SwarmOps for Java

Numeric & Heuristic Optimization
Source-Code Library for Java
The Manual
Revision 1.0

By

Magnus Erik Hvass Pedersen

June 2011

Contents

1. Introduction

SwarmOps is a source-code library for doing numerical optimization in Java. It features popular optimizers which do not use the gradient of the problem being optimized. SwarmOps also makes it easy to discover the behavioural or control parameters making an optimizer perform well. This is done by using another overlaid optimizer and is known here as Meta-Optimization but is also known in the literature as Meta-Evolution, Super-Optimization, Parameter Calibration, Parameter Tuning, etc. The success of SwarmOps in doing meta-optimization is mainly due to three things:

1. SwarmOps uses the same interface for an optimization problem and an optimization method, meaning that an optimization method is also considered an optimization problem. This modular approach allows for meta-optimization, meta-meta-optimization, and so on.
2. SwarmOps uses a simple time-saving technique called Pre-Emptive Fitness Evaluation which makes meta-optimization more tractable to execute.
3. SwarmOps features a simple optimization method that works well as the overlaid meta-optimizer because it is usually able to find good behavioural parameters for an optimizer using a small number of iterations, again making meta-optimization more tractable to execute.

1.1 Installation

To use the SwarmOps JAR-library in Eclipse:

1. Unpack the SwarmOps archive to a convenient directory.
2. Open the workspace in which you will use SwarmOps.
3. Add the SwarmOps JAR-file to the build-path of the projects that must use it.

4. Import swarmops.optimizers.* and other classes you will need in your source-files.

To use the SwarmOps test-projects in Eclipse:

1. Unpack the SwarmOps archive to a convenient directory.
2. Create a new Eclipse workspace in that directory.
3. Select the menu: File / Import / Existing projects, and import the SwarmOps projects.

1.2 Tutorials

Several examples on how to use SwarmOps are supplied with the source-code and are well documented. Tutorials have therefore been omitted in this manual.

1.3 Updates

Updates to SwarmOps can be found on the internet: www.hvass-labs.org

1.4 License

This manual may be downloaded, printed, and used for any personal purpose, be it commercial or non-commercial, provided the author(s) are not held responsible for your actions, or any damage caused by your use of the manual. If you want to distribute the manual commercially, for example in a printed book, or on a web-page that requires payment, then you must obtain a license from the author(s).

2. What Is Optimization?

Solutions to some problems are not merely deemed correct or incorrect but are rated in terms of quality. Such problems are known as optimization problems because the goal is to find the candidate solution with the best, that is, *optimal* quality.

Fitness Function

SwarmOps works for real-valued and single-objective optimization problems, that is, optimization problems that map candidate solutions from n-dimensional real-valued spaces to one-dimensional real-valued spaces. Mathematically speaking we consider optimization problems to be functions f of the following form:

$$f: \mathbb{R}^n \to \mathbb{R}$$

In SwarmOps it is assumed that f is a minimization problem, meaning that we are searching for the candidate solution $\vec{x} \in \mathbb{R}^n$ with the smallest value $f(\vec{x})$. Mathematically this may be written as:

$$\text{Find } \vec{x} \text{ such that } \forall \vec{y} \in \mathbb{R}^n\text{:}\ \ f(\vec{x}) \leq f(\vec{y})$$

Typically, however, it is not possible to locate the exact optimum and we must be satisfied with a candidate solution of sufficiently good quality but perhaps not quite optimal. In this manual we refer to the optimization problem f as the fitness function but it is also known in the literature as the cost function, objective function, error function, quality measure, etc. We may refer to candidate solutions as positions, agents or particles, and to all possible candidate solutions as the search-space.

Maximization

SwarmOps can also be used with maximization problems. If $h: \mathbb{R}^n \to \mathbb{R}$ is a maximization problem then the equivalent minimization problem is: $f(\vec{x}) = -h(\vec{x})$

Gradient-Based Optimization

The classic way of optimizing a fitness function f is to first deduce its gradient $\nabla f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ consisting of the partial differentials of f, that is:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]$$

Then the gradient is followed iteratively in the direction of steepest descent, or a quasi-Newton optimizer can be used. This requires not only for the fitness function f to be differentiable, but the gradient can also be very laborious to derive and the execution can be very time-consuming.

Heuristic Optimization

An alternative to gradient-based optimization methods is to let the optimization be guided solely by the fitness values. This kind of optimization has no explicit knowledge of how the fitness landscape looks but merely considers the fitness function to be a black box that takes candidate solutions as input and produces some fitness value as output. This is known in the literature as Derivate Free Optimization, Direct Search, Heuristic Optimization, Meta-Heuristics, Black-Box Optimization etc.

