OptiM 9.0 Manual

Optimizing Your Solutions

Written by: Jacek Mieloszyk

Warsaw, 2017

1. INTRODUCTION

1.1 WHAT IS OPTIM

It's an engineering toolbox, written in C++, that can be used for any problem which requires numerical optimization. OptiM uses the most known optimization algorithms. It is capable of linking programs for analyze that support scripting, macros or use input files. Objective function is determined in a simple dynamically linked library. Additional functions contain useful tools to define optimization tasks quickly. The program is very flexible and can fulfill the most demanding needs of the user.

1.2 SOFTWARE LICENSE AGREEMENT

This is a legal agreement ("this Agreement") between OptiM authors ("the Authors") and the licensee ("the Licensee"). The Authors license the OptiM Software ("the Software") only if all the following terms are accepted by the Licensee. The Software includes the OptiM byte code executable and any files and documents associated with it.
By installing the Software, the Licensee is indicating that he/she has read and understands this Agreement and agrees to be bound by its terms and conditions. If this Agreement is unacceptable to the Licensee, the Licensee must destroy any copies of the Software in the Licensee's possession immediately.

1. LICENSE CONDITIONS AND RIGHTS

Libraries included with the program remain integral part of the program and can not be linked with other software.

The Licensee has right to use the Software and its documentation for non-commercial purpose.

The Licensee may use the Software for commercial and common education purposes after registration. The registration includes sending name and logo of the institution that will use the software on an e-mail*: jmieloszyk@meil.pw.edu.pl*, with declaration containing permission to process the data to advertise OptiM.

The Licensee may not reverse engineer, disassemble, decompile, or unjar the Software, or otherwise attempt to derive the source code of the Software.
The Licensee acknowledges that Software furnished hereunder is under test and may be defective. No claims whatsoever can be made on the Authors based on any expectation about the Software.

Revised and/or new versions of the License may be published with new versions of the Software.

2. TERM, TERMINATION AND SURVIVAL

The Licensee may terminate this Agreement at any time by destroying all copies of the Software in possession.
If the Licensee fails to comply with any term of this Agreement, this Agreement is terminated and the Licensee has no further right to use the Software.
On termination, the Licensee shall have no claim on or arising from the Software.

3. NO WARRANTY

The Software is licensed to the Licensee on an "AS IS" basis. The Licensee is solely responsible for determining the suitability of the Software and accepts full responsibility and risks associated with the use of the Software.

4. MAINTENANCE AND SUPPORT

The Authors are not required to provide maintenance or support to the Licensee.

5. LIMITATION OF LIABILITY

In no event will the Authors be liable for any damages, including but not limited to any loss of revenue, profit, or data, however caused, directly or indirectly, by the Software or by this Agreement.

6. DISTRIBUTION

The Software can be copied and redistributed, under condition that no fee is charged for theservice

1.3 VERSIONS REFERENCE

Version	Date	Changes
1.0		-Simple program with Steepest Descent and Newton method
2.0	2009-08-21	-Choelsky matrix factorization for Newton method added
		-Polak Ribiere direction search method added
		-Quasi Newton direction search method added
		-Alfa search algorithm with Strong Wolf conditions added
		-Alfa search interpolation with second and third order interpolation added
		-Objective function difference stop criterion added
		-Gradient equal to zero stop criterion added
	2010-01-26	-Conditions for reliable gradient computation
2.1	2010-05-06	-Improved Alfa search
		-Armijo condition for Alfa search added
		-Improved Hessian computation of Newton method
		-Smaller memory allocation requirements
		-Dump Hessian restarts for Quasi Newton method added
		-Penalty quadratic function constrains added
	2010-07-09	-Some other errors removed
3.0	2010-08-01	-Genetic Algorithms
		-GUI
		-OptiM libraries
	2011-01-05	-Many user work efficiency improving function
3.1	2011-01-13	-Tester of objective function added
		-Functions for users added
		-Fast update of OptiM.ini file added
		-Real time results displaying
		-Warnings about lack of input files
		-Warnings about overwriting files
		-Other warnings
	2011-01-27	-Many errors removed
3.2	2011-02-01	-GA cos(MOM) improvement
		-resolution of variables in GA
		-write dF and p to Log
	2011-08-15	-Small errors removal
4.0	2011-08-16	-Linux version of OptiM
		-GUI main layout changed
		-Monte Carlo optimization added
		-Swarming optimization added
		-Ability to define users functions of derivatives
	2011-10-28	-Other small changes and errors removal
5.0	2012-12-28	-Hook-Jeeves optimization method added

		-Powell optimization method added -Nelder-Mead optimization method added
5.1	2013-05-31	-Statistics tool added -Stop criterion based on statistics analysis added -Equally distributed starting points added
6.0	2014-10-25	-Lightly loaded dynamic libraries -Multithreads -Improved declaration of dynamic tables
7.0 – 7.1	2015-09-17	-Simulated Annealing optimization method added -Multiple functions added for Pareto front -Ploting function during optimization and for post processing -Better configuration files handling and warnings -Improved functionality of tools -Number of bugs improved
8.0	2016-04-12	-GA chromosome bug removed -Files choosing improved -Powel algorithm bugs removed -Plot bugs fixed -Added legend for plots -Flag for plotting during optimization -VarEdit tool added for setting design variables -Hard limit for constrains -Option to continue optimization process -Installation procedure changed
9.0	2017-02-18	-Gain bug fixed -Xflags and Cflags incorporated in VarEdit -Added option for analytical gradient -Improved example for external software execution in Linux -Nelder Mead algorithm computed parallel -Gradient and Hessian computed parallel -Added limits for axis in plots -Added popup menu in plot for printing and picture saving -Added information about compiler used in About -Improved path to manual from OptiM's menu

(PLEASE REPORT ANY OTHER ERRORS OR DESIRABLE IMPROVMENTS)

2. THEORY GUIDE

This section describes theory that is hidden inside OptiM algorithms. Better understanding of the processes driving the software will improve user skills.

Defining optimization task is very demanding. It is normal that the task must be corrected few times to achieve sufficient analysis reliability and get desirable quality results.

Function optimization without constrains can be defined by basic equation (2.1). It means that the function f, which depends on the optimization variables x, should be minimized. The function has n number of design variables x, which are real numbers.

$$\min_x f(x) \qquad x \in R^n \tag{2.1}$$

Basic notation:

f - minimized objective function
x - vector of design variables
α - one directional step/gain size

i - number of the design variable in x vector
$_k$ - current iteration number
$_n$ - number of the design variables

2.1 DIRECTIONAL NON-GRADIENT OPTIMIZATION

This section describes family of deterministic directional non-gradient methods.

2.1.1 Simulated Annealing

Simulated annealing algorithm was inspired by physical process of annealing in metal parts. During the process metal can change it's structure and the process drives towards minimization of the internal stresses. The bigger the temperature of the metal is, the bigger changes may go on. Because of the structure changes, temporarily structure can have worse properties, but eventually it has less internal stresses.

This analogy is applied in the optimization algorithms. Following the procedure on an example of a "hill climber", simulated annealing algorithm starts from a chosen point (it may be random) and tries to climb on the highest hill Fig. 2.1. At first it is easy to find better solution and local optimum, but to get even higher first the climber has to go lower. After making this step the climber can go forward to the highest point.

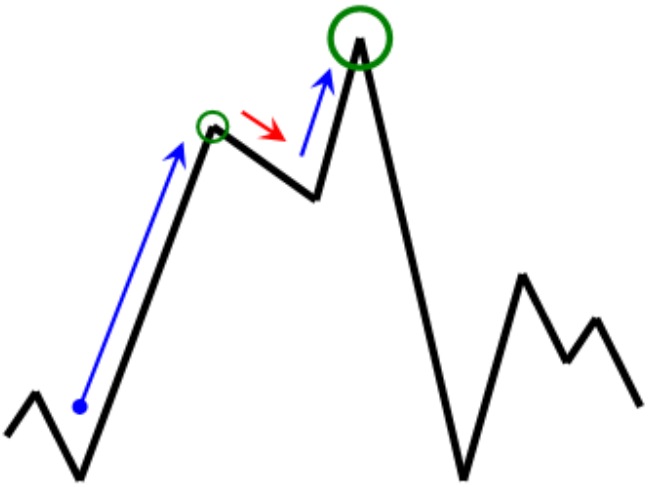

Figure 2.1 Simulated annealing algorithm – hill climber.

In the subsequent iterations the numerical algorithm updates the design variables with the formula (2.2). Parameter *sign* is chosen randomly "+", or "-". *res* is a relaxation parameter set by user. Difference between objective function value, current and from previous iteration, is described by (2.4), at the beginning it is equal to 1. Based on the new derived values of design variables objective function is computed. Depending on the objective functions difference and current artificial temperature acceptance probability (2.4) of the current solution is calculated. If the temperature is higher the probability to accept current solution is also higher. Artificial temperature drops during subsequent iterations according to (2.5), allowing for less number of worse solutions with iterations.

$$\vec{x}_{k+1} = \vec{x}_k \cdot sign \cdot res \cdot df \cdot (\vec{x}_{\max} - \vec{x}_{\min}) \tag{2.2}$$

$$df = f_k - f_{k-1} \tag{2.3}$$

$$p = e^{\frac{df}{T}} \tag{2.4}$$

$$T_{k+1} = T_k \cdot (1 - CoolRate) \tag{2.5}$$

2.1.2 Hooke-Jeeves

In the beginning trial steps are taken, by changing every optimization variable, to improve the objective function. After the trial steps overall search direction is estimated to minimize objective function in this estimated direction. The algorithm is described in few steps below, on an example of two dimensional function, with a drawing Fig. 2.2.

Starting point of optimization is set and value of the objective function in that point calculated. Vector of changes $\vec{V}$ is initialized for every search direction (design variable) with set by user perturbation dV (2.6).

$$V_i = dV \qquad for\ every\ i \tag{2.6}$$

Trial step in the x_1 direction is taken with the step size V_1 added to the variable vector (2.7). Than the objective function is estimated, with the changed x_1 and vector of design variables $\vec{x}_{trial}$ (2.8). If the objective function was improved the trial vector of the design variables is accepted and this becomes the new starting point.

$$x_{1,trial} = x_1 + V_1 \tag{2.7}$$

$$f(\vec{x}_{trial}) \tag{2.8}$$

The procedure is repeated for the variable x_2. The first attempt to minimize the objective function with step like (2.7) failed and the search direction was changed (2.9) (step 4 on Fig. 2.2). If the step still doesn't improve the solution, there are another attempts with reduced step size (2.10) and procedure repeat from (2.7).

$$x_{1,trial} = x_1 - V_1 \tag{2.9}$$

$$V_i = V_i \cdot reduceStep \tag{2.10}$$

After making trial steps in the separate directions of variables, whole vector of direction $\vec{V}$ is set. Next the algorithm tries to make global optimization step in the direction of the vector $\vec{V}$. The global step is done with *Increment Step* defined by user (2.11), (2.12) and objective function evaluated again (2.8).

$$\vec{V} = \vec{V} \cdot IncrementStep \tag{2.11}$$

$$\vec{x}_{trial} = \vec{x} + \vec{V} \tag{2.12}$$

If the objective function was not improved with initially incremented vector $\vec{V}$ for the next attempt the incrimination is reduced (2.13), vector of varibles updated (2.12) and objective function computed once again (2.8).

$$\vec{V} = \vec{V} \cdot reduceStep \tag{2.13}$$

Described behavior of the algorithm is repeated on Fig. 2.2 (steps 5-9, 9-14, 15-18).

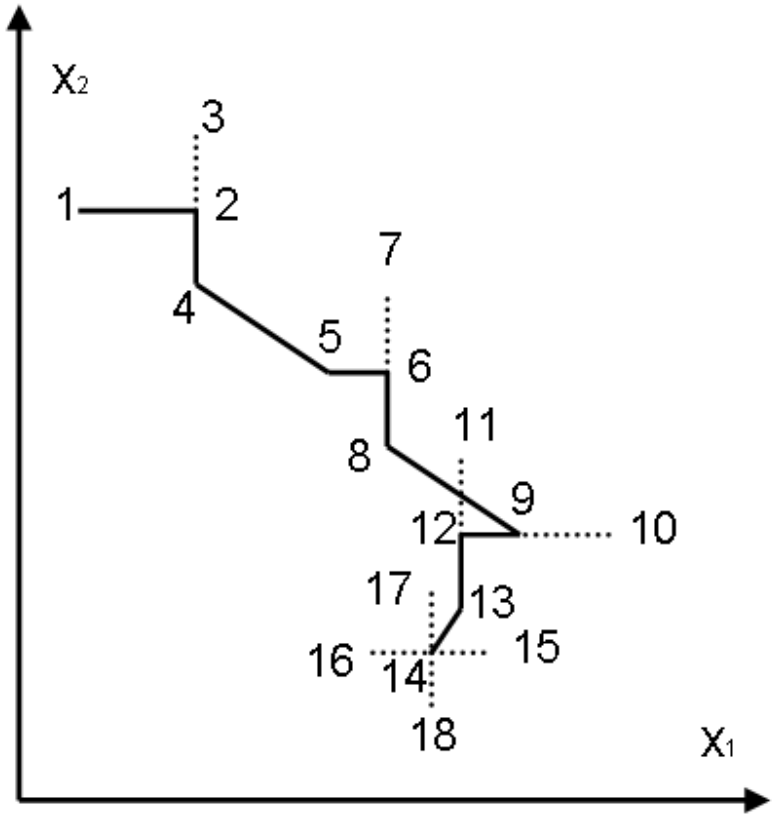

Figure 2.2 Hook-Jeeves optimum search method.

2.1.3 Powell

Powell optimization method is similar to Hooke-Jeeves. The main difference is that the coordinate system in which search direction is estimated changes with iterations to improve speed of convergence.

First optimum gain m for every variable is estimated, utilizing single variable minimization method of Golden Proportion. During the Golden Proportion search trial variable is changed () and objective function evaluated with the single changed variable (2.15).

$$x_{i,trial} = x_i - m_i \cdot e_i \tag{2.14}$$

$$f(\vec{x}_{trial}) \tag{2.15}$$

Direction vector is estimated basing on the formula (2.16). Algorithm tries to minimize in subsequent iterations the objective function using α with the formula (2.17)

$$\vec{e}_k = \frac{\left(\vec{x}_k - \vec{x}_{k-1}\right)}{\left|\vec{x}_k - \vec{x}_{k-1}\right|} \tag{2.16}$$

$$\vec{x}_{k+1} = \vec{x}_k + \alpha \cdot \vec{e} \cdot \vec{m} \tag{2.17}$$

The biggest gain of the single variable minimization m_i is found. If condition (2.18) is satisfied, than the base vector for the next iteration is updated (2.19).

$$\frac{m_k \cdot d}{\left|x_{k+1} - x_k\right|} \geq 0.8 \tag{2.18}$$

$$\vec{e}_{k+1} = \vec{e}_k \tag{2.19}$$

2.1.4 Nelder-Mead

This method, which is also called Crawling Simplex, deals well even with very nonlinear functions. Simplex can be described in n-dimensional space as polyhedron with n+1 vertices. During optimization process according to rules implemented in the algorithm position of the vortices of the polyhedron change, heading to the optimum point. The algorithm is described in points.

