2012 American Control Conference
Fairmont Queen Elizabeth, Montréal, Canada
June 27-June 29, 2012

A Generic Particle Swarm Optimization Matlab Function

Soren Ebbesen, Pascal Kiwitz and Lino Guzzella

Abstract— **Particle swarm optimization (PSO) is rapidly gaining popularity but an official implementation of the PSO algorithm in MATLAB is yet to be released. In this paper, we present a generic particle swarm optimization MATLAB function. The syntax necessary to interface the function is practically identical to that of existing MATLAB functions such as `fmincon` and `ga`. We demonstrate our PSO function by means of two examples: the first example is an academic test problem; the second example is a simplified problem of optimizing the gear ratios in a hybrid electric drivetrain. The PSO function is available online.**

I. INTRODUCTION

Particle Swarm Optimization is a numerical search algorithm which is used to find parameters that minimize a given objective, or *fitness* function. The PSO algorithm was introduced less than two decades ago by Kennedy and Eberhart [1]. Over the years, PSO has gained significant popularity due to its simple structure and high performance. The fitness function can be non-linear and can be subjected to any number of linear and non-linear constraints. Numerous publications, such as [2]–[6] to name a few, demonstrate the merit of PSO in a diverse range of applications.

MATLAB by MathWorks, Inc. is widely used software for numerical computing. While MATLAB makes several algorithms for numerical optimization available, the PSO algorithm is yet to be included. In 2003, Birge [7] published a PSO function for MATLAB. In addition, a few other more recent implementations of the PSO algorithm are available online from the MATLAB file-exchange server[1], for example the *PSO Research Toolbox* by Evers. Yet, none of those were accompanied by published documentation.

The aforementioned PSO function by Birge is a simple yet capable implementation. Unfortunately, its syntax differs significantly from standard MATLAB optimization functions such as `fmincon` and `ga`. Thus, switching between the different search methods is somewhat involved. In addition, while the PSO algorithm is very well suited for parallelization, the function does not easily allow to deploy the algorithm on a MATLAB Distributed Computing Server.

In this paper, we introduce a generic PSO function for MATLAB. The syntax used to interface the function is practically identical to that of `fmincon` and `ga`. In addition, it is possible to specify whether the fitness function is *vectorized*. In this context, vectorized means that the function accepts a vector of input values and will return a vector of corresponding output values rather than calling the function with one input at the time. This can reduce the runtime significantly and enables the algorithm to run on a computer cluster. Other useful functionalities include the possibility to specify a hybrid function, such as `fmincon` or `fminsearch`, which automatically continues the optimization after the PSO algorithm terminates. Moreover, it is possible to specify user-defined output and plotting functions that are called periodically while the algorithm is running. Furthermore, our PSO function enables the user to recover a swarm from an intermediate state, rather than restarting the optimization, in case the function terminated prematurely due to some extraordinary event. Finally, for simplicity, the function and all necessary sub-functions are contained in a single file.

This paper is organized in the following way: in Section II the PSO algorithm is introduced; in Section III we present the syntax and commands necessary to interface the generic PSO MATLAB function; in Section IV we demonstrate the function using two examples; finally, in Section V we state our conclusions.

Soren Ebbesen, Pascal Kiwitz and Lino Guzzella are with the Institute for Dynamic Systems and Control ETH Zurich, 8092 Zurich, Switzerland sebbesen@idsc.mavt.ethz.ch, pascal.kiwitz@idsc.mavt.ethz.ch, lguzzella@ethz.ch

[1] www.mathworks.com/matlabcentral/fileexchange

II. PARTICLE SWARM OPTIMIZATION

Particle swarm optimization is a stochastic search method inspired by the coordinated motion of animals living in groups. The change in direction and velocity of each individual particle is the effect of cognitive, social and stochastic influences. The common goal of all group members is to find the most favorable location within a specified search space. Mathematically speaking, particle swarm optimization can be used to solve optimization problems of the form

$$\min_x : \quad f(x) \tag{1}$$

$$\begin{aligned} \text{subject to} : \quad A \cdot x &\leq b \\ A_{eq} \cdot x &= b_{eq} \\ c(x) &\leq 0 \\ c_{eq}(x) &= 0 \\ lb \leq x &\leq ub \end{aligned} \tag{2}$$

where x, b, b_{eq}, lb, and ub are vectors, and A and A_{eq} are matrices. The functions $f(x)$, $c(x)$, and $c_{eq}(x)$ can be nonlinear functions. The fitness function $f(x)$ quantifies the performance of x.

