proc( x1,x2,x3 ) #user's objective function
        local s;
        s:= x1^2+(x2-1)^2)+1.;
     end proc;
```

All variables are scalars and should be explicitly declared and referred to in the body of $f$, no matter how many variables are present. This is an example of calculation of the Euclidean norm of a vector $x$

```
ff := proc( x1,x2,x3,x4,x5 ) #user's objective function
        local s;
        s:= sqrt( x1^2+x2^2+x3^2+x4^2+x5^2);
     end proc;
```

# Chapter 5

# Examples of usage

## 5.1 Sample code

There are several examples of the usage of `GANSO` Maple toolbox provided in the distribution. The best way to use `GANSO` toolbox is to modify the examples provided.

Sample optimization problems
**Problem 1**
Rastrigin funciton Nr 1.

$$\text{minimize } f(x) = x_1^2 + x_2^2 - \cos(18x_1) - \cos(18x_2)$$
$$\text{s.t.} \ -1 \le x_i \le 1, i = 1, \ldots, 2.$$

There are no constraints except the box constraints, therefore we can use the simplified version of the subroutines (those ending with 0).

```
 GANSOPATH:="c:/yourdirectory/mwrap_ganso.dll":
 Minecam0 :=define_external('MWRAP_MinimizeECAM_0','MAPLE', LIB=GANSOPATH):
 Mindfbm0 :=define_external('MWRAP_MinimizeDFBM_0','MAPLE', LIB=GANSOPATH):
 Minrandomstart0 := define_external('MWRAP_MinimizeRandomStart_0',
         'MAPLE', LIB=GANSOPATH):
 Mindso0 := define_external('MWRAP_MinimizeDSO_0','MAPLE', LIB=GANSOPATH):

f := proc( n, x)  #user's objective function
       local s;
       s:= evalf(x[1]^2+x[2]^2-cos(18*x[1])-cos(18*x[2]));
       s  #this last value is what is returned
    end proc;

# defining vectors;
    dimension:=2;
    Lo:=Vector(1..2,datatype=float[8],[-1,-1]):
    Up:=Vector(1..2,datatype=float[8],[1,1]):
    iter:=1000: val:=1.0: # this is needed
    LipConst:=40.0:
    x0:=Vector(1..2,datatype=float[8],[1,1]);

# executing
    Minecam0(dimension,x0,val,f,LipConst,Lo,Up,iter,0);
    print(x0); #the ECAM solution
    Minrandomstart0(dimension,x0,val,f,Lo,Up,100);
    print(x0); #the random start solution
    Mindso0(dimension,x0,val,f,Lo,Up,3,3);
    print(x0); #the DSO method solution

# try local search from a staring vector(0.5,0.5), no constraints
    Lo:=Vector(1..2,datatype=float[8],[-1e20,-1e20]):
    Up:=Vector(1..2,datatype=float[8],[1e20,1e20]):
    x0:=Vector(1..2,datatype=float[8],[0.5,0.5]);
    Mindfbm0(dimension,x0,val,f,Lo,Up,iter);
    print(x0); #the solution
```

A faster method using hardware floats

```
 GANSOPATH:="c:/yourdirectory/mwrap_ganso.dll":
 MinecamOHF :=define_external('MWRAP_HFMinimizeECAM_0','MAPLE', LIB=GANSOPATH):
 MindfbmOHF :=define_external('MWRAP_HFMinimizeDFBM_0','MAPLE', LIB=GANSOPATH):
 MinrandomstartOHF := define_external('MWRAP_HFMinimizeRandomStart_0',
          'MAPLE', LIB=GANSOPATH):
 MindsoOHF := define_external('MWRAP_HFMinimizeDSO_0','MAPLE', LIB=GANSOPATH):

f := proc( x1,x2)  #user's objective function
        local s;
        s:= x1^2+x2^2-cos(18*x1)-cos(18*x2));
    end proc;

# defining vectors;
    dimension:=2;
    Lo:=Vector(1..2,datatype=float[8],[-1,-1]):
    Up:=Vector(1..2,datatype=float[8],[1,1]):
    iter:=1000: val:=1.0: # this is needed
    LipConst:=40.0:
    x0:=Vector(1..2,datatype=float[8],[1,1]);

# executing
    MinecamOHF(dimension,x0,val,f,LipConst,Lo,Up,iter,0);
    print(x0); #the ECAM solution
    MinrandomstartOHF(dimension,x0,val,f,Lo,Up,100);
    print(x0); #the random start solution
    MindsoOHF(dimension,x0,val,f,Lo,Up,3,3);
    print(x0); #the DSO method solution