1. Simplex initialization with $n+1$ vortices in n dimensional space.

2. Computation of objective function in the vertexes of the polyhedron.

3. Finding indexes of the vertexes with the worst and the best values of the objective function.

4. Computation of the center of symmetry of the simplex, neglecting the worst vertex, according to formula (2.20). Estimation of objective function in the center of symmetry of the simplex (2.9).

$$P' = \frac{\sum_{i=1}^{n+1} P_i}{n}, i \neq vertex_{worst} \tag{2.20}$$

$$fs = f\left(P'\right) \tag{2.21}$$

5. Reflection of the worst vertex with accordance to the point P' and estimation of the value of the objective function in the new vertex P^*.

6. If objective function in the P^* is smaller than previously estimated minimum objective function value, than vertex P^{**} is calculated utilizing formula (2.22) and estimating objective function value in that point.

$$P^{**} = (1+\gamma)P^* - \gamma P' \tag{2.22}$$

   If $f(P^{**}) < f_{min}$, then replace $f(P_{worst})$ with point P^{**}.
   If $f(P^{**}) \geq f_{min}$, then replace $f(P_{worst})$ with point P^*.

7. If objective function in the P^* is bigger than previously estimated minimum objective function value, than make contraction of P_{worst} with accordance to P' utilizing formula (2.23)

$$P^{***} = \beta Ph + (1-\beta)P' \tag{2.23}$$

   If $f(P^{***}) \geq f_{max}$, make simplex reduction utilizing formula (2.24)

$$Pi = \frac{Pi + Pl}{2},\ i = 1, 2, ..., n+1 \tag{2.24}$$

   If $f(P^{***}) < f_{max}$, then replace P_{worst} with point P^{***}.

8. If $f(P^*) < f(P_i)$ for $i = 1, 2, ..., n+1, i \neq h$, replace P_{worst} with point P^*.

9. Check stop criterion.

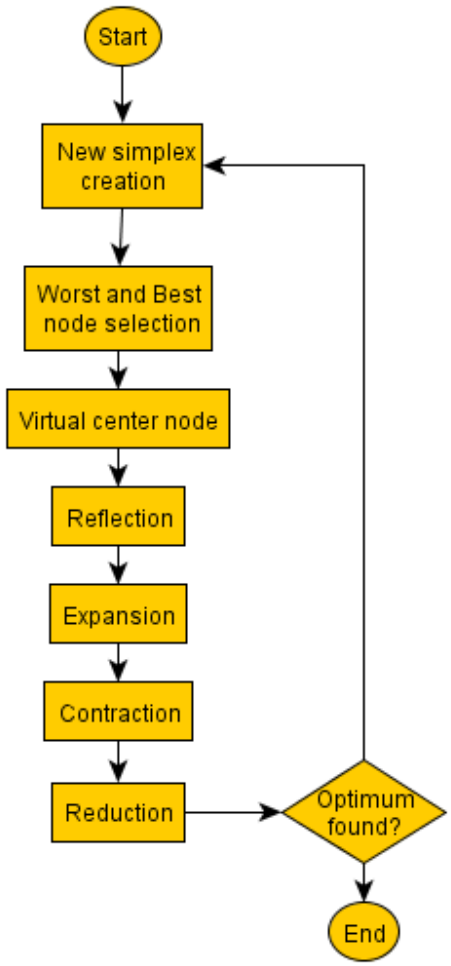

Figure 2.3 Nelder-Mead optimization schema.

2.2 GRADIENT BASED OPTIMIZATION

Gradient methods belong to bigger group of deterministic optimization methods. In directional methods starting point is chosen arbitrary, from initial calculations, or statistical assumptions. The closer the initial configuration is to the optimum design the faster and more probably the best solution will be found. It is possible to visualize such a task for a function depending on two variables, *Fig. 2.4* shows the starting point, optimum solution and isolines of the optimized function. In the real conditions designer does not know the layout of the isolines. The question is: How to get to the optimum solution in the fastest way.

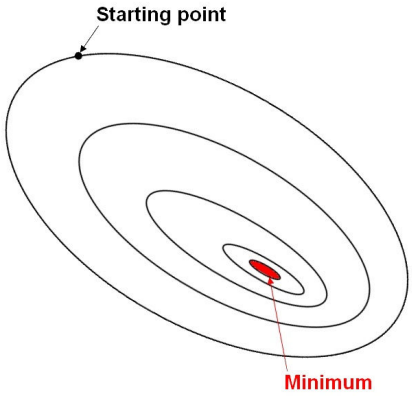

Figure 2.4 Optimization space with starting point and minimum.

Gradient algorithm has to make small control steps to calculate local directional derivatives, which show how fast the function values rises or drops in the tried direction. *Fig. 2.5* shows this situation. Now the best direction, to change the design variables can be estimated. The direction depends on the direction search algorithm used. More complex algorithms make some corrections using knowledge about values of the second derivatives of analyzed function and history of optimization during iterations. This subject will be explained more precisely in the following sections.

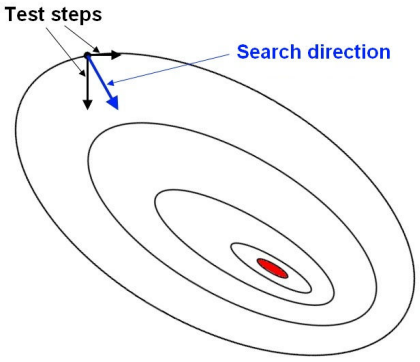

Figure 2.5 Estimation of search direction.

Having the direction in which the design will move, it is still unknown how big step should be taken *Fig 2.6*. Taking to big or to small step may increase solution time and in the worst cases the solution can be driven far away from the optimum point. Direction search methods have to be introduced here, which estimate the optimum step size. After this operation design is updated and moved to the new point from which the procedure is repeated *Fig 2.7*.

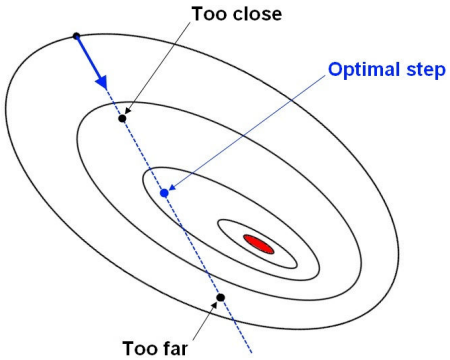

Figure 2.6 Estimation of step size.

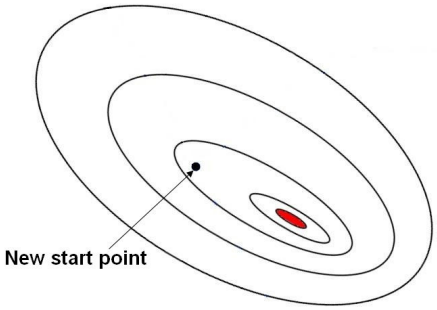

Figure 2.7 New starting point.

Direction search methods

OptiM has four types of direction search methods implemented, varying in complexity and efficiency. In all of the methods directional derivatives are computed in the same way from equations (2.25) and (2.26). Solution from the first derivative is used to calculate the second derivatives to save computational time. Every new derivative needs one objective function estimation and symmetrical values of Hessian are copied.

$$\frac{\partial f}{\partial x_i} \approx \frac{f\left(x+\Delta x_i\right)-f\left(x\right)}{\Delta x_i} \tag{2.25}$$

$$\frac{\partial^2 f\left(x\right)}{\partial x_i \partial x_j} = \frac{f\left(x+\Delta x_i+\Delta x_j\right)-f\left(x+\Delta x_i\right)-f\left(x+\Delta x_j\right)+f\left(x\right)}{\Delta x_i \Delta x_j} \tag{2.26}$$

More complex methods are butter for design functions which have many variables and are strongly nonlinear. On the other hand the simplest Steepest Descend method is the most stable method and should theoretically guaranty reaching the optimum.

2.2.1 Steepest Descend Method

The simplest method which uses only the first derivative and does not take into account optimization history. There is mathematical proof that using the method guaranties reaching the optimum. This method is highly inefficient for strongly nonlinear functions, but deals well with simpler cases.

2.2.2 Conjugate Gradient Method

Compared to the Steepest Descend Method it uses additionally history of optimization process, which enables to make corrections of the direction estimated. It behaves much butter in cases with nonlinear functions.

2.2.3 Quasi Newton Method

Method derived from classical Newton Method. It should be comparably efficient to the pure Newton Method, but uses only the first directional derivatives. Second derivatives are derived on the base of optimization history and already computed first derivatives. Good quality and small costs. There is big family of Quasi Newton Methods, but OptiM uses the BFGS algorithm which is considered as the most efficient one and most often used.

2.2.4 Newton Method

In this method first and second derivatives are computed directly. The method does not depend on the optimization history, but direct computations of the second derivatives give the best estimation of direction. The cost of computing the Hessian is returned for highly nonlinear functions making this method the most efficient for such cases.

2.2.5 One Direction Search Methods

At the beginning initial step size is tried. If the step satisfies conditions described below, the solution is updated and the process starts again. If the conditions are not satisfied new optimum step is estimated and tried. From the case with the initial step new information is available – $f(\alpha)$, $df/d\alpha$, so it is possible to create second order function dependent only on the α value and estimating the optimum of the function. If the second step size also is bad function of third order is used to estimate appropriate step size.

Armijo Condition

It is the simplest and necessary condition. If the optimized function after doing estimated step size is smaller than the initial one the condition is satisfied (2.27). If it is not satisfied different step size has to be estimated.

$$f\left(x_k + \alpha \cdot p_k\right) < f\left(x_k\right) + C_1 \cdot \alpha \cdot \nabla f_k^{\,T} \cdot p_k \tag{2.27}$$

Strong Wolf Condition

This condition ensures that the optimization process is well convergent and that the step size isn't too small, ensuring efficiency of the optimization process. This condition can be expressed mathematically by taking in to account equations (2.27) and (2.28).

$$\left|\nabla f\left(x_k + \alpha_k \cdot p_k\right)^T \cdot p_k\right| \le C_2 \cdot \left|\nabla f_k^{\,T} \cdot p_k\right| \tag{2.28}$$

Solution update

New design variables are updated according to equation (2.29).

$$x_{k+1} = x_k + \alpha_k \cdot p_k \tag{2.29}$$

2.3 HEURISTIC OPTIMIZATION METHODS

Heuristic optimization methods do not have the same restrictions as directional ones. The optimized function does not get caught by local optimums. Although, they have stochastic nature, so every time user starts optimization, results can be slightly different or the time arriving to the global optimum can change.

2.3.1 Monte Carlo Optimization

Monte Carlo optimization method is simple, yet very efficient to find global minimum. First the search area is defined and in the specified boundaries points are selected or randomly drawn. Algorithm finds currently the best solution around which new search area is defined *Fig. 2.8*. In every iteration the search area is shrunken with *Radius* parameter. If the area is decreased to slowly minimum may be found after long time. If it is decreased to quickly it may cut off the optimum solution.

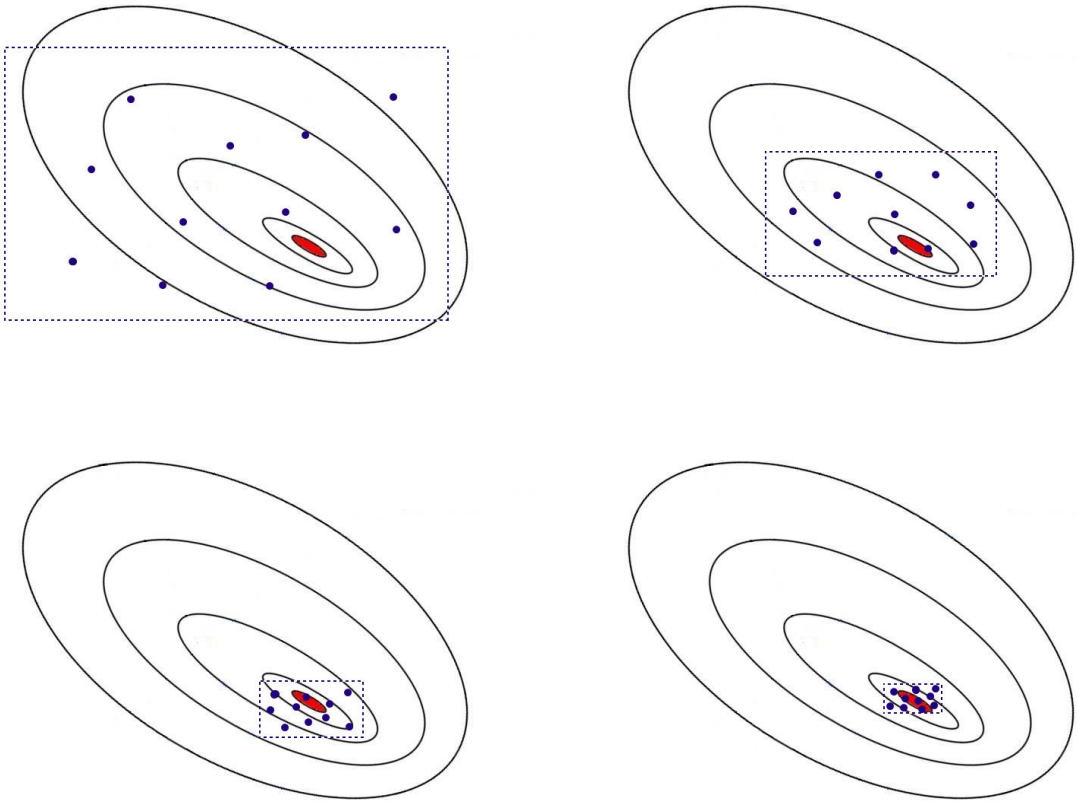

Figure 2.8 Monte Carlo optimization method visualization.

2.3.2 Genetic Optimization

Design variables real values are transformed into genes of the length defined by the user. Minimum value will be represented by zeros and maximum by ones and every value between by combination of 0 and 1. See the example below:

Airplane design parameters:

T/W	W/S	AR	TR	t/c	Clmax		
000	010	100	001	111	101	←	the whole string is a genome

(genes of length 3, parameter T/W has min value, parameter t/c has max value)

Depending on the length of the genes one can achieve resolution of the design variable by calculating (2.30). Longer genes increase design variables resolution, but also computation time.

$$resolution = \frac{x_{\max} - x_{\min}}{2^l - 1} \tag{2.30}$$

Every individual from the population has it's own features described by the genome (design variables). Individuals with high objective function values are favored by the *Measure of Merit* (*MOM*) function, which can have values of 1 for the best and 0 for the worst individual. Measure of Merit can have different characteristics enabling user to promote the best individuals or discourage the worst ones. In OptiM user can set few types of *MOM* Fig 2.9.