A. Algorithm

Over the years, several modifications to the original PSO algorithm have been suggested. We adopted the following

intuitive formulation

$$v_i^{k+1} = \phi^k v_i^k + \alpha_1 \left[\gamma_{1,i}\left(P_i - x_i^k\right)\right] + \alpha_2 \left[\gamma_{2,i}\left(G - x_i^k\right)\right] \quad (3)$$

$$x_i^{k+1} = x_i^k + v_i^{k+1}. \quad (4)$$

The vectors x_i^k and v_i^k are the current position and velocity of the i-th particle in the k-th generation. The swarm consists of N particles, i.e., $i = \{1,\ldots,N\}$. Furthermore, P_i is the personal best position of each individual and G is the global best position observed among all particles up to the current generation. The parameters $\gamma_{1,2} \in [0,1]$ are uniformly distributed random values and $\alpha_{1,2}$ are acceleration constants. The function ϕ is the particle inertia which gives rise to a certain momentum of the particles.

Figure 1 shows a graphical interpretation of the PSO algorithm in a two-dimensional space: the new velocity v_i^{k+1} is the sum of a momentum that tends to keep the particle on its current path; an attraction towards its personal best position P; and finally, an attraction towards the global best position of all group members G. Finally, the new position x^{k+1} is the sum of the current position x^k and the velocity v^{k+1}.

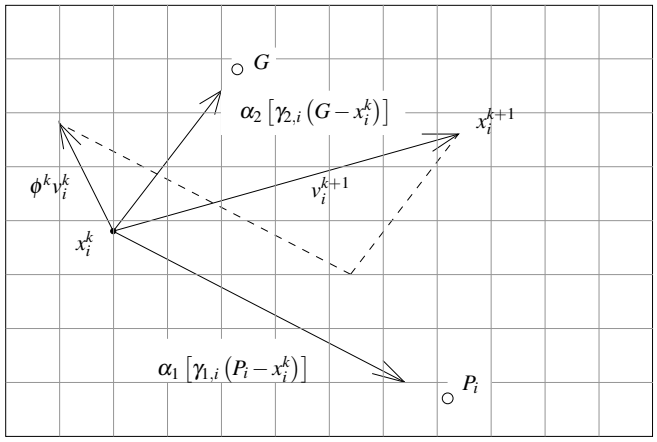

Fig. 1. Graphical interpretation of the PSO algorithm.

B. Stability

Necessary and sufficient conditions for stability of the swarm were derived in [8]. The conditions are

$$\alpha_1 + \alpha_2 < 4 \quad (5)$$

and

$$\frac{\alpha_1 + \alpha_2}{2} - 1 < \phi < 1. \quad (6)$$

These conditions guarantee convergence to a stable equilibrium. However, there is no guarantee that the proposed solution is the global optimum.

C. Constraints

There are several ways of dealing with the constraints (2) in particle swarm optimization. Some methods are demonstrated in [9], [10]. Three distinct constraint handling methods are implemented in the current PSO function. These we refer to as *penalize*, *absorb*, and *nearest*.

The first method penalizes particles violating a constraint by assigning a high objective function value to the particle. This value must be higher than the highest value attainable within the feasible region of the search space. Consequently, particles are free to move across the constraints, yet they will remain attracted to the feasible region of the search space which they will eventually re-enter.

The second method absorbs the particles on the boundary of the feasible region, defined by the constraints, if the particle would otherwise be moving across. In contrast to the former method, the fitness value of the particles is evaluated. This method may yield better results than penalizing, assuming the global minimum is located on or near a constraint.

The last method retracts the particle from beyond the violated constraints and places the particle on the nearest constraint. Identically to *absorb*, the fitness function is evaluated.

Figure 2 illustrates all three methods. The set Ω is the feasible region of the search space. The position x^{k+1} and velocity v^{k+1} are the final position and velocity, according to the PSO algorithm, respectively. The white circles indicate the final position after the chosen constraint handling method has been applied. The velocity of the particle is altered accordingly.