# try local search from a staring vector(0.5,0.5), no constraints
    Lo:=Vector(1..2,datatype=float[8],[-1e20,-1e20]):
    Up:=Vector(1..2,datatype=float[8],[1e20,1e20]):
    x0:=Vector(1..2,datatype=float[8],[0.5,0.5]);
    MindfbmOHF(dimension,x0,val,f,Lo,Up,iter);
    print(x0); #the solution
```

**Problem 2**

A problem in the chemical equilibrium at constant temperature and pressure from the book J. Bracken and G.P. McCormick, "Selected Applications of Nonlinear Programming." John Wiley & Sons, New York, 1968.

$$\begin{aligned} \text{minimize } f(x) = \sum_{i=1}^{10} x_i \left( c_i + \ln \frac{x_i}{\sum_{j=1}^{10} x_j} \right) & \\ \text{s.t. } x_1 + 2x_2 + 2x_3 + x_6 + x_{10} &= 2 \\ x_4 + 2x_5 + x_6 + x_7 &= 1 \\ x_3 + x_7 + x_8 + 2x_9 + x_{10} &= 1 \\ x_i \geq 0, i = 1, \ldots, 10. & \end{aligned}$$

The vector $c$ is given as

$$c = (-6.089, -17.164, -34.054, -5.9514, -24.721, \\ -14.986, -24.1, -10.708, -26.662, -22.179).$$

A modification of this problem includes inequality constraint

$$x_3 + x_7 \leq 0.2.$$

In the code below constraints $x_i \geq 0.0001$ are used for the sake of simplicity of the objective function evaluation.

```
GANSOPATH:="c:/yourdirectory/mwrap_ganso.dll":
Mindfbm :=define_external('MWRAP_MinimizeDFBM','MAPLE', LIB=GANSOPATH):

# global variables: vector c
c:=array(1..10,[-6.089, -17.164, -34.054, -5.9514, -24.721, -14.986, -24.1,
         -10.708, -26.662, -22.179]):

f := proc( n, x ) #user's objective function
       local s,t,i;
       t:=add(x[i], i=1..n): # use add(), not sum()
       #  note abs under logarithm, to ensure that s is computable
       #  even for incorrect (negative) arguments
       s:=evalf(add(x[i]*(c[i]+log(abs(x[i]/t ))), i=1..n ));
            #this last value is what is returned
   end proc;
```

```
# defining vectors;
    dimension:=10; eps:=.0001;
    Lo:=Vector(1..10,datatype=float[8]):
    Up:=Vector(1..10,datatype=float[8]):
    x0:=Vector(1..10,datatype=float[8]):

    for i from 1 to dimension do
        Lo[i]:=eps;
        Up[i]:=1e20;
        x0[i]:=-1;
       od:
    iter:=1000: val:=1.0: # this is needed

    EQ:=Matrix(1..3,1..dimension,datatype=float[8],
    [[1,2,2,0,0, 1,0,0,0,1],
    [0,0,0,1,2, 1,1,0,0,0],
    [0,0,1,0,0, 0,1,1,2,1]]):
    RHSE:=Vector(1..3,datatype=float[8],[2,1,1]):
    IEQ:=Matrix(1..1,1..dimension,datatype=float[8]):
    RHSI:=Vector(1..1,datatype=float[8]):
    basic:=Vector(1..10,datatype=integer[4],[0,0,0,0,0,0,0,0,0,0]):
 # we need to declare all matrices/vectors, even if not used

# executing
    Mindfbm(dimension,x0,val,f,3,0,EQ, IEQ, RHSE, RHSI,Lo,Up,basic,iter);
    print(x0); #the  solution

# now set up the desired basic variables, 10-3 = 7 variables
    basic:=Vector(1..10,datatype=integer[4],[2,5,6,7,8,9,10,0,0,0]):
    for i from 1 to dimension do x0[i]:=-1: od: # clear x0

    Mindfbm(dimension,x0,val,f,3,0,EQ, IEQ, RHSE, RHSI,Lo,Up,basic,iter);
    print(x0); #the  solution

 # now add inequality constraint
    for i from 1 to dimension do
      x0[i]:=-1:
      IEQ[1,i]:=0;
    od:
    IEQ[1,3]:=1; IEQ[1,7]:=1; RHSI[1]:=0.2:

    Mindfbm(dimension,x0,val,f,3,1,EQ, IEQ, RHSE, RHSI,Lo,Up,basic,iter);
    print(x0); #the  solution
```

This is a faster method which uses hardware floats