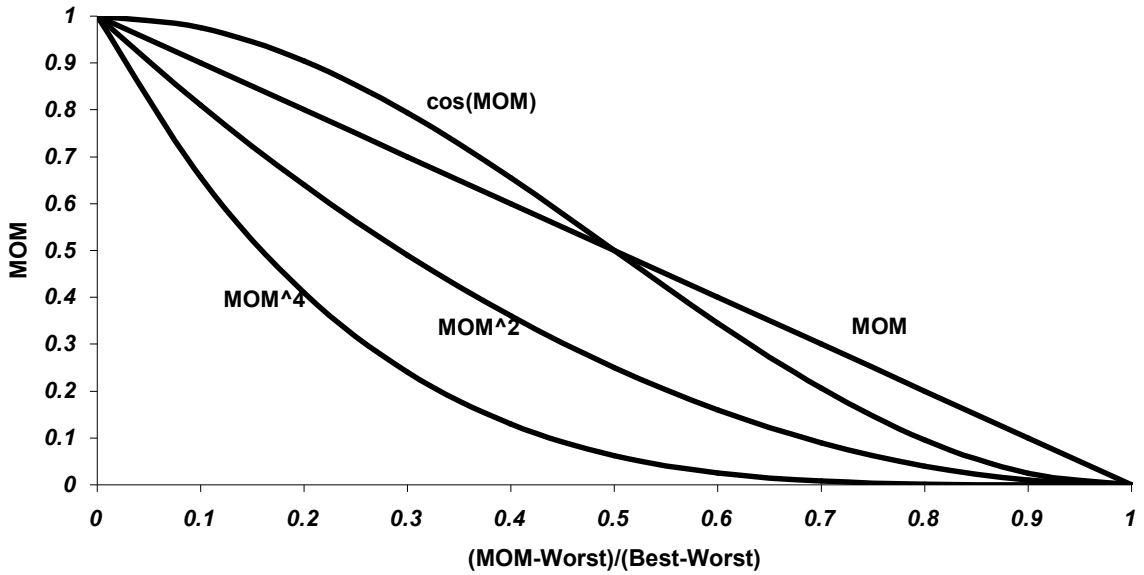

Figure 2.9 Different types of Measure of Merit function.

To avoid counter-convergence effect, where the best individual in new population is worse than the best individual from the previous population, *Elitism* rule is introduced. The rule says that the best individual or group of the best individuals is passed to the next generation. If to many individuals will pass to the next generation, they can easily dominate the optimization and cause premature convergence. To create new generation few strategies are available in OptiM. The strategies use common methods of Crossover and Mutation.

2.3.2.1 Crossover Methods

Crossover method is a way of mixing genomes of two selected individuals. Three types of crossover are implemented in OptiM.

Single Point:

The genomes of each individual are divided into two parts. First half of genome is taken from the first individual and second part from the second individual. Combined parts create new individual.

Genome 1 010 110 101 000 011
Genome 2 111 011 110 000 010
Genome New 010 110 100 000 010

Uniform:

Genomes single positions/bits from two selected individuals are compared and if the bits have different values, 0 or 1 is randomly selected, else the bit is rewritten to the new genome.

Genome 1 010 110 101 000 011
Genome 2 111 011 110 000 010
Genome New 110 111 111 000 011

Parameter Wise:

Whole genes are compared of the two individuals, if they are not the same one of the genes is randomly selected for the new individual.

Genome 1 010 110 101 000 011
Genome 2 111 011 110 000 010
Genome New 111 110 110 000 010

2.3.2.2 Mutation Method

After new genomes are created from Crossover method they go through mutation process, which changes values of bits from 0 to 1 and opposite for the defined percentage of the genome. The percentage of the genome to be mutated can be set manually or automatically from the equation (2.31).

$$P_{mutation} = 1 - \left(1 - \left(\frac{1}{propfac}\right)\right)^{numbits} \tag{2.31}$$

It is advised to set automatic mode for most cases, which will influence only small part of the genome to be mutated. In those cases crossover will play the crucial role, but mutation will not allow for premature convergence. If user wants the optimization process to be dominated by the mutation method than manual setting is desirable.

2.3.2.3 Killer Queen Strategy Selection

This is the simplest strategy in which crossover is not used at all. From the population the best individual is selected and the new population is created by high mutation of the best individual. The mutation rate can get up to 90% and should be set manually.

2.3.2.4 Roulette Strategy Selection

In this strategy individuals selected for reproduction have some probability with which they can be selected. The probability of selection can be expressed by (2.32), and visualized like in *Fig 2.10*.

$$SlotSize_i = \frac{MOM_i}{\sum_i MOM_i} \tag{2.32}$$

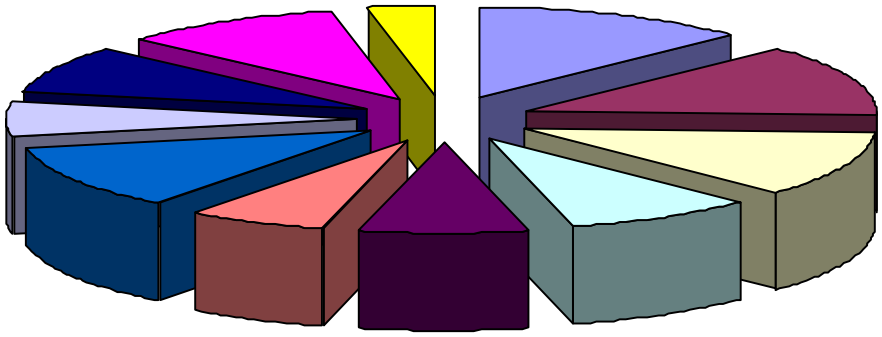

Figure 2.10 Probability of selection of individuals.

Then the roulette is spined and values between 0 and sum of *MOM* is drawn, which indicates one individual for reproduction. Roulette is spined as many times as many individuals are needed for reproduction to create new population.

2.3.2.5 Breeder Pool Selection Strategy

In this strategy specified percentage of the best individuals are selected for reproduction. Only those individuals are used for creation of the new population. From the specified group individuals for crossover and mutation are selected in a manner described in a roulette strategy.

2.3.2.6 Tournament Selection Strategy

This time tournament is held between individuals of the current population. Two couples are selected from the population in a manner described in roulette strategy. Couples fight and the one with better objective function value fights next with the winner from the second couple. Winner of the tournament is allowed for reproduction. Other individuals for reproduction are selected in a same way.

2.3.3 Swarming Optimization

Swarming optimization mimics behavior of a swarm looking for example for food. Swarm consists of particles selected or randomly drawn from defined search area. Every particle tries to find the optimum by movement in the search area. The movement is defined by the particle position and vector of velocity. The velocity vector is built from three components: velocity of the particle from the previous iteration, velocity from recorded design variables for the best objective function evaluated of the particle and velocity from the design variables of the globally best particle *Fig. 2.11* (2.33). Some randomness of the movement is also added by r variable to make the algorithm feasible and robust. The random variable is higher or lower from 1 by the fraction of user defined resolution (2.34). Constants $w, s1, s2$ control influence of the components of velocity on the final velocity vector of the particles. The vector of velocity influences design variables (2.35). Wrongly chosen constants can stop optimization process to converge. OptiM has set commonly used constants, but they may need to be adjusted for a particular case.

$$\vec{V}_k = w \cdot \vec{V}_{k-1} + s_1 \cdot r_1 \cdot \left(\vec{x}_{best\_for\_the\_particle} - \vec{x}_k\right) + s_2 \cdot r_2 \cdot \left(\vec{x}_{globally\_best} - \vec{x}_k\right) \quad (2.33)$$

$$r = \frac{\left(rand(\ ) \cdot \left(resolution + 1\right)\right)}{resolution} \qquad \text{where} \quad rand(\ ) = \begin{cases} 1 \\ -1 \end{cases} \quad (2.34)$$

$$\vec{x}_{k+1} = \vec{x}_k + \vec{V}_k \quad (2.35)$$

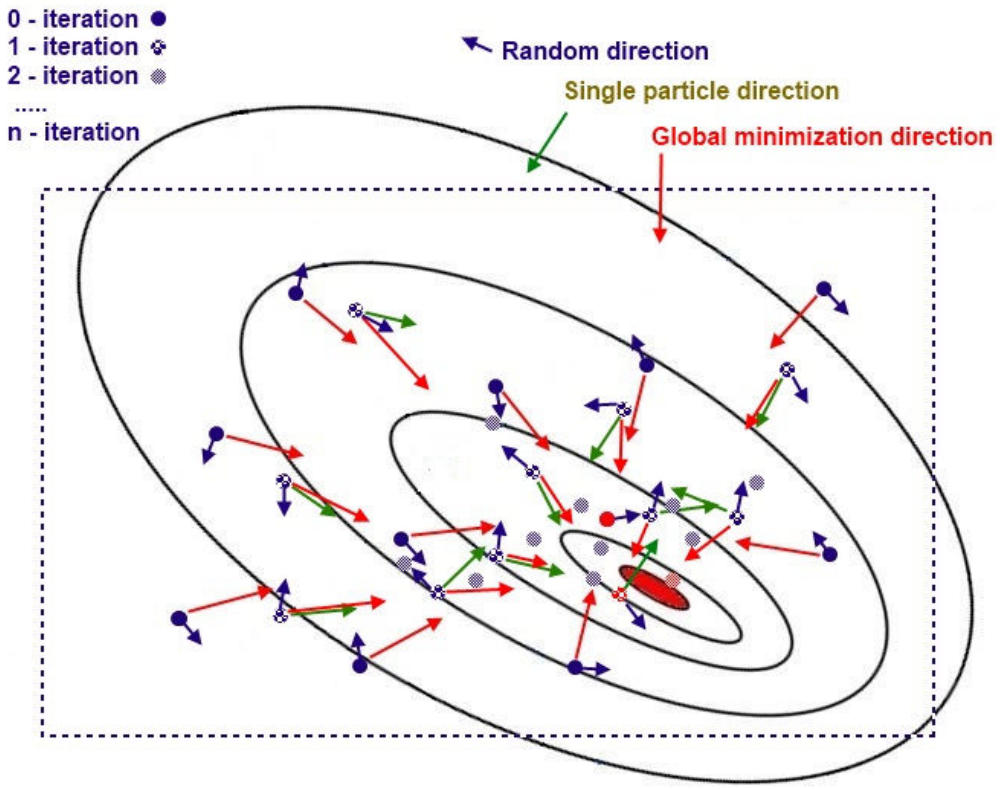

Figure 2.11 Example of swarming optimization.

3. OPTIM INSTALTION

3.1 OPTIM INSTALATION PROCEDURE

Download *OptiM* package appropriate for the used system Linux/Windows. Run setup and proceed with the instructions. To create new optimization tasks compiler to create the dynamic library is needed. *OptiM* and original libraries were written using *GCC* under Linux and *MinGW* under Windows. Although, *OBJ_F( )* function, which can be treated as *main( )* function for the library, has prefix *extern "C"*, making it independent from code language, compiler type and version.

In Windows version of setup there is possibility to install also, simple IDE interface for C++ *Dev-Cpp*. After installation it has to be configured and coupled with *MinGW*, what is described in chapter 3.3.

3.2 COMPILATOR INSTALATION

3.2.1 Linux

Most Linux distributions have already installed C++ compiler. To make sure GCC compiler is installed on your Linux system, by writing *gcc –version* in terminal. You will get message telling what version of GCC is installed on your system. If there is no compiler installed please refer to many tutorials available in the internet how to do it for your distribution.

3.2.2 Windows

Install *MinGW*, which can be downloaded from it's official site: *http://www.mingw.org/.* During installation make sure to install *MSYS*, terminal emulating Linux environment (compilation of the library should be also possible in *Cygwin* environment). Create new folder in home directory of *MSYS*, it will be your new working directory for the project you start. You can also move *OptiM*'s *Examples* directory there.

Check if file *fstab* exists: `C:/MinGW/msys/1.0/etc/fstab`. If not, create it and add record: `C:\MinGW /mingw` (Assuming the original installation path, otherwise set your instaltation path). This will map *MinGW* on the *MSYS* environment, making compilers visible for the *MSYS*. To make sure installation was completed successfully run *MSYS* and write *gcc –version* in terminal. You will get message telling what version of compiler is installed on your system.

3.2.3 Dev-Cpp IDE

If option to install *Dev-Cpp* was chosen *OptiM* dynamic libraries can be compiled in the IDE. *Dev-Cpp* is simple and reliable interface, but it isn't developed any more. To update compilers it uses it has to be configured first. From menu choose *Tools* >

Compiler Options. Add new compiler configuration with + button, give it a name (for example MinGW) and apply. It should be available in the compilers list Fig. 3.1. Next step is to add directories to the MinGW compiler resources. On Fig. 3.2 directories (for the default MinGW directory instalation) are presented in subsequent window tabs. Directories with *msys* are only needed if for example user plans to install own additional libraries. Apply all the changes with *OK* button.

After creating new project it's setting also have to be adjusted, to use the newly created compiler configuration. From menu choose *Project* > *Project Options*. In the *Compiler* tab choose the new compiler configuration Fig. 3.3.

To create the dynamic library for *OptiM*, *makefile* has to be appropriately written. The easiest way is to point to one of the *makefiles* from examples and modify them if needed. To do these, in the *Makefile* tab, check box for custom *makefile* and choose path to the *makefile* Fig. 3.4.

Examples contain **.dev* project files with all the adjustments.

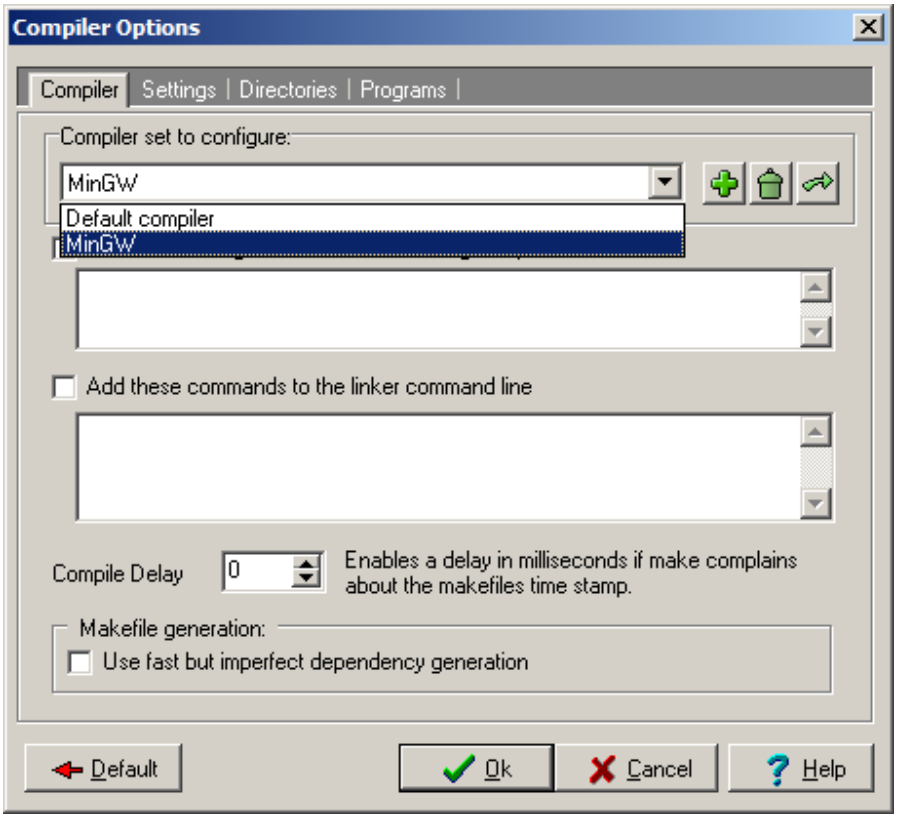

Figure 3.1 Adding new compiler configuration.