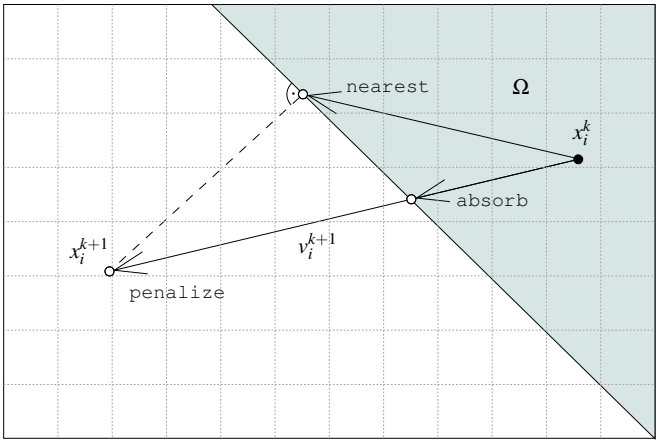

Fig. 2. Graphical interpretation of the constraint handling methods.

III. GENERIC PSO MATLAB FUNCTION

The generic `pso` MATLAB function presented herein is an implementation of the PSO algorithm introduced in Section II. It can be used to solve optimization problems in the form of (1) and (2). In this section, the syntax and commands needed to solve such problems using the `pso` function are demonstrated. That said, the syntax and commands of the `pso` function are largely identical to those of the MATLAB generic algorithm function `ga`. Thus, `pso` can be applied with little effort, in particular if a `ga`-based optimization routine is already in place. For the same reason, this section is limited to explain those points that are unique to the `pso` function. For details on shared functionality, we recommend

TABLE I
INPUT ARGUMENTS.

fitnessfcn:	Function handle of fitness function
nvars:	Number of design variables
Aineq:	A matrix for linear inequality constraints
bineq:	b vector for linear inequality constraints
Aeq:	A matrix for linear equality constraints
beq:	b vector for linear equality constraints
lb:	Lower bound on x
ub:	Upper bound on x
nonlcon:	Function handle of nonlinear constraint function
options:	Options structure created by calling pso with no inputs and a single output

TABLE II
OUTPUT ARGUMENTS.

x:	Variables minimizing fitness function
fval:	The value of the fitness function at x
exitflag:	Integer identifying the reason the algorithm terminated
output:	Structure containing output from each generation and other information about the performance of the algorithm

the official documentation of ga which is openly available at http://www.mathworks.com.

The pso function is normally called using the following syntax

```
>> [x,fval,exitflag,output] = pso(fitnessfcn,...
          nvars,A,b,Aeq,Beq,lb,ub,nonlcon,options)
```

Users familiar with ga or fmincon will recognize this syntax. In fact, the only differences in syntax between those functions and pso are in the options structure. For convenience, the role of each input and output argument is summarized in Table I and II, respectively. Alternatively, pso may be called using the syntax

```
>> [x,fval,exitflag,output] = pso(problem)
```

where problem is a structure containing the input arguments of Table I.

A. Options

The options structure controls the behavior of the PSO function. The options available to pso can be inspected at any time by calling the function with no input nor output arguments, that is

```
>> pso
```

This will generate an output in the command window indicating all available options, their default values, and their class, e.g., matrix, scalar, function handle, etc.

The default options structure is generated by calling pso without input arguments but with a single output argument, that is

```
>> options = pso
```

TABLE III
OPTIONS-STRUCTURE (OPTIONS).

PopInitRange:	Range of random initial population
PopulationSize:	Number of particles in swarm
Generations:	Maximum number of generations
TimeLimit:	Maximum time before pso terminates
FitnessLimit:	Fitness value at which pso terminates
StallGenLimit:	Terminate if fitness value changes less than TolFun over StallGenLimit
StallTimeLimit:	Terminate if fitness value changes less than TolFun over StallTimeLimit
TolFun:	Tolerance on fitness value
TolCon:	Acceptable constraint violation
HybridFcn:	Function called after pso terminates
Display:	Display output in command window
OutputFcns:	User specified output function called after each generation
PlotFcns:	User specified plot function called after each generation
Vectorized	Specify whether fitness function is vectorized
InitialPopulation:	Initial position of particles
InitialVelocities:	Initial velocities of particles
InitialGeneration:	Initial generation number
PopInitBest:	Initial personal best of particles
CognitiveAttraction:	Attraction towards personal best
SocialAttraction:	Attraction towards global best
VelocityLimit:	Limit absolute velocity of particles
BoundaryMethod:	Set method of enforcing constraints

Table III shows the available options. Notice that all but the last eight options are identical to those found in the ga options structure. The eight unique options are introduced in detail below.