```
 GANSOPATH:="c:/yourdirectory/mwrap_ganso.dll":
 MindfbmHF :=define_external('MWRAP_HFMinimizeDFBM','MAPLE', LIB=GANSOPATH):
 MindsoHF := define_external('MWRAP_HFMinimizeDSO','MAPLE', LIB=GANSOPATH):

 # global variables: vector c
 c:=array(1..10,[-6.089, -17.164, -34.054, -5.9514, -24.721, -14.986, -24.1,
          -10.708, -26.662, -22.179]):

 f := proc( x1,x2,x3,x4,x5,x6,x7,x8,x9,x10 ) #user's objective function
        local s,t;
        t:=x1+x2+x3+x4+x5+x6+x7+x8+x9+x10;
    s:= x1*(c[1]+ln(abs(x1/t)))+
        x2*(c[2]+ln(abs(x2/t)))+
        x3*(c[3]+ln(abs(x3/t)))+
        x4*(c[4]+ln(abs(x4/t)))+
        x5*(c[5]+ln(abs(x5/t)))+
        x6*(c[6]+ln(abs(x6/t)))+
        x7*(c[7]+ln(abs(x7/t)))+
        x8*(c[8]+ln(abs(x8/t)))+
        x9*(c[9]+ln(abs(x9/t)))+
        x10*(c[10]+ln(abs(x10/t)));
    end proc;

# defining vectors;
    dimension:=10; eps:=.0001;
    Lo:=Vector(1..10,datatype=float[8]):
    Up:=Vector(1..10,datatype=float[8]):
    x0:=Vector(1..10,datatype=float[8]):

    for i from 1 to dimension do
        Lo[i]:=eps;
        Up[i]:=1e20;
        x0[i]:=-1;
       od:
    iter:=1000: val:=1.0: # this is needed

    EQ:=Matrix(1..3,1..dimension,datatype=float[8],
    [[1,2,2,0,0, 1,0,0,0,1],
    [0,0,0,1,2, 1,1,0,0,0],
    [0,0,1,0,0, 0,1,1,2,1]]):
    RHSE:=Vector(1..3,datatype=float[8],[2,1,1]):
    IEQ:=Matrix(1..1,1..dimension,datatype=float[8]):
    RHSI:=Vector(1..1,datatype=float[8]):
    basic:=Vector(1..10,datatype=integer[4],[0,0,0,0,0,0,0,0,0,0]):
 # we need to declare all matrices/vectors, even if not used
# executing
    MindfbmHF(dimension,x0,val,f,3,0,EQ, IEQ, RHSE, RHSI,Lo,Up,basic,iter);
    print(x0); #the  solution

 # we can also use other methods, which would be too expensive to use
 # in the previous example
    for i from 1 to dimension do
        Lo[i]:=eps;
        Up[i]:=1;  # note that we need a bounded domain
        x0[i]:=-1;
       od:
    iter:=1000: val:=1.0: # this is needed