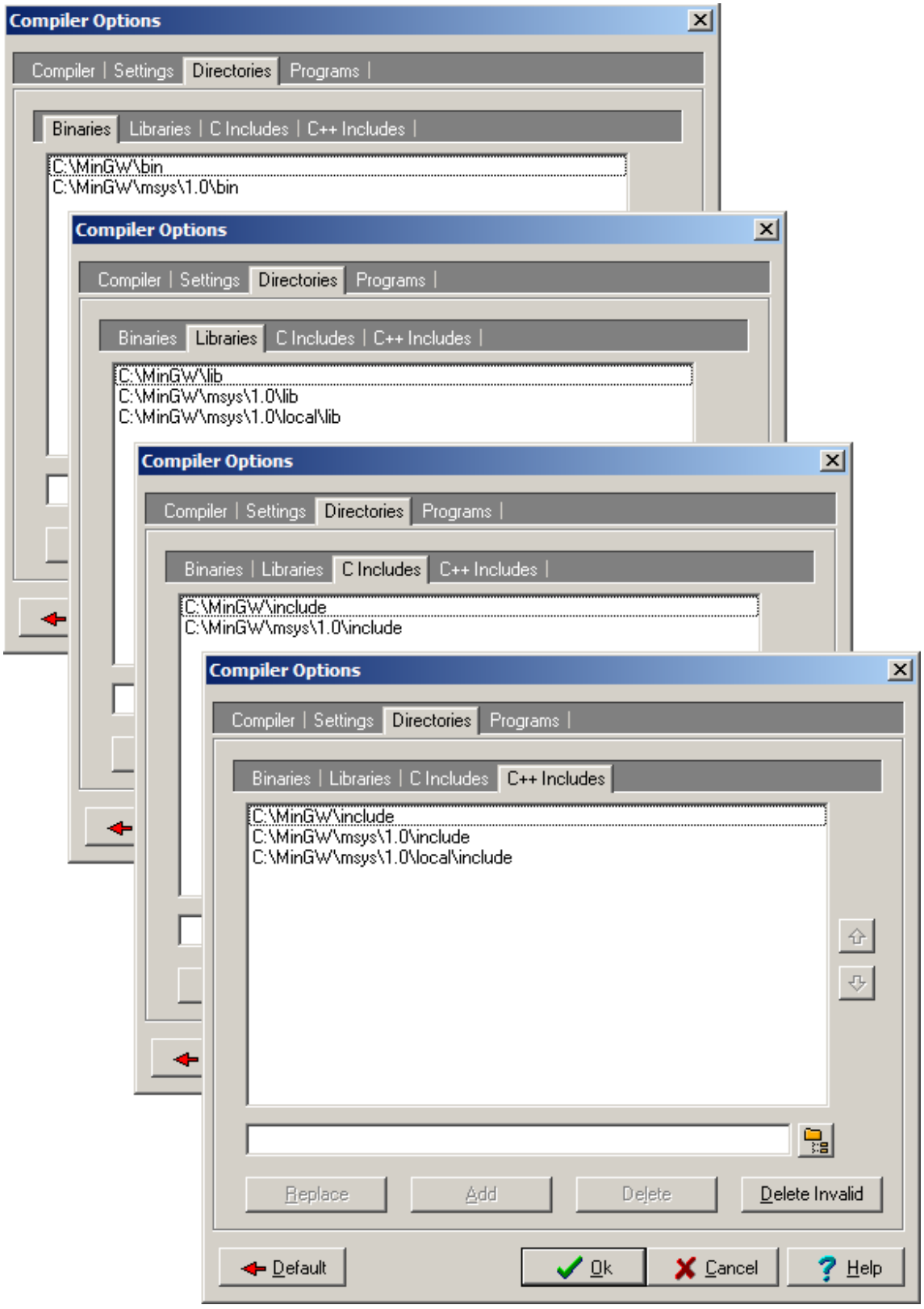

Figure 3.2 Setting directories for MinGW compiler.

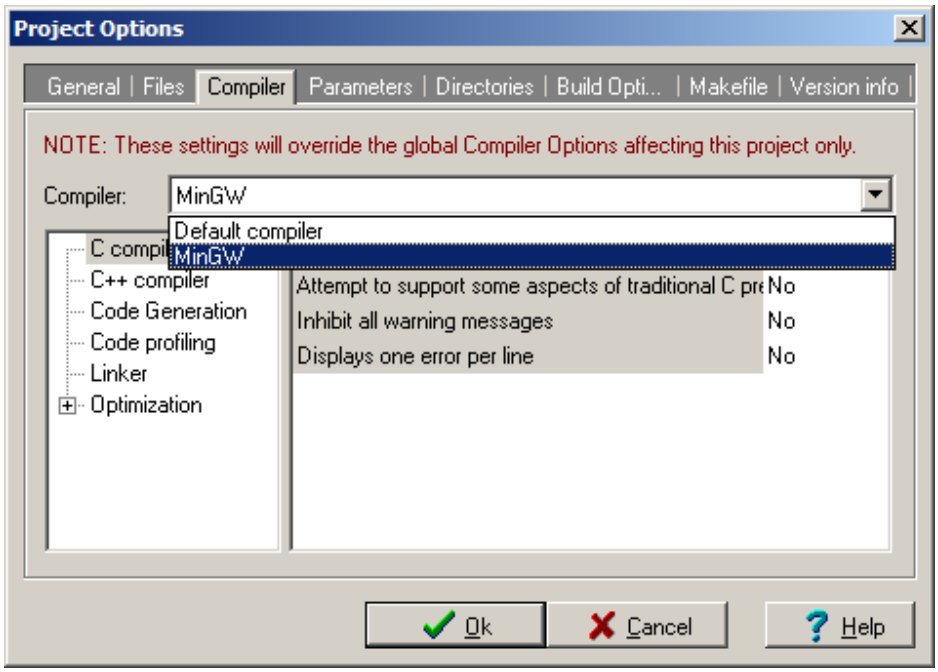

Figure 3.3 Choosing new compiler configuration for a project.

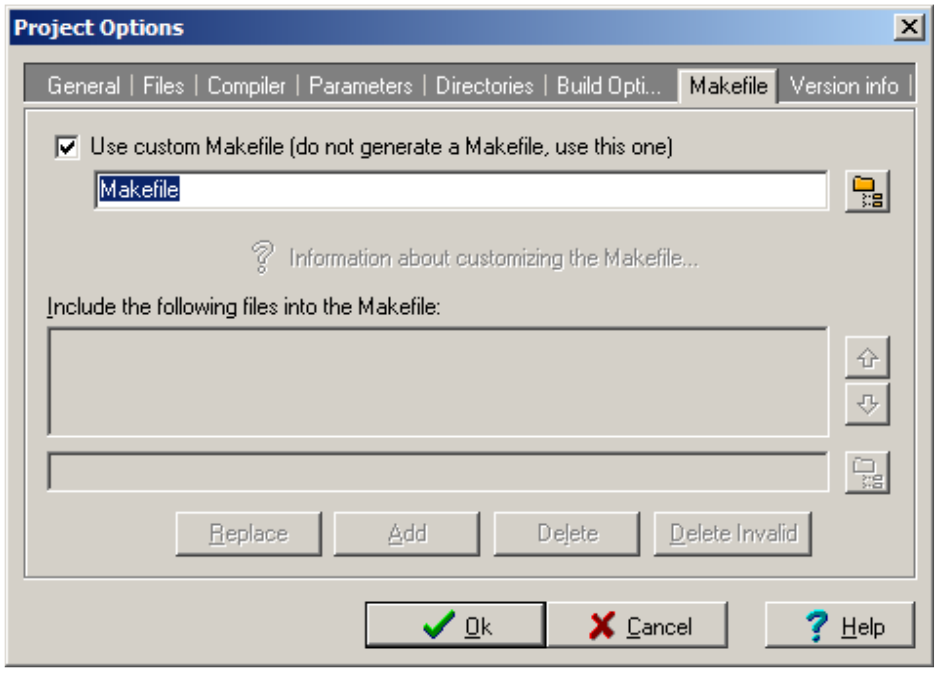

Figure 3.4 Choosing custom make file for a project.

3.3 COMPILING THE DYNAMIC LIBRARIE

Open *OptiM_Lib.cpp* file in any text editor. It is very convenient to use editor with C++ syntax highlight, many are available free on the web. Basic *OptiM_Lib.cpp* contains similar code to the one shown below. First few lines include header files of standard libraries and object files and also header file of the *OptiM* library. User can add other header files. Below included files the definition of the objective function is present. Input parameters are respectively:

- current iteration number,
- thread identification number (useful in parallel computations, especially for defining unique file names for every thread, threads Id starts from "0")
- structure with parameters from OptiM GUI (sometimes it is useful to get from the structure number of design parameters, or working directory path for own path creation – shown in examples)
- vector of design variables
- vector of constrains
- vector of objective functions

Inside the function user defines the optimization task. To improve work efficiency utilities for *OptiM* are added (*OptiM_Tools.h*). They contain helpful functions of matrix algebra and other, see chapter 5 for more details. The utilities have to be always included, because of the structure *Param* definition.

```
#include <Param.h>
#include <OptiM_Tools.h>
#include "OptiM_Lib.h"

using namespace std;

double OBJ_F(int current_iter, int thread_id, Param *Par,
             double *X, double *C, double *F)
{
    F[0] = 100*(X[1] - X[0]*X[0])*(X[1] - X[0]*X[0]) + (1-X[0])*(1-X[0]);
    return F[0];
}
```

OptiM was written utilizing cross platform C++ code. Compilation process of the *OptiM* dynamic library is very similar for Linux and Windows systems. After editing *OptiM_Lib.cpp* in the terminal go to the *Lib* directory and type *make* to compile the library. If *Dev-Cpp* was appropriately configured from menu choose *Execute* > *Compile,* or press appropriate button, or use key shortcut *Ctrl* + *F9*. Run OptiM to optimize.

4. OPTIM USER GUIDE

After opening OptiM you will see the main window with logo of the program Fig.4.1. At the top of the window main menu is placed. The same menu is available under right mouse button for faster access. Most commands from the menu have key shortcuts, which are listed in the appendices. All options that can be set are described in the following parts of the manual. After pushing button with OptiM logo in the lower right corner of the window optimization start. Currently used solver for computations can be seen on the title bar of the OptiM. The big logo will disappear and currently computed results will appear in the main window. Bars at the bottom of the window show progress of optimization.

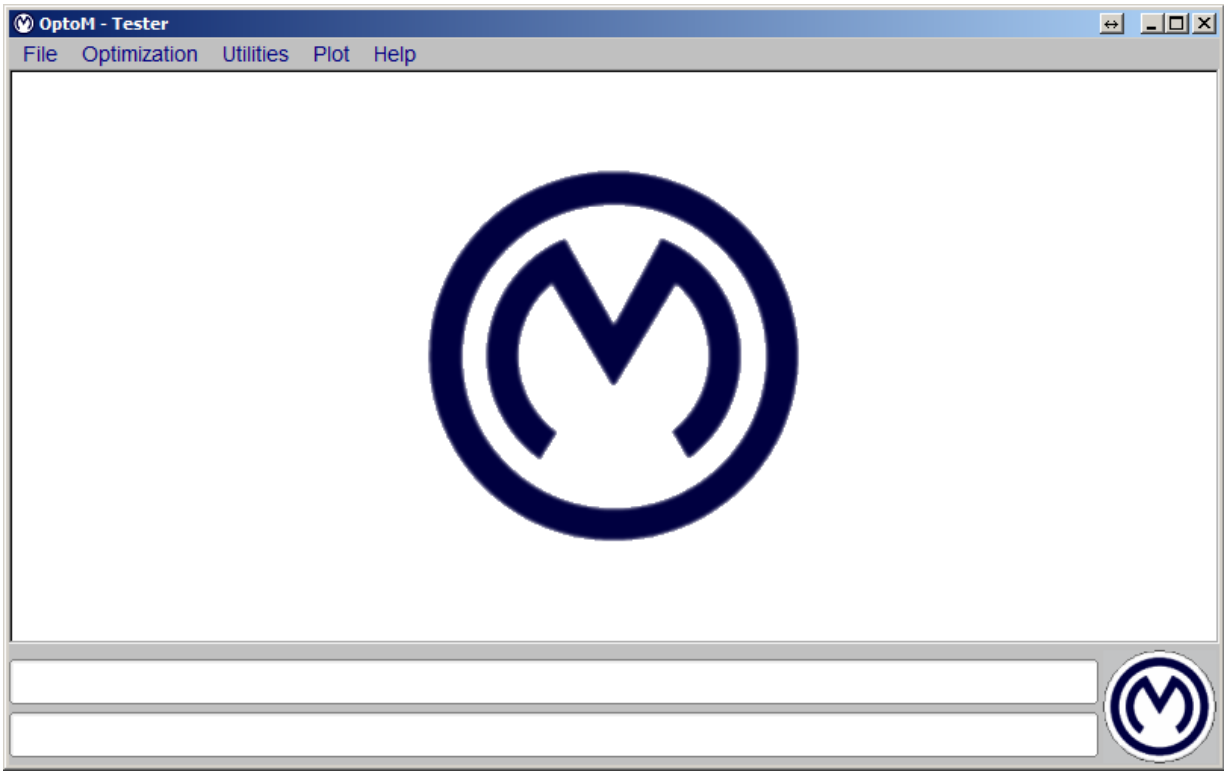

Figure 4.1 OptiM main window.

4.1 File

4.1.1 File > Parameters

In this window Fig. 4.2 all current settings in OptiM are printed. The settings can be saved in *OptiM.om* file, or with any other name. OptiM during opening checks in the local directory if *OptiM.om* file exists. If it is found, parameters from the file are set, else the default parameters are set. User can also read in parameters from different file manually.

OptiM can be started in batch mode with file name as parameter. Structure of the file have to be the same as *OptiM.om* file structure.

The best way to change om files content, is to make changes of parameters in the OptiM GUI to avoid input errors.

OK – closes the *Parameters* window

Load Parameters – loads OptiM parameters from selected file

Save Parameters – saves OptiM parameters to selected file

Update OptiM.om – saves or overwrites OptiM.om file in the working directory

Attention!!! There are no warnings about files overweighting!!!