The option InitialPopulation is used to specify the exact initial position of one or more particles. This is practical if knowledge of one or more possible locations of the global optimum is available. In addition, this option together with the options InitialVelocities, InitialGeneration, and PopInitBest may be used to recover the algorithm in a certain state. This is useful if the algorithm terminated prematurely due to some extra-ordinary event. To ensure the swarm can be recovered, the state of the swarm must be recorded after each generation. This can be done by appropriately defining an output function in which the current state is saved.

The two options CognitiveAttraction and SocialAttraction correspond to the parameters α_1 and α_2 of Eq. (3), respectively. Thus, they are used to specify the attraction of the particles towards their personal best position and the global best position of the entire swarm, respectively. A warning is issued if the values of these two constants do not respect the conditions for

stability (6). In general, it is recommended to set the social attraction larger than the cognitive attraction.

The option `VelocityLimit` may be used to limit the absolute velocity of the particles in either direction of the search space. If no limits are imposed, a certain degree of oscillation may occur where the particles are bouncing back and forth between the boundaries of the search space. This causes slow convergence. On the other hand, a too stringent velocity limit may also lead to slow convergence because the particles need more time to move across the search space.

Finally, the option `BoundaryMethod` specifies how particles violating one or more constraints are handled (cf. Section II-C). If set to `'penalize'`, then the fitness value of particles violating one or more constraints is assigned `realmax` (largest positive floating-point number). In contrast, if `'absorb'`, then the particles will be absorbed on the constraint rather than crossing it. The exact position of the particle will be the intersection between the constraint and the straight line between the old and new positions. The method uses a bi-section algorithm. The last method `'nearest'` places the particles on the nearest constraint. This is done internally using either linear or sequential quadratic programming depending on the type of constraint. The latter two methods are computationally more demanding than penalizing because the position of the particle on the constraint is computed. However, they may yield better results if the global optimum is on or near a constraint.

B. Output and Plotting Functions

The options `OutputFcns` and `PlotFcns` can be cell-arrays of function-handles to user-defined functions. These functions are called once during initialization, once at the end of each generation, and once after the particle swarm algorithm terminates. The functions are called automatically from within the `pso` function using the syntax

```
state = function_handle(options,state,flag)
```

The input argument `flag` is either `'init'`,`'iter'` or `'done'` indicating the point at which the function is called. The argument `state` is a structure containing information about the current state of the swarm such as generation number, particle positions, velocities, personal bests, global best, and a stop-flag. The PSO algorithm is interrupted if `state.StopFlag = true`. The stop-flag is `false` per default.

C. Particle Inertia Function

The particle inertia function ϕ^k of (3) is typically defined as a linearly decreasing function of k, that is

$$\phi^k = \frac{\phi_b - \phi_a}{K} \cdot (k-1) + \phi_a \quad \text{for} \quad k = 1, \ldots, K \tag{7}$$

where K is the maximum number of generations, defined by the option `Generations`. The parameters ϕ_a and ϕ_b are defined to comply with the stability condition (6), that is

$$\phi_a = 1 - \varepsilon \quad \text{and} \quad \phi_b = \frac{\alpha_1 + \alpha_2}{2} - 1 + \varepsilon \tag{8}$$

with $\varepsilon \ll 1$. A linearly decreasing particle inertia improves initial exploration of the search space, but finally ensures stronger attraction towards the personal and global best positions as the generation count increases.

IV. EXAMPLES

In this section, the functionality of `pso` is demonstrated by two different examples. The first example is the *Ackley* problem which is a common algebraic test problem. The second example demonstrates `pso` applied to a simplified automotive engineering problem of finding fuel optimal gear ratios of a manual transmission for a hybrid electric drivetrain.