 MindsoHF(dimension,x0,val,f,3,0,EQ, IEQ,RHSE,RHSI,Lo,Up,basic,30.0,3,3);
```

## 5.2 Tips

- Use the subroutines with *HF* (that rely on hardware floats) for all but very simple problems.

- Always scale your problem, including the objective function and constraints. Try to scale objective function to a reasonable range, say [-100,100]. Scale variables to a range around [-10,10]. This will help the algorithms deliver better accuracy and stability.

- Ensure the constraints are linearly independent. The algorithm will return an error otherwise. Also, if choosing the basic variables yourselves, ensure they are linearly independent.

- Use adequate parameters. All algorithms are generic, they can be executed with any dimension or number of iterations, but be realistic in your expectations. ECAM can be safely used for up to 6-8 variables, but its complexity grows very quickly with the number of iterations and dimension. Use about 10000 iterations, if more, be prepared for a long running time and large memory demand. If using ECAM+DFBM or RandomStart, at each iteration expensive local search by DFBM will be performed – be prepared to wait. When using DFBM+ECAM, use small dimensional subspace for ECAM ($< 10$).

- Do not unnecessarily specify box constraints for DFBM, especially the upper bound. It works faster with fewer constraints. However all other methods do require box constraints.

- Always try the methods on simpler problems before using them for production runs. Estimate the running time/memory requirements using a small number of iterations.

- Specifying negative *maxiter* parameter in all methods (*speed* parameter in MinDSO and MinIterativeDSO) has the effect of not using DFBM to improve the solution at the last stage. This will return the minimum found by the global algorithm (ECAM or DSO), not improved by local search.

## 5.3 Where to get help

The software library `GANSO` and its components, are distributed by CIAO AS IS, with no warranty, explicit or implied, of merchantability or fitness for a particular purpose. CIAO will provide limited technical support for registered users, by electronic media. CIAO, at its sole discretion, may provide advice to registered users on the proper use of `GANSO` and its components.

Any queries regarding technical information, sales and licensing should be directed to `j.ugon@ballarat.edu.au`. For updates check `http://www.ganso.com.au`

# Bibliography

[Bag02] A. Bagirov. A method for minimization of quasidifferentiable functions. *Optimization Methods and Software*, 17:31–60, 2002.

[Bag03] A. Bagirov. Continuous subdifferential approximations and their applications. *Journal of Mathematical Sciences*, 115:2567–2609, 2003.

[BB02] L.M. Batten and G. Beliakov. Fast algorithm for the cutting angle method of global optimization. *Journal of Global Optimization*, 24:149–161, 2002.

[Bel04] G. Beliakov. The cutting angle method - a tool for constrained global optimization. *Optimization Methods and Software*, 19:137–151, 2004.

[Bel05] G. Beliakov. A review of applications of the cutting angle methods. In A. Rubinov and V. Jeyakumar, editors, *Continuous Optimization: Current Trends and Modern Applications*, pages 209–248. Springer, New York, 2005.

[BGLS00] J.F. Bonnans, J.C. Gilbert, C. Lemarechal, and C.A. Sagastizabal. *Numerical Optimization. Theoretical and Practical Aspects.* Springer, Berlin, Heidelberg, 2000.

[BR01] A. Bagirov and A. Rubinov. Modified versions of the cutting angle method. In N. Hadjisavvas and P.M. Pardalos, editors, *Convex analysis and global optimization*, volume 54 of *Nonconvex optimization and its applications*, pages 245–268. Kluwer, Dordrecht, 2001.

[Cla83] F.H. Clarke. *Optimization and Nonsmooth Analysis.* John Wiley, New York, 1983.

[DR86] V. F. Demyanov and A. Rubinov. *Quasi-differential Calculus.* Optimization Software, Inc., New York, 1986.

[DR95] V.F Demyanov and A.M. Rubinov. *Constructive Nonsmooth Analysis.* Peter Lang, Frankfurt am Main, 1995.

[HP95] R. Horst and P. M. Pardalos. *Handbook of Global Optimization.* Nonconvex Optimization and its Applications. Kluwer Academic Publishers, Dordrecht; Boston, 1995.

[HPT00] R. Horst, P. Pardalos, and N.V. Thoai, editors. *Introduction to Global Optimization*, volume 48 of *Nonconvex Optimization and its Applications.* Kluwer Academic Publishers, Dordrecht, 2nd edition, 2000.

[HT93] R. Horst and H. Tuy. *Global optimization: deterministic approaches.* Springer-Verlag, Berlin; New York, 2nd rev. edition, 1993.

[Mam04] M.A. Mammadov. A new global optimization algorithm based on dynamical systems approach. In A. Rubinov and M. Sniedovich, editors, *6th Intl Conf. on Optimization: Techniques and Applications*, Ballarat, 2004. University of Ballarat.

[MO05] M.A. Mammadov and R. Orsi. H_infinity systhesis via a nonsmooth, nonconvex optimization approach. *Pacific Journal of Optimization*, 1:405–420, 2005.

[MRY05] M.A. Mammadov, A. Rubinov, and J. Yearwood. Dynamical systems described by relational elasticities with applications to global optimization. In A. Rubinov and V. Jeyakumar, editors, *Continuous Optimisation: Current Trends and Modern Applications*, pages 365–385. Springer, New York, 2005.

[MW93] J. Moré and S. J. Wright. *Optimization Software Guide.* SIAM, Philadelphia, 1993.

[Pin96] J. Pintér. *Global Optimization in Action: Continuous and Lipschitz Optimization–Algorithms, Implementations, and Applications.* Nonconvex optimization and its applications; v. 6. Kluwer Academic Publishers, Dordrecht; Boston, 1996.

[Roc70] R.T. Rockafellar. *Convex Analysis.* Princeton University Press, Princeton, 1970.

[Rub00] A.M. Rubinov. *Abstract Convexity and Global Optimization*, volume 44 of *Nonconvex optimization and its applications.* Kluwer Academic Publishers, Dordrecht; Boston, 2000.

[SS00] R. G. Strongin and Y. D. Sergeyev. *Global Optimization with Nonconvex Constraints: Sequential and Parallel Algorithms.* Nonconvex optimization and its applications; v.45. Kluwer Academic, Dordrecht; London, 2000.