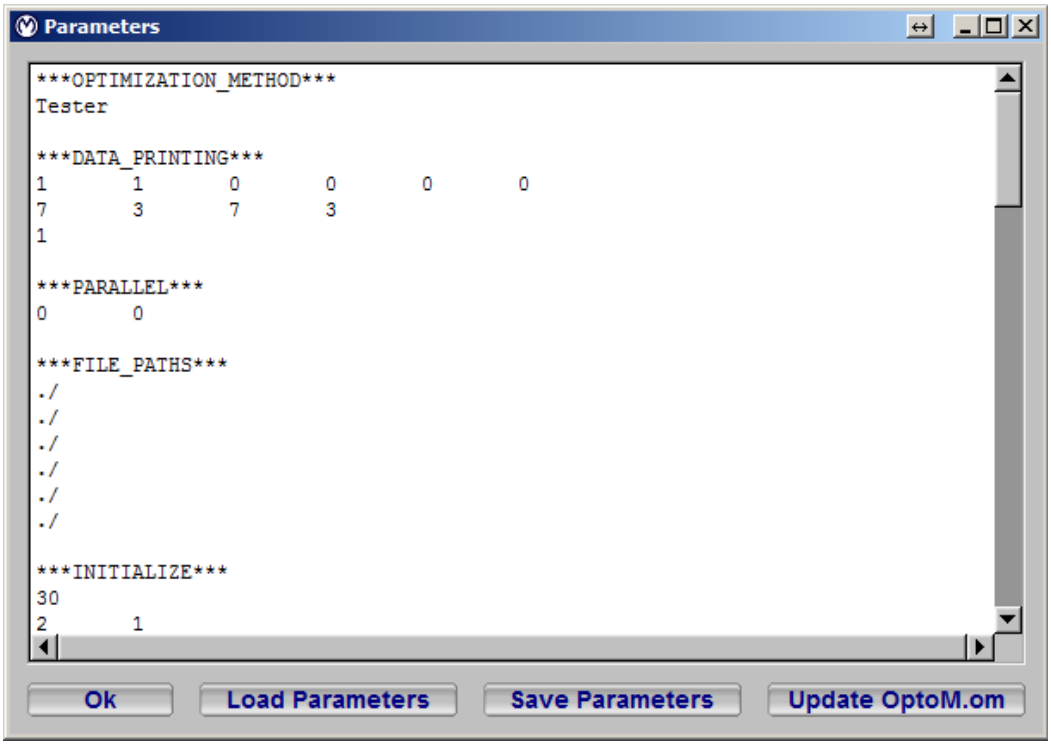

Figure 4.2 Parameters window.

4.1.2 File > Data Format

In this window Fig. 4.3 output data can be formatted to user needs.

On Screen – Things that are displayed on screen can be set here: design variables, objective function, constrains, sum of constrains, derivatives, sum of derivatives

Format on screen – Format for printing numbers on screen is defined hire: field width containing the whole number of digits and number of digits after the dot

Format to file – Format for printing to file is defined hire: field width containing the whole number of digits and number of digits after the dot

Separator in file – Many programs have different demands for white spaces, two options are available as a separator: four spaces or tab

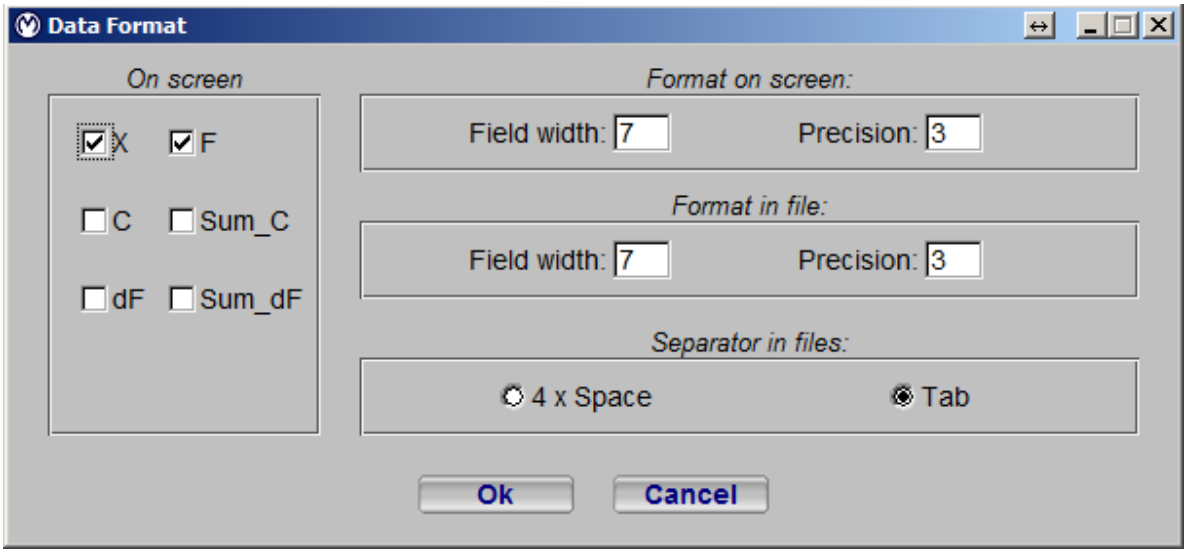

Figure 4.3 Data Printing window.

4.1.3 File > Parallel

Optimization algorithms, which have to analyze groups of solutions can be computed parallel. Currently parallelized algorithms are:

- Monte Carlo
- Genetic Algorithm
- Particle Swarm Optimization (PSO) - Swarming

By default maximum number of processor cores is detected and set, but user can change the number. Leaving one, or two processor cores not running allows for working with less computationally demanding programs. Parallel processing can be easily turned off by the flag in the *Parallel* window Fig. 4.4

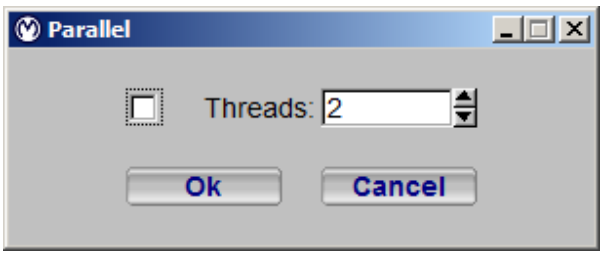

Figure 4.4 Setting for parallel optimization.

4.1.4 File > Exit

Leaves the OptiM.

4.2 Optimization

4.2.1 Optimization > File paths

In this window all needed file paths are set. Every directory can be set using *Browse* button.

WorkDir – This is the working directory of the project. It is added at the beginning of the paths if local directories are detected for particular files. It is especially useful when migrating from one computer to other, which has different catalogues structure. If local directories are set for files, only working directory catalogue has to changed.

Lib – Compiled dynamic library with the optimization task.

X Limit – Configuration file with defined Min and Max range of the design variables .

Solution – Output file with optimization history.

Log – Log file with detailed information about optimization process. Very useful while detecting problems with optimization process.

Pareto – Extended results from all analysis done during optimization. Useful for creating Pareto fronts and advanced post processing.

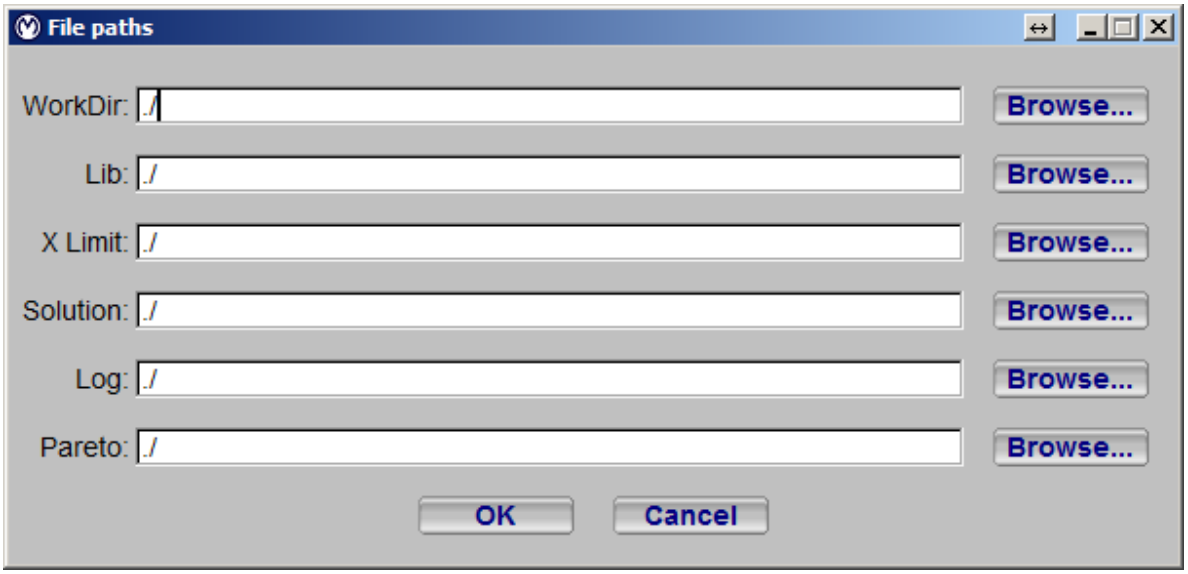

Figure 4.5 File paths window.

4.2.2 Optimization > Initialize

Initial settings for optimization.

Iteration Limit – Limit of iterations after which optimization stops

Number of X – Number of design variables

Random/Uniform – Way of creating design variables for optimization algorithms with populations. Variable's values are randomly selected, or uniformly distributed between *Xmin, Xmax*.

Number of C – Number of constrains

μ – Coefficient for constrains penalty function

Number of F – Number of objective functions

Initialize Fo – Flag to initialize starting value of the main objective function (gradient method)

Fo – Value of the main objective function if initialization flag is on

OK – Approves changes and closes the window

Cancel – Closes window without approving changes

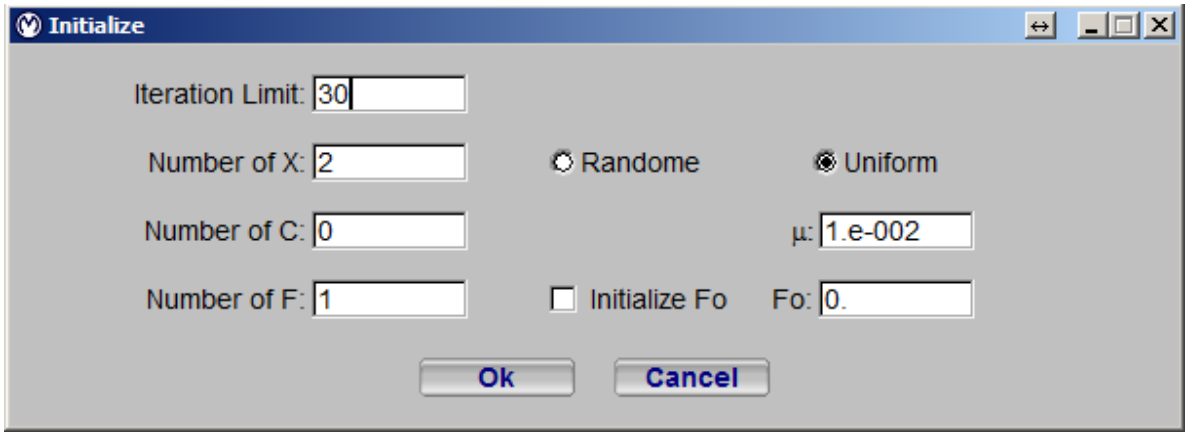

Figure 4.6 Initialize window.

4.2.3 Optimization > Solver

Place where optimization solver is set. Tester – is a single objective function evaluation, for testing appropriate definition of the objective function and connection of dynamic library. During optimization process objective function is called in the exactly same way, but with changing design parameters. User can choose optimization solvers between:

- Tester
- Annealing
- Hooke Jeeves
- Powell
- Nelder Mead
- Gradient
- Monte Carlo
- Genetic
- Swarming

Details of the algorithms were described in the Theory Guide section.

4.2.4 Optimization > Annealing

Settings for *Annealing* optimization algorithm.

Temperature – Artificial temperature (see *Theory Guide* for details)

CoolingRate – (see *Theory Guide* for details)

Resolution – Relaxation parameter (see *Theory Guide* for details)

Scale dF – If checked than objective function df (see *Theory Guide* for details)difference is multiplied by $T_{current}/T_{ref}$

dFmin – Minimum allowable df difference. If the difference is smaller constant specified value is used.

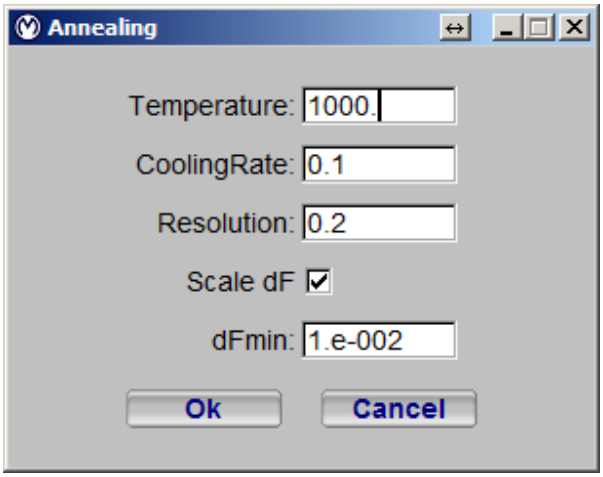

Figure 4.7 Annealing window.

4.2.5 Optimization > HookeJeeves

Settings for *Hooke-Jeeves* optimization algorithm.

dV – Initial value of step (speed of changes) of trial steps

Increment Step – Constant used to increase steps in case of successful optimization in the previous iteration

Reduction Step – Constant used to reduce steps in case of unsuccessful optimization in the previous iteration

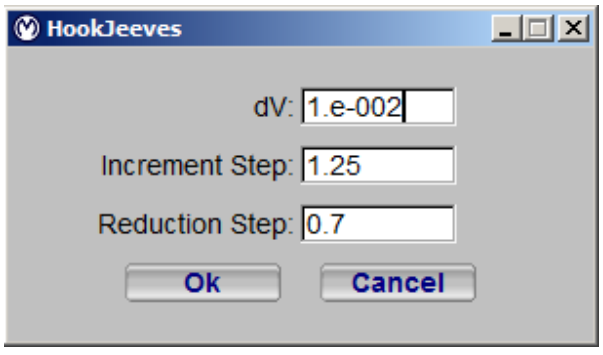

Figure 4.8 HookJeeves settings window.

4.2.6 Optimization > Powell

Settings for Powell type of optimization.

Direction Iterations – Limit of iterations for direction search loop

Alfa Step Iterations – Limit of iterations for Alfa search loop

Increment Step – Constant used to increase steps in case of successful optimization in the previous iteration

Reduction Step – Constant used to reduce steps in case of unsuccessful optimization in the previous iteration

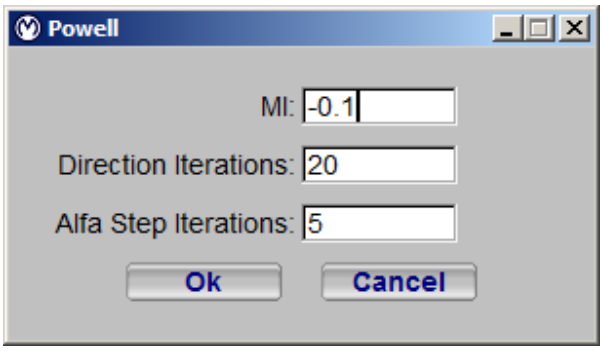

Figure 4.9 Powell settings window.

4.2.7 Optimization > NelderMead

Settings for NelderMead type of optimization.