A. Ackley Problem

The fitness function to be minimized takes the following form

$$f(x) = 20 + e - 20 \cdot \exp\left(-0.2 \cdot \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n} \sum_{i=1}^{n} \cos(2\pi \cdot x_i)\right). \tag{9}$$

The global minimum of this function is the origin, unless the origin is excluded by constraints. This is true regardless of the dimension of x, i.e., the value of $n \in \mathbb{N}$. For this example, we chose $n = 2$ to be able to visualize the problem. We implemented the fitness function in the following way

```
function f = ackley(x)

% Dimension
n = size(x,2);

% Ackley function
f = 20 + exp(1) ...
   -20*exp(-0.2*sqrt((1/n).*sum(x.^2,2))) ...
   -exp((1/n).*sum(cos(2*pi*x),2));
```

Moreover, we impose the following constraints

$$x_1 \leq x_2 \tag{10}$$

$$x_1^2 \leq 4 \cdot x_2 \tag{11}$$

$$-2 \leq \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq 2 \tag{12}$$

Note that the analytical solution to this constrained problem remains in the origin. The corresponding optimal fitness value is equal to zero. The non-linear inequality constraint (11) was implemented in the following way

```
function [c,ceq] = mynonlcon(x)

% Non-linear inequality constraints
c(1) = x(:,1).^2 - 4*x(:,2);

% Non-linear equality constraints
ceq = [];
```

Figure 3 shows the fitness function (9) including the constraints (10) to (12). The shaded area Ω indicates the feasible region of the search space. Notice the existence of several local minima.

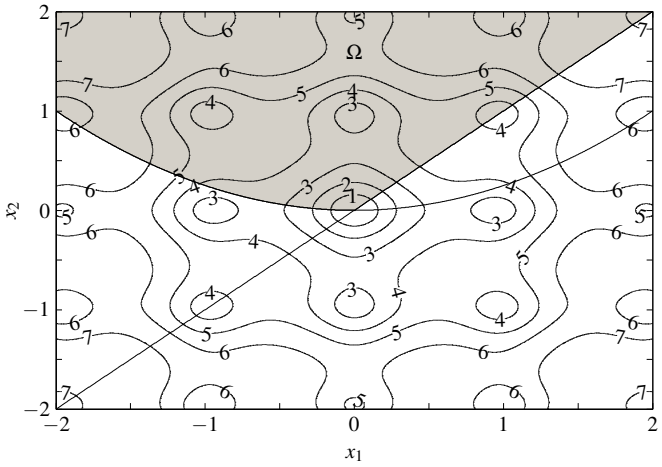

Fig. 3. The fitness function of the Ackley problem including linear and non-linear constraints.

The Ackley problem can be set up and solved by means of the `pso` function using the syntax and commands given below.

```
% EXAMPLE 1: ACKLEY PROBLEM
% Options
options = pso;
options.PopulationSize  = 24;
options.Vectorized      = 'on';
options.BoundaryMethod  = 'absorb';
options.Display         = 'off';

% Problem
problem = struct;
problem.fitnessfcn  = @ackley;
problem.nvars       = 2;
problem.Aineq       = [ 1 -1];
problem.bineq       = 0;
problem.lb          = [-2 -2];
problem.ub          = [ 2  2];
problem.nonlcon     = @mynonlcon;
problem.options     = options;

% Optimize
[x,fval,exitflag,output] = pso(problem)
```

Executing the code above gave rise to the following output:

```
x = 1.0e-13 *

     0.1166   -0.1011

fval = 4.1744e-14

exitflag = 1

output = problemtype: 'nonlinerconstr'
         generations: 61
           funccount: 1464
             message: [1x173 char]
       maxconstraint: 3.7907e-12
```

The `exitflag` and `output.message` indicate that the average cumulative change in value of the fitness function over `options.StallGenLimit` generations was less than `options.TolFun` and constraint violation less than `options.TolCon`. The vanishing value of `options.maxconstraint` confirms that at least one constraint is active in the solution. Finally, the values of `x` and `fval` show that we indeed found the analytical solution albeit with some insignificant numerical error.

B. Hybrid Vehicle Gear Ratio Optimization

In this example, we demonstrate a proficient interaction between the `pso` function and a generic dynamic programming MATLAB function by Sundström [11], namely the `dpm` function. The system under investigation is a hybrid electric vehicle comprising a manual transmission with six gears. The optimization problem is to find the six gear ratios that minimize the fuel consumed by the engine over a given driving cycle (JN1015). The vehicle is described using a discrete-time quasi-static model [12]. The control input u_j is the torque split factor determining how the traction torque, dictated by the driving cycle, is distributed between the engine and the electric motor. Subscript $j = 0,\ldots,N$ denotes the discrete time steps of the driving cycle.

The optimal values of the six gear ratios $x = [x_1,x_2,x_3,x_4,x_5,x_6]^T$ are influenced by the chosen control strategy and vice versa. Thus, to find the gear ratios with the largest possible potential of reducing fuel consumption, we only consider the globally optimal strategy u^*. This strategy is unique to every possible x. Thus, two optimization problems exist: one problem is the dynamic optimization problem of finding u^* given x. This problem is solved by means of dynamic programming (DP) using the `dpm` function. The other problem is the static optimization problem of finding the optimal gear ratios x^* given u^*. This problem is solved by means of particle swarm optimization using the `pso` function. Figure 4 illustrates how these two optimization problems are combined. The fitness function $f(x^k \,|\, u^*)$ is the optimal fuel consumption $(\text{L}/100\text{km})$ with respect to the six gear ratios x given the optimal control strategy u^*. The variable k is the generation number.

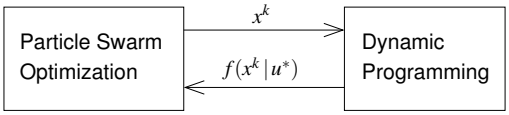

Fig. 4. Optimization of gearbox ratios using a compound of dynamic programming and particle swarm optimization.

The static optimization problem solved by PSO takes the following form

$$\min_x : \quad f(x \,|\, u^*) = \sum_{j=0}^{N} \Delta m_f(x \mid u_j^*) \tag{13}$$

$$\text{subject to} : \quad x_{i+1} \le x_i \quad \text{for } i = \{1,\ldots,5\} \tag{14}$$

$$x_{\min} \le x \le x_{\max} \tag{15}$$

where the constraints (14) ensure that the gear ratios decrease monotonically. The bounds (15) ensure the solution remain within reasonable values. The problem can be set up and solved using the syntax and commands given below.

```
% EXAMPLE 2: GEAR RATIO OPTIMIZATION
% Options
options = pso;
options.PopulationSize  = 24;
options.PlotFcns        = @psoplotbestf;
options.Display         = 'iter';
options.Vectorized      = 'off';
options.TolFun          = 1e-6;
options.StallGenLimit   = 50;

% Problem
problem = struct;
problem.fitnessfcn = @hev_main;
problem.nvars      = 6;
problem.Aineq      = [-1  1  0  0  0  0
                       0 -1  1  0  0  0
                       0  0 -1  1  0  0
                       0  0  0 -1  1  0
                       0  0  0  0 -1  1];
problem.bineq   = zeros(size(problem.Aineq,1),1);
problem.lb      = [13.5,  7.6, 5.0, 3.9, 3.0, 2.8];
problem.ub      = [20.4, 11.5, 7.6, 5.5, 4.4, 4.2];
problem.options = options;

% Optimize
[x,fval,exitflag,output] = pso(problem);
```

The fitness function `hev_main` comprises the dynamic optimization problem, i.e., the DP algorithm and the vehicle model, and returns the minimum fuel consumption (13) given x. Executing the code above returned following output:

```
x = 16.4048  7.6920  5.9285  3.7483  3.6259  3.5890

fval = 3.6764

exitflag = 1

output = problemtype: 'linearconstraints'
         generations: 95
           funccount: 2280
             message: [1x173 char]
       maxconstraint: -0.0085
```

The fuel consumption corresponding to the solution x was $3.6764\,\mathrm{L}/100\mathrm{km}$. The less-than-zero value of `output.maxconstraint` indicates that the solution is strictly within the interior of the search space and not on (or beyond) the boundaries. Figure 5 shows the output of the plotting function `psoplotbestf`. The mean value of the swarm is close to the the global best value. This observation raises confidence in that all particles have converged to approximately the same solution.