Reflection – Constant used for reflection procedure, for more details check theory chapter

Expansion – Constant used for expansion procedure, for more details check theory chapter

Contraction – Constant used for contraction procedure, for more details check theory chapter

Reduction – Constant used for reduction procedure, for more details check theory chapter

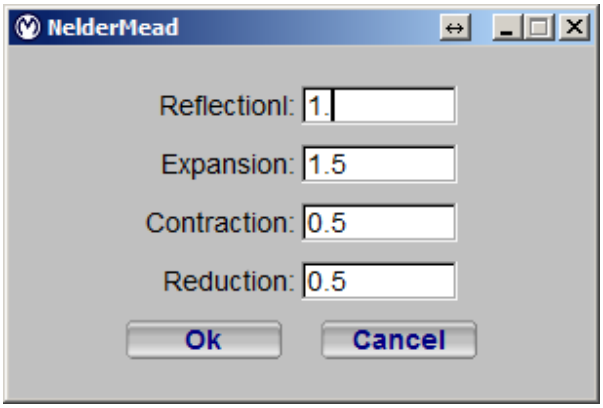

Figure 4.10 NelderMead settings window.

4.2.8 Optimization > Gradient

This part of the manual contains settings of parameters of Search Direction and estimation of Alfa step for Gradient optimization.

4.2.9 Optimization > Gradient > Direction

Window where direction solver is chosen, step and way direction derivatives are estimated. From Direction User also has to set *Eps* parameter and equation which defines how the directional derivatives are calculated.

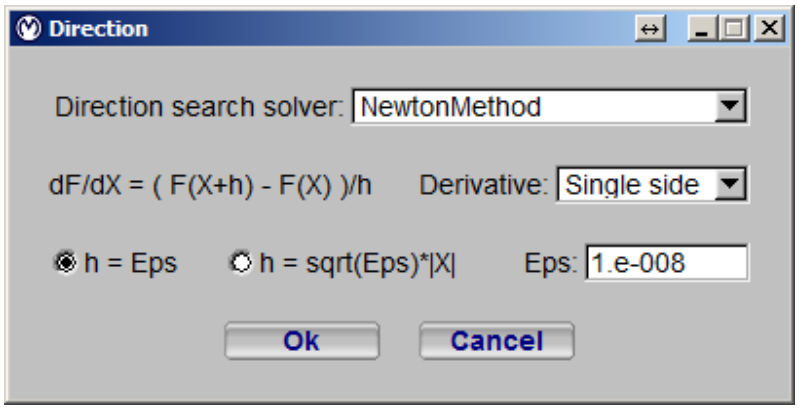

Figure 4.11 Direction window.

Direction search solver – Four methods can be chosen:
Steepest Descent, Conjugate Gradient,
Quasi Newton, Newton Method
for more details check theory chapter

Derivative – Single side or double side way of computing derivatives can be chosen.

h – Scaled perturbation to compute derivative with finite difference method.

Eps – Perturbation to compute derivative with finite difference method.

OK – approves changes and leaves the window

Cancel – leaves window without approving changes

4.2.10 Optimization > Gradient > Alfa Search

Window where one direction search parameters are set.

Alfa convexity criterion – conditions which apply size of step in estimated direction, for more details check theory chapter

Alfa iteration limit – number of attempts to find appropriate step size, after that optimization stops, any plus integer number is allowed, but it is unpractical to exceed 10

Alfa min – if during interpolation/extrapolation process Alfa is smaller then Alfa min optimization stops

Alfa low – low value of Alfa step used for interpolation/extrapolation, This value should be sufficiently low for good accuracy, but higher then Alfa min

Alfa high – Initial value of Alfa, set to one by default, it may affect ThAlfa value

c1 – coefficient for the first Alfa search condition, typically 0.1

c2 – coefficient for the second Alfa search condition, typically 0.9

u – value used for calculation of Alfa step sensitivity, similar to Eps value in Direction Search, it may affect dThAlfa value

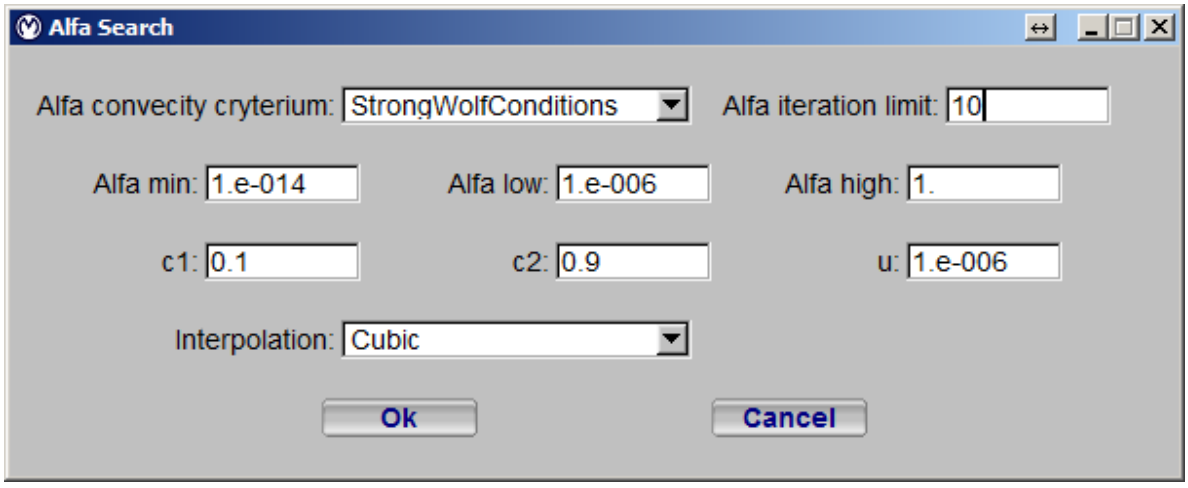

Figure 4.12 Alfa Search window.

4.2.11 Optimization > Monte Carlo

Settings for Monte Carlo type of optimization. User sets amount of samples evaluated during every iteration. Resolutions defines for how many parts each variable is divided between limits set. During every iteration limits shrink proportionally to the Radius coefficient. The bigger the Radius coefficient the more probable is finding the global optimum, but optimization takes longer.

Samples – Number of samples to compute in an iteration. Computations can be done parallel.

Resolution – Resolution with which range between Xmin and Xmax is divided

Radius – Coefficient of reduction of the area for more details check theory chapter

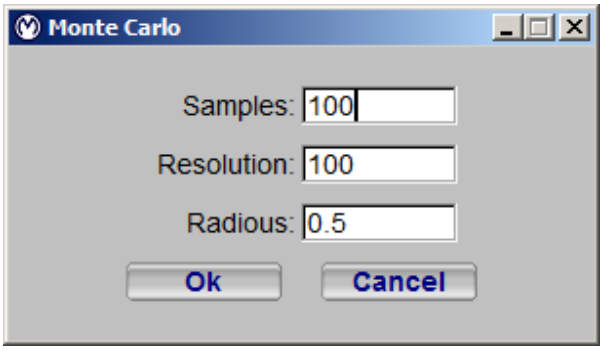

Figure 4.13 Monte Carlo window.

4.2.12 Optimization > Genetic Algorithm

This part of the manual contains settings of parameters for Genetic Algorithms.

4.2.13 Optimization > Genetic Algorithm > Genetic Selection

General settings for genetic algorithms as well as parameters for selection of individuals for reproduction strategy.

Population – Size of population.
Computations can be done parallel.

Genes length – Number of bits defining genes length

Resolution – Prints accuracy of variables, which depends on genes length

MOM Weight – Definition of Measure of Merit function,
for more details see theory chapter

Cos() Power – Additional parameter for cos(MOM) function

GA Selection Strategy – See chapter about theory

BreederPoolElit – Additional parameter for BreederPool selection strategy defining percentage of individuals that pass to next generation

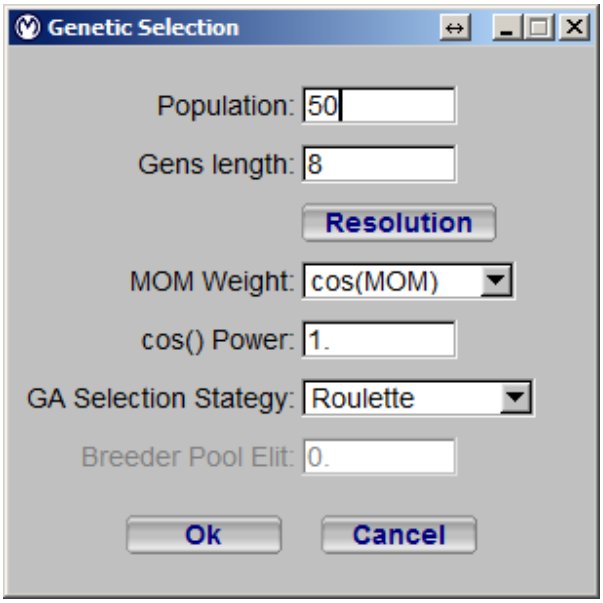

Figure 4.14 Genetic Selection window.

4.2.14 Optimization > Genetic Algorithm > Genetic Methods

Choice of methods used during reproduction.

Cross Over – Method of cross over of genomes of selected individuals, details can be found in theory guide

Mutation Factor – Mutation factor can be defined automatically or manually, in most cases automatic definition is desired, see theory guide

Mutation % – Mutation percentage defining how big part of genome is mutated

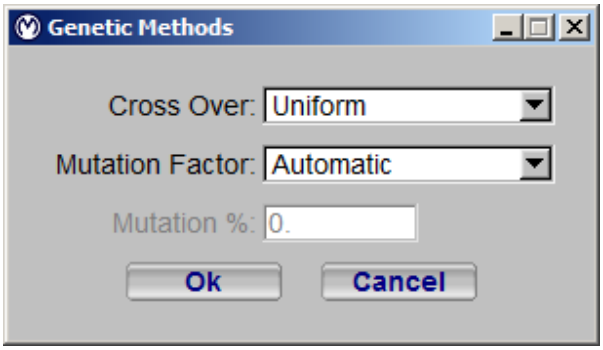

Figure 4.15 Genetic Methods window.

4.2.15 Optimization > Swarming

Here are defined setting for Swarming optimization algorithm.

Particles – Number of particles in swarm to analyze. Computations can be done parallel.

Resolution – Number of parts for which every variable is divided with in prescribed limits

w – Scaling factor of global speed vector, see theory chapter for details

s1 – Scaling factor of best particle position, see theory chapter for details

s2 – Scaling factor of globally best particle, see theory chapter for details

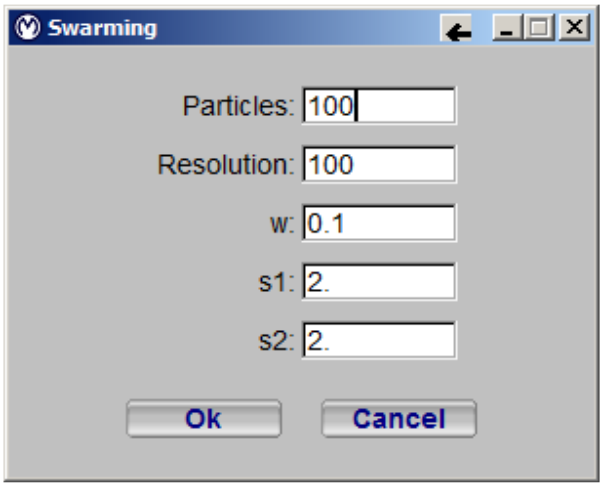

Figure 4.16 Swarming window.

4.2.16 Optimization > Stop Criterion

Defines conditions after which optimization stops. User can set the conditions, considering them as good criterion of reaching optimum. Without this constrains optimization will finish after specified number of iterations defined in initial conditions. Every line contains separate condition and delta value defining the accuracy.

OBJ_F difference – Stops if changes of objective function are smaller than delta value

Gradient sum – Stops if sum of the first derivatives is less than delta, Theoretically optimum is reached when sum of gradient is equal to zero

OBJ_F sig – Stops if standard deviation of objective function is below specified value

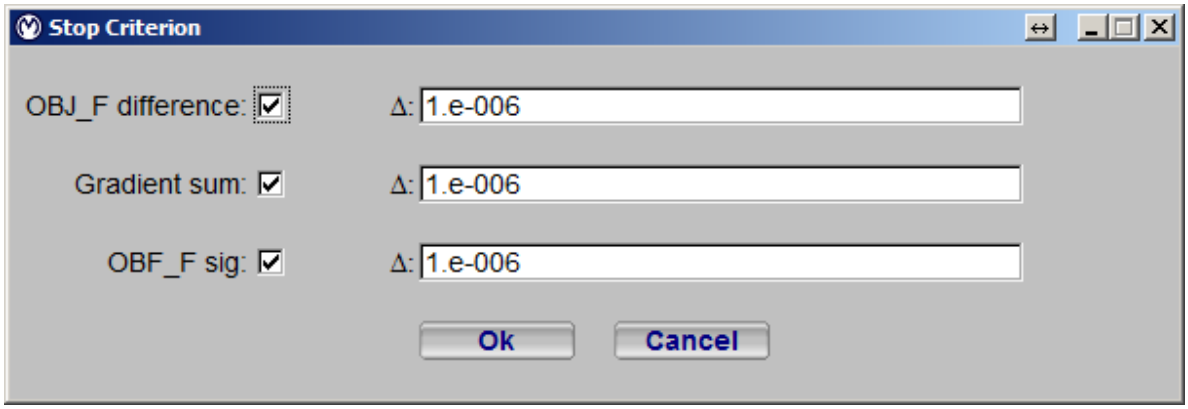

Figure 4.17 Stop Criterion window.

4.2.17 Optimization > Flags

Flags can be set to use only part of defined design variables and constrains during optimization. To activate flags check box should be turned on. Flags are defined by string of “0” and “1”. Length of string should be equal to number of design variables.

X flags – input line with flags for design variables

C flags – input line with flags for constrains

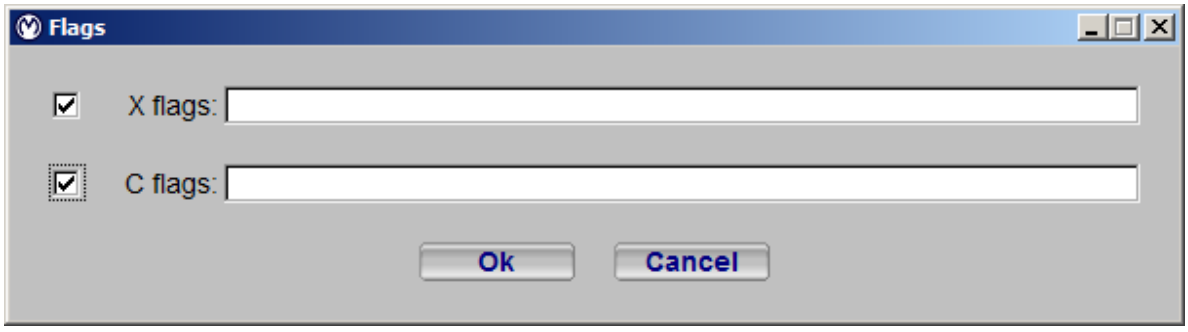

Figure 4.18 Flags window.