V. CONCLUSION

In this paper, we introduced a generic PSO function for MATLAB. The function uses practically the same syntax as common MATLAB functions such as `fmincon` and `ga`. Thus, the learning curve is flat for users already familiar with the syntax of those. In addition, the `pso` function can be substituted into existing `fmincon` or `ga` based optimization frameworks with little effort. The `pso` function, the sample functions presented herein, including the plotting function `psoplotbestf`, can be downloaded at `http://www.idsc.ethz.ch/Downloads`.

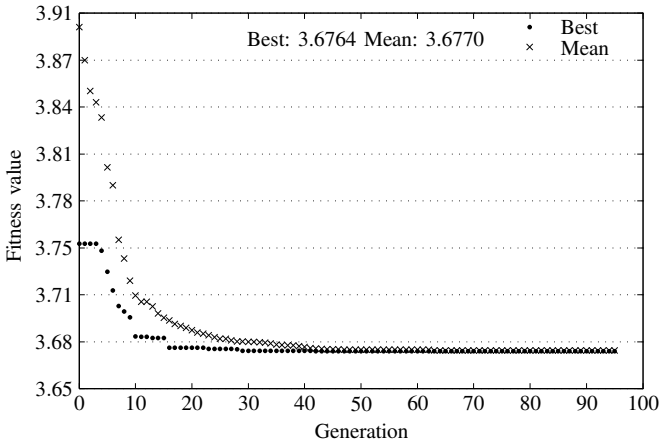

Fig. 5. The global best fitness value and the mean of all particles over generation number.

VI. ACKNOWLEDGMENTS

We would like to thank Daimler AG for having supported this project. We also thank our colleagues for testing the `pso` function and providing useful feedback.

REFERENCES

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, IEEE International Conference on*, vol. 4, nov/dec 1995, pp. 1942–1948.

[2] J. Duro and J. de Oliveira, "Particle swarm optimization applied to the chess game," in *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, june 2008, pp. 3702 –3709.

[3] P. Faria, Z. Vale, J. Soares, and J. Ferreira, "Particle swarm optimization applied to integrated demand response resources scheduling," in *Computational Intelligence Applications In Smart Grid (CIASG), 2011 IEEE Symposium on*, april 2011, pp. 1 –8.

[4] G. Lambert-Torres, H. Martins, M. Coutinho, C. Salomon, and F. Vieira, "Particle swarm optimization applied to system restoration," in *PowerTech, 2009 IEEE Bucharest*, 2009, pp. 1–6.

[5] M. Lanza, J. R. Perez, and J. Basterrechea, "Particle swarm optimization applied to planar arrays synthesis using subarrays," in *Antennas and Propagation (EuCAP), 2010 Proceedings of the Fourth European Conference on*, 2010, pp. 1–5.

[6] I. Oumarou, D. Jiang, and C. Yijia, "Particle swarm optimization applied to optimal power flow solution," in *Natural Computation, 2009. ICNC '09. Fifth International Conference on*, vol. 3, August 2009, pp. 284–288.

[7] B. Birge, "PSOt - a particle swarm optimization toolbox for use with matlab," in *Swarm Intelligence Symposium 2003, Proceedings of the IEEE*, 2003, pp. 182–186.

[8] R. Perez and K. Behdinan, "Particle swarm approach for structural design optimization," *Computers & Structures*, vol. 85, no. 19-20, pp. 1579–1588, 2007.

[9] G. Pulido and C. Coello, "A constraint-handling mechanism for particle swarm optimization," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 2, 2004, pp. 1396–1403.

[10] C. Coello, G. Pulido, and M. Lechuga, "Handling multiple objectives with particle swarm optimization," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 3, pp. 256–279, june 2004.

[11] O. Sundstrom and L. Guzzella, "A generic dynamic programming matlab function," in *Control Applications, (CCA) & Intelligent Control, (ISIC), 2009 IEEE*, July 2009, pp. 1625–1630.

[12] L. Guzzella and A. Sciarretta, *Vehicle Propulsion Systems: Introduction to Modeling and Optimization*. Springer, 2005.