4.3 Optimization

4.3.1 Utilities > InitFromSol

File with definition of the optimization variables **.var* can be build based on the solutions written in the **.sol* file. User can choose iteration number from which to build the **.var* file. Minimum and maximum values will be equal. This is fine for directional optimization algorithms, but for other optimization algorithm minimum and maximum values should be different.

Number of Variables – Number of design variables used for the optimization from which results are obtained.

Init from – Iteration from which the *.var file should be build.

Sol file – Input file with results from an optimization.

Output – Output file **.var*.

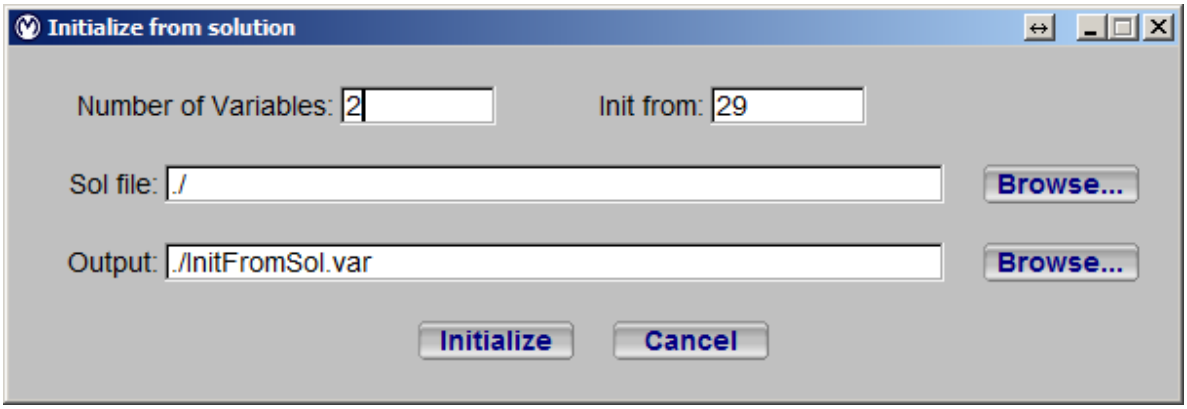

Figure 4.19 Initialize from solution window.

4.3.2 Utilities > Statistics

This tool makes simple statistical analysis for methods that generate groups of points for optimization. It shows average value (*mi*) and standard deviation (*sig*) for every variable and objective function in every iteration.

Pareto data – Input **.pareto* file

Output – Output file with statistics.

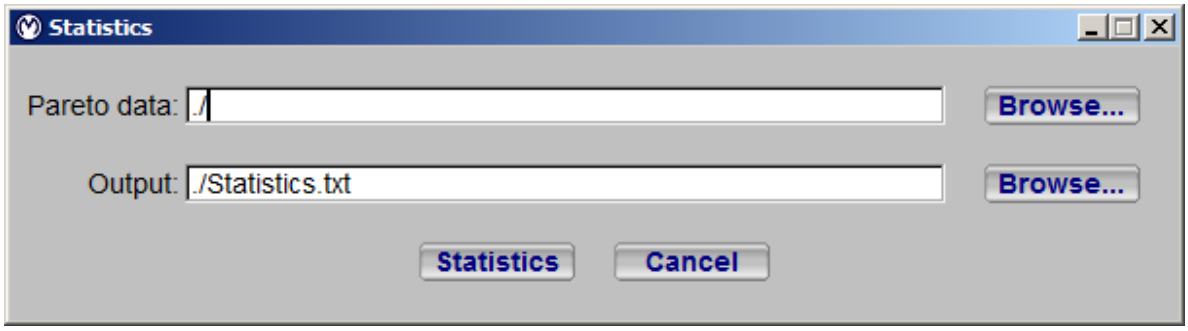

Figure 4.20 Statistics window.

4.3.3 Utilities > MinMax

Finds minimum and maximum values, in an iteration, for every variable, constrain and objective function.

Pareto data – Input **.pareto* file

Output – Output file with min/max data.

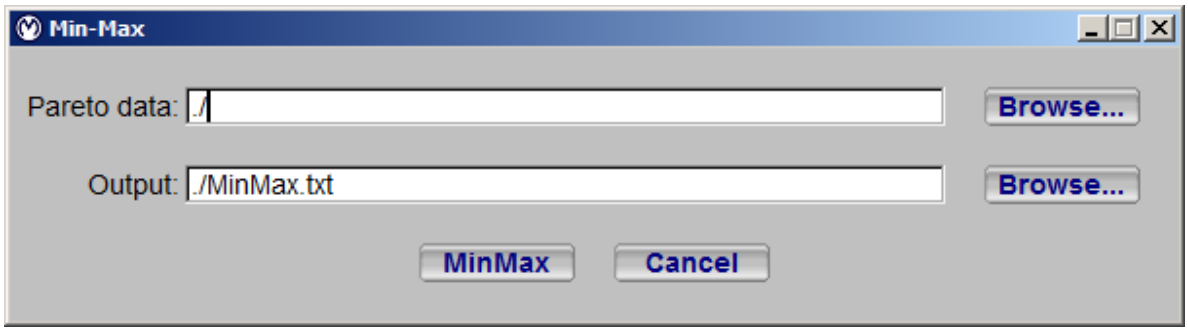

Figure 4.21 Min-Max window.

4.3.4 Utilities > Norm

Normalizes the data from solution file, by dividing it by average value.

Number of Variables – Number of design variables used for the optimization from which results are obtained.

Iteration limit – Maximum number of iterations in the solution file.

Sol file – Input file with results from an optimization.

Output – Output file with normalized solutions.

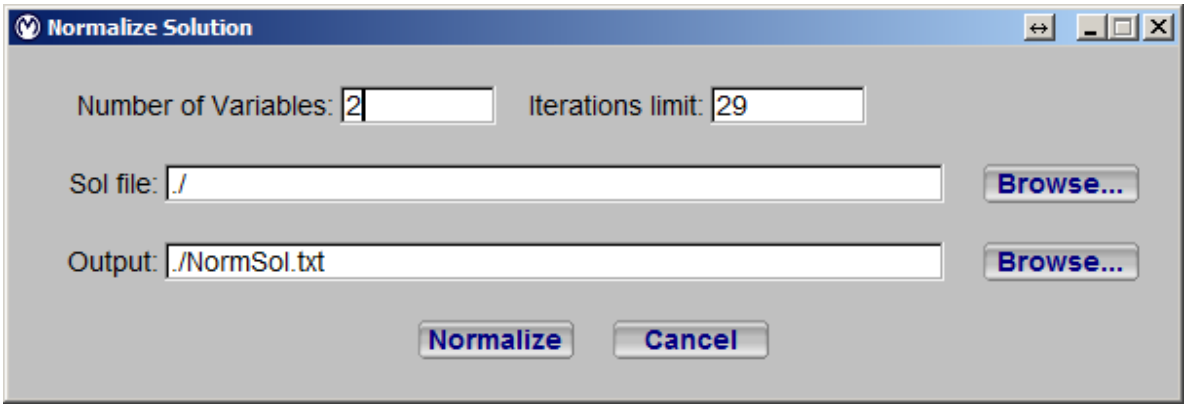

Figure 4.22 Normalize Solution window.

4.4 Plot

4.4.1 Plot > Settings

Basic settings for plotting.

Plot – Flag for displaying the plot during optimization (by default turned off in batch mode).

Legend – Flag for displaying legend on the plot.

Grid – Flag for displaying grid on the plot.

Background – Option for *White/Black* background.

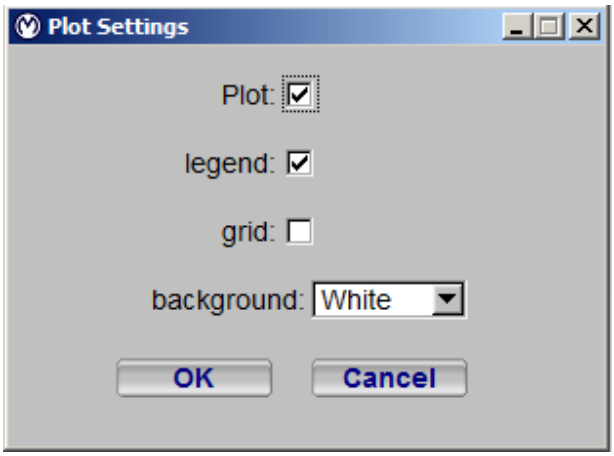

Figure 4.23 Plot settings window.

4.4.2 Plot > Post

Plotting in fast postprocessing, which is based on data from *.pareto file.

Pareto file – Pareto file with data.

Iter – Used to define which iterations should be considered. The options to choose are: *All*, *Single* + number of the iteration, *Fmin* (from all iterations for the best indywidual/sample).

X axis – Parameters for the X axis.

Y axis – Parameters for the Y axis.

Set limits – Options to set *min/max* limits for the plot's axis.

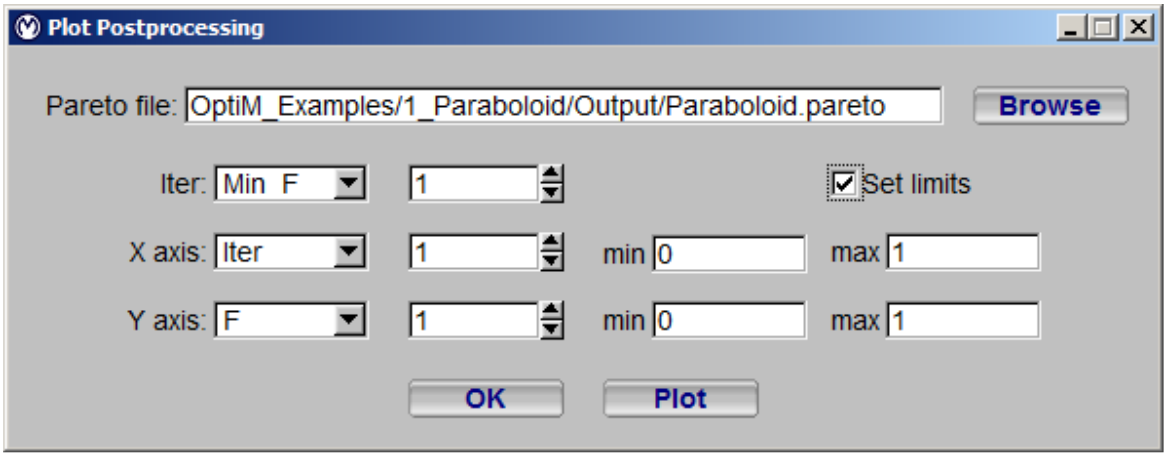

Figure 4.24 Plot Postprocessing window.

4.4.3 Plot > Popup menu

After plot is displayed it can be printed or saved as a picture. Options are available under mouse right button.

Print – Standard printing menu to print the plot.

Save PNG – Saves the plot as png picture.

4.5 Help

4.5.1 Help > Manual

Opens Optim's manual.

4.5.2 Help > License

License of the OptiM.

4.5.3 Help > About

Short information about the OptiM version.

5. VAREDIT USER GUIDE

VarEdit is a simple GUI *Figure 5.1* for definition of input variables, constrains and output options for the OptiM. The data is saved in plain text format, with *.var file extension, and can be edited manually. Tables for variables, constrains and objective functions are limited to 1000 rows. OptiM doesn't have such a limitation. If more input data for OptiM is needed, it should be generated without using VarEdit tool. Program can be also opened with single parameter, which is the configuration file.

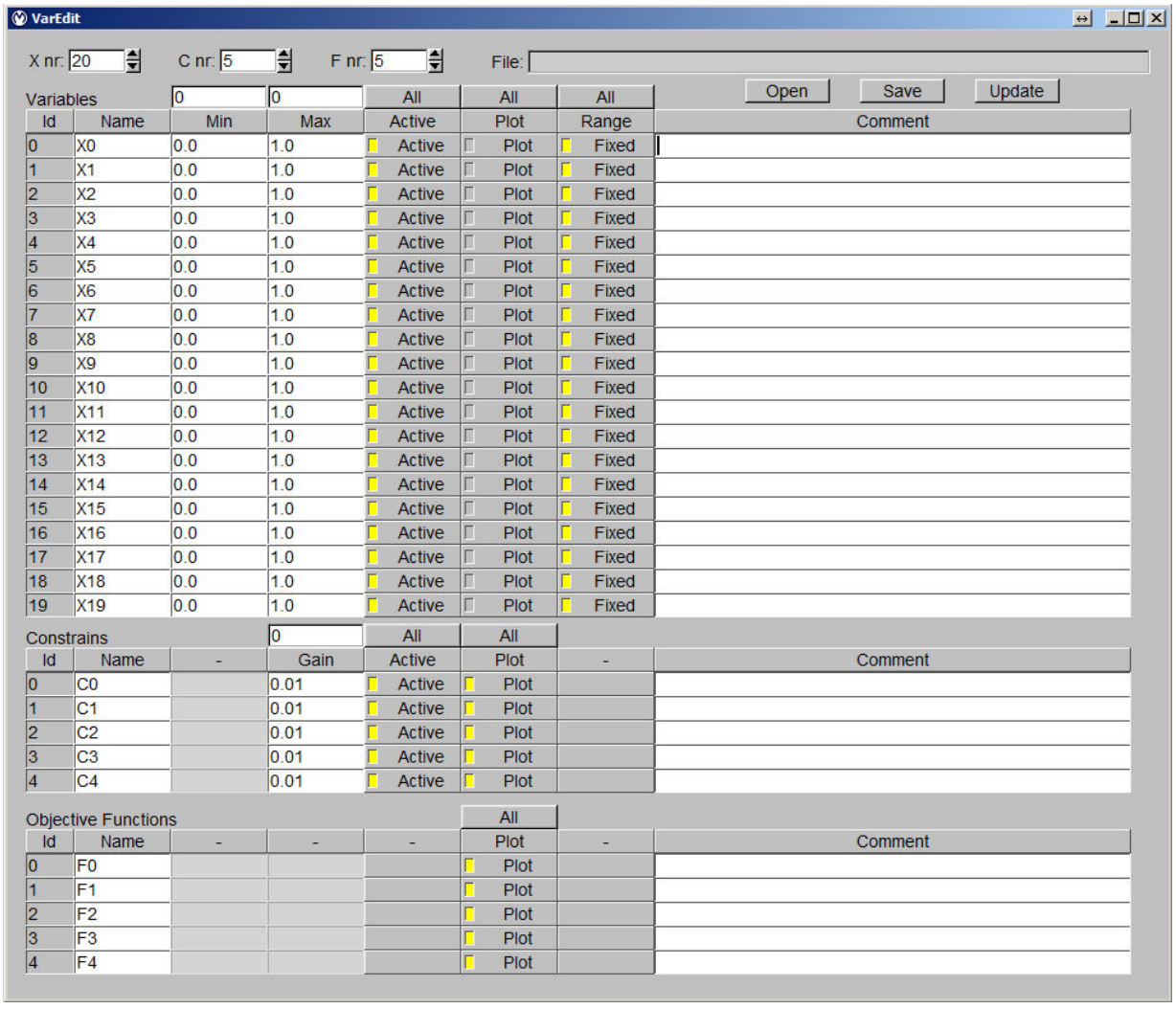

Figure 5.1 VarEdit window.

File – The configuration file user works on. It can be opened, saved and updated if file path is specified.

X nr – Number of optimization variables.

C nr – Number of constrains.

F nr – Number of monitored objective functions.

5.1 Variables

Fields and buttons on the top of the columns set values in all rows for the particular column. For algorithm which start optimization from single design point, for example gradient method, values of variables are calculated as average from Min and Max values. Same value can be specified for Min and Max.

Id – Identification number, user refers to it in tables in the dynamic library source code.

Name – Name of the variable showed in results, logs, plots.

Min – Minimum value of the optimization variable.

Max – Maximum value of the optimization variable.

Active – Flag sets varying, or constant value of the design variable.

Plot – Flag sets to plot, or not the variable during optimization.

Range – Few algorithms allow for crossing specified Min, Max values of variables. Flag sets if the boundaries are fixed, or not.

Comment – Optional user comment to the variable, used only in VarEdit.

5.2 Constrains

Fields and buttons on the top of the columns set values in all rows for the particular column.

Id – Identification number, user refers to it in tables in the dynamic library source code.

Name – Name of the constrain showed in results, logs, plots.

Gain – Gain of the constrain, which specifies it's strength.

Active – Flag sets varying, or constant value of the constrain.

Plot – Flag sets to plot, or not the constrain during optimization.

Comment – Optional user comment to the constrain, used only in VarEdit.

5.3 Objective Functions

Id – Identification number, user refers to it in tables in the dynamic library source code.

Name – Name of the objective function showed in results, logs, plots.

Plot – Flag sets to plot, or not the objective function during optimization.

Comment – Optional user comment to the monitored objective function, used only in VarEdit

6. NUMERICAL UTILITIS

These part of the manual describes additional functions that can be used in the objective function. Many of the functions apply to matrix algebra. The functions should help to work efficiently while defining the optimization task.

6.1 Delete File

```
int DelFile(char *FileName);

FileName    - path with name to be delated

example: DelFile("AnalysisResults.txt");
```

It simply deletes file without any warnings. It can be used to delete files from previous analysis during the iterative optimization process.

6.2 Pipe

```
int Pipe(char *Program, char *input, char *output, char *buffer);

Program     – path with name of program/script for analysis
input       - file containing command passed with pipe
output      - optional output from analysis program passed from screen to file
buffer      - string containing commands for analysis program

example: Pipe("Xfoil.exe", "",  "Xfoil_Log.txt", buf);
```

It creates C pipe to program which makes analysis. The program has to have ability to work in a console mode. The pipe can run script written in system shell (Windows also), which contains user defined command for analysis process. OptiM waits until pipe executes all commands and than proceeds with optimization. Commands for analysis program can be passed by string defined in objective function or by indicating file with commands. If user does not want to pass one of the arguments than leave nothing between apostrophes. The parameters are concatenated to form script command: Program < input > output. Thirst commands from input file will be executed than from the buffer.

6.3 Error

```
int Error(char *Message);

Message     - Text of message that will be displayed

example: Error("OBJ_F could not be executed");
```

This function can be used to display message if error occurred. After that OptiM closes.

6.4 Penalty

```
double Penalty(double mi, double &C, double Left, char sign, double Right);

mi          - parameter for penalty function constrain (see theory guide)
C           - values of penalty after border crossing passed to output files
Left        - left side of constrain inequality
sign        - sign of inequality, three options: "<" "=" ">"
Right       - right side of constrain inequality

example: Penalty(Par.mi, C[0],  X[1]*X[2], "<", 4);
```

The function helps user to easily put penalty barrier constrains on the design. *mi* parameter can be changed in GUI, to pass it write *Par->mi*. *C* returns value of selected constrain to constrains table, write *C[number of constrain].* Next is the definition of constrain inequality.

6.5 Algebra formulas

```
int Gauss(int n, double **A, double *X, double *D);
int Gauss_Jordan(int n, double **A, double *D);

n           - number of variables (size of matrix)
A           - pointer to A matrix
X           - vector X
D           - vector D

example: Gauss(n, A, X, D);
example: Gauss_Jordan(n, A, D);
```

Similar methods of solving system of equations: **[A]*{X} = {D}**. Very popular methods, efficient when solving small matrices *n* = ~3000. Matrix *A* and vector *D* is known. In Gauss method vector *X* contains the solution, in Gauss Jordan method vector *D*.

```
int Scale_A(int n, double *X, double *dF, double **H);

n           - number of variables (size of matrix)
X           - vector
dF          - vector
H           - matrix

example: Scale_A(n, X, dF, H);
```

Scales the matrix.

```
int x_yT(int n, double *x, double *y, double **A);

n           - size of vectors
x           - thirst vector
y           - second vector
A           - matrix with results
```

```
example: x_yT(n, x, y, A);
```

It solves: **[A] = {x}*{y}^{T}**

```
int A_x(int n, double *C, double **dH, double *Xo);

n           - size of
C           - vector with results
dH          - matrix
Xo          - vector

example: A_x(n, C, dH, Xo);
```

It solves: **{C} = [dH]*{x}**

```
double detA(int n, double **A);

n           - size of matrix
A           - the matrix

example: Result = detA(n, A);
```

It solves: **Result = det|A|**

```
double xT_A_y(int n, double *x, double **A, double *y, double **C);

n           - size of matrix
x           - thirst vector
A           - matrix
y           - second vector
C           - additional empty matrix

example: Result = xT_A_y(n, x, A, y, C);
```

It solves: **Result = {x}^{T}*[A]*{y}**

```
double xT_y(int n, double *x, double *y);

n           - size of vectors
x           - thirst pointer to vector
y           - second pointer to vector

example: Result = xT_y(n, x, y);
```

It solves: **Result = {x}^{T}*{y}**

7. EXAMPLES

Installation of the OptiM contains few examples of projects with definition of optimization tasks. Typical structure of the catalogues Fig. 6.1 in the project consists of sub-catalogs *Input, Lib, Output* and the project file for OptiM **.om*. Catalogue Input contains files with the design variables. Structure of the file is self-explanatory. First is the version number of the file **.var*. Next is a header line explaining what contain the columns: *Id* of the variable (the same number should be used in the dynamic library for the table with the optimization variables *X[Id]*), name of the optimization variable, minimum value of the optimization variable, maximum value of the optimization variable, optional comment. Catalogue *Lib* contains source code of the dynamic library with the optimization task. Catalogue *Output* is the output directory for computed data and logs from the optimization process.

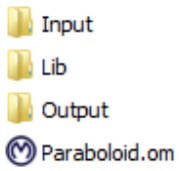

Figure 7.1 OptiM project catalogues structure.

Normally to run the optimization problem user has to compile the dynamic library for OptiM with defined optimization task, as described briefly in chapter 3.2. The examples are already precompiled and can be optimized immediately. Examples should be read in through *Menu* > *Parameters* > *Load* and selecting *Project.om* file. Before running optimization on the examples, working directory of a project has to be set in *Menu* > *File paths*. Also file path to the dynamic library from examples might have wrong ending due to the operating system. Ensure that for Windows dynamic library ends with *.dll*, and for Unix systems *.so*. If the working directory won't be set, or will be set incorrectly warning messages will appear and correction will be needed.

7.1 EXAMPLE – PARABOLOID

This is very simple example with mathematical function of paraboloid, with shifted minimum point to [-30, 20]. The paraboloid is described by equation (6.1). Analytical optimum is described by (6.2).

$$f_{obj} = f_0 = (x_0 + 30)^2 + (x_1 - 20)^2 \tag{6.1}$$

$$f_{\min} = 0 \qquad \text{for} \qquad x_0 = -30 \qquad x_1 = 20 \tag{6.2}$$

7.2 EXAMPLE – ROSENBROCK FUNCTION

This example is based on the general Rosenbrock function (6.3), called also Rosenbrock valley, or Rosenbrock banana function. It is often used as a bench mark for performance of optimization algorithms. To get to the optimum point directional algorithms have to change the direction of search. Path of the minimum points lies at the bottom of the valley. Sides of the valley are very steep and the path of minimum points descends very slowly. It is easy to get on the path, but hard to follow it. Part of the algorithm responsible for the direction estimation has to be very effective.

In the example typical constants are assumed and have values of *a=100*, *b=1* creating specific Rosenbrock function (6.4). Equation (6.4) is also assigned to f_0 function, which can be printed and plotted during optimization. Additional functions for printing and plotting f_1 (6.5) and f_2 (6.6) are defined, which help to understand the influence of the separate parts of the Rosenbrock equation. Analytically solution of the minimum of the function (6.4) is presented as (6.7).

$$f_{obj} = a \cdot \left(x_1 - {x_0}^2\right)^2 + \left(b - x_0\right)^2 \tag{6.3}$$

$$f_{obj} = f_0 = 100 \cdot \left(x_1 - {x_0}^2\right)^2 + \left(1 - x_0\right)^2 \tag{6.4}$$

$$f_1 = 100 \cdot \left(x_1 - {x_0}^2\right)^2 \tag{6.5}$$

$$f_2 = \left(1 - x_0\right)^2 \tag{6.6}$$

$$f_{\min} = 0 \qquad \text{for} \qquad x_0 = 1 \qquad x_1 = 1 \tag{6.7}$$

7.3 EXAMPLE – ROSENROCK MULTIDIMENSIONAL

The example is an extension of the previous one. The Rosenbrock function has multiple optimization variables and can be described by the equation (6.8). After setting the constants to *a=100*, *b=1* function (6.9) is obtained. Because of the multiple variables solving the problem for minimum becomes much harder. Function (6.9) has one global minimum for *n=3* (6.10) and additional local minimum for *n=4* to *n=7* (6.11).

$$f_{obj} = \sum_{i=1}^{n-1} \left[a \cdot \left(x_{i+1} - {x_i}^2\right)^2 + \left(b - x_i\right)^2 \right] \tag{6.8}$$

$$f_{obj} = f_0 = \sum_{i=1}^{n-1} \left[100 \cdot \left(x_{i+1} - {x_i}^2\right)^2 + \left(1 - x_i\right)^2 \right] \tag{6.9}$$

$$f_{\min} = 0 \qquad \text{for} \qquad n = 3 \qquad x_i = 1 \tag{6.10}$$

$$f_{\min\_local} = 0 \qquad \text{for} \qquad 4 \le n \le 7 \qquad x_0 \approx -1 \qquad x_i \approx 1 \tag{6.11}$$

7.4 EXAMPLE – RASTRIGIN FUNCTION

The Rastrigin function (6.12) is also often used as a bench mark for optimization algorithms. The function has number of local minimums. Directional algorithms will quickly get caught in one of the local minimums. Only heuristic algorithms, with global field of search through the design variables, will be able to find the global optimum.

In the example two dimensional function is considered and *A=10*, which produces particular function (6.13). Analytical solution for minimum is (6.14).

$$f_{obj} = A \cdot n + \sum_{i=1}^{n} \left[x_i^{\,2} - A \cdot \cos(2 \cdot \pi \cdot x_i) \right] \tag{6.12}$$

$$f_{obj} = f_0 = 20 + \sum_{i=1}^{2} \left[x_i^{\,2} - 10 \cdot \cos(2 \cdot \pi \cdot x_i) \right] \tag{6.13}$$

$$f_{\min} = 0 \qquad \text{for} \qquad x_0 = 0 \qquad x_1 = 0 \tag{6.14}$$

The example shows the simplest possible way to execute external program for analysis. Except of the dynamic library, also simple program is compiled with the Rastrigin function, which will be computed inside. Normally the program will be dedicated software (often commercial) for sophisticated analysis, which runs in a butch mode.

WARNING! for people with epilepsy! Analyzed program is executed multiple times and blinks fast on screen.

Optimization can be run parallel. In that case user has to remember to provide separate names of configuration files for every thread. Otherwise parameters will be read and written to the same files, which will result in an unexpected behavior.

7.5 EXAMPLE – WING OPTIMIZATION

The example shows, practical optimization problem of the airplane's wing, with electric propulsion. Vertical forces: lift force and weight of an aircraft (6.15) and horizontal forces drag and thrust force (6.16) should be equal. The equation (6.16) can be also viewed in terms of power balance. After derivation of flight velocity (6.17) from equation (6.15) it can be put to equation (6.18) for power balance. Making the numerical

model closer to real flight conditions power consumption of on board electronics and efficiency coefficients have to be incorporated (6.19).

$$m \cdot g = \frac{1}{2} \cdot \rho \cdot V^2 \cdot C_L \cdot S \tag{6.15}$$

$$T = \frac{1}{2} \cdot \rho \cdot V^2 \cdot C_D \cdot S \tag{6.16}$$

$$V = \sqrt{\frac{2 \cdot m \cdot g}{\rho \cdot C_L \cdot S}} \tag{6.17}$$

$$P_{flight} = T \cdot V = T \cdot \sqrt{\frac{2 \cdot m \cdot g}{\rho \cdot CL \cdot S}} \tag{6.18}$$

$$P_{total} = \frac{P_{flight}}{\eta_{esc} \cdot \eta_{mot} \cdot \eta_{grb} \cdot \eta_{prop}} + \frac{P_{avio} + P_{payload}}{\eta_{bec}} \tag{6.19}$$

More power needed for extended time of flight means bigger butteries for power supply (6.20). Increased mass of butteries forces heavier aircraft structure and propulsion system. The mass of the structure for small aircrafts can be approximated with good accuracy with equation (6.21), dependent on wingspan and wing aspect ratio. Mass of propulsion system depends mainly from power needed for flight and can be approximated with equation (6.22).

$$m_{batt} = \frac{P_{total} \cdot t}{\eta_{dchrg} \cdot k_{batt}} \tag{6.20}$$

$$m_{struct} = 0.44 \cdot b^{3.1} \cdot AR^{-0.25} \tag{6.21}$$

$$m_{propusion} = k_{propulsion} \cdot P_{flight} \tag{6.22}$$

Friction drag is corrected for changing low Reynolds numbers, approximately below *Re* ~500000, which is often the case of small electric UAV aircrafts. The second improvement is dropped simplification for small climb angles, which is often not true for small electric aircrafts with proportionally big motors.

APPENDIX A - LIST OF OPTIM KEY SHORTCUTS

File

Parameters	Ctrl + P
Data Format	Ctrl + D
Parallel	Ctrl + L

Optimization

File paths	Ctrl + O
Initialize	Ctrl + I
Solver	
Tester	T
Annealing	1
HookJeeves	2
Powell	3
NelderMead	4
Gradient	5
Monte Carlo	6
Genetic	7
Swarming	8
Annealing	Shift + Ctrl + A
HookJeeves	Shift + Ctrl + J
Powell	Shift + Ctrl + W
NelderMead	Shift + Ctrl + N
Gradient	
Direction	Shift + Ctrl + D
Alfa Search	Shift + Ctrl + F
Monte Carlo	Shift + Ctrl + M
Genetic Algorithm	
Genetic Selection	Shift + Ctrl + G
Genetic Methods	Shift + Ctrl + E
Swarming	Shift + Ctrl + S
Stop Criterion	Ctrl + C
Flags	Ctrl + F

Help

Manual	Ctrl + H