2.1 Constraints

Constraints split the search-space into regions of feasible and infeasible candidate solutions. For instance, an engineering problem could have a mathematical model that should be optimized but actually producing the solution in the real world puts some constraints on what is feasible. There are different ways of supporting and handling constraints in heuristic optimization.

Boundaries

A simple form of constraints is search-space boundaries. Instead of letting f map from the entire n-dimensional real-valued space, it is often practical to use only a part of this vast search-space. The lower and upper boundaries that constitute the search-space are denoted as $\vec{b}_{lo}$ and $\vec{b}_{up}$ so the fitness function is of the form:

$$f\colon [\vec{b}_{lo}, \vec{b}_{up}] \to \mathbb{R}$$

Such boundaries are typically enforced in the optimization methods by moving candidate solutions back to the boundary value if they have exceeded the boundaries. This is the default type of constraints in SwarmOps.

Penalty Functions

More complicated constraints are supported transparently by any heuristic optimizer by penalizing infeasible candidate solutions, that is, by adding a penalty function to the fitness function. Examples can be found in the Penalized benchmark problems.

General Constraints

SwarmOps supports general constraints by taking feasibility (constraint satisfaction) into account when comparing candidate solutions. Normally we determine whether candidate solution $\vec{x}$ is better than $\vec{y}$ by comparing their fitness: $f(\vec{x}) < f(\vec{y})$, but it is also possible to take feasibility into account. Feasibility is a Boolean; either a candidate solution is feasible or it is infeasible, so the comparison operator is:

$$\boxed{\begin{array}{c} (\vec{x} \text{ is better than } \vec{y}) \\ \Updownarrow \\ (\vec{y} \text{ is infeasible and } \vec{x} \text{ is feasible}) \text{ or} \\ (\vec{y} \text{ is infeasible and } \vec{x} \text{ is infeasible and } f(\vec{x}) < f(\vec{y})) \text{ or} \\ (\vec{y} \text{ is feasible and } \vec{x} \text{ is feasible and } f(\vec{x}) < f(\vec{y})) \end{array}}$$

Note that the actual implementation of this comparison is simplified somewhat. Also, when $\vec{y}$ is feasible and $\vec{x}$ is infeasible then their fitness need not be computed because $\vec{x}$ is worse than $\vec{y}$ due to their mutual feasibility. This is used in the implementation to avoid fitness computations whenever possible.

Phases of Constrained Optimization

Using the above comparison operator means that optimization has two phases. First the optimizer will likely only find infeasible candidate solutions, so it optimizes the fitness of infeasible solutions. Then at some point the optimizer hopefully discovers a feasible candidate solution and regardless of its fitness it will then become the best-found solution of the optimizer and form the basis of the further search, so now the fitness of feasible solutions is being optimized.

Difficulty of Constrained Optimization

While SwarmOps gives you the ability to implement any constraint imaginable, they will make it increasingly difficult for the optimizer to find feasibly optimal solutions, because constraints narrow the feasible regions of the search-space. You should therefore also narrow the initialization and search-space boundaries to be as close to the feasible region as possible.

Implementation

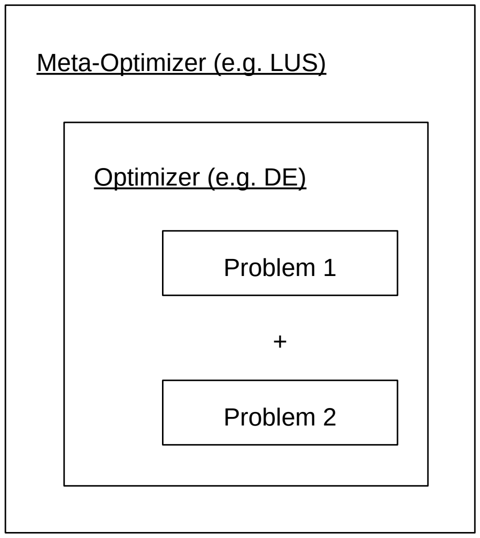
There are two methods in the Problem-class where you can implement constraints:

- EnforceConstraints() allows you to make repairs to a candidate solution before its feasibility and fitness is evaluated. For example, when search-space boundaries are used as constraints then the repairing would consist of moving candidate solutions back between boundaries if they were overstepped. This is done by default.
- Feasible() evaluates and returns the feasibility of a candidate solution without altering it.

The TestCustomProblem tutorial program gives an example of their usage.

2.2 Meta-Optimization

Optimization methods usually have a number of user-defined parameters that govern the behaviour and efficacy of the optimization method. These are called the optimizer's behavioural or control parameters. Finding a good choice of these behavioural parameters has previously been done manually by hand-tuning and sometimes using coarse mathematical analysis. It has also become a common belief amongst researchers that the behavioural parameters can be adapted during optimization so as to improve overall optimization performance, however, this has been demonstrated to be unlikely in general, see (1) (2) (3). Tuning behavioural parameters can be considered an optimization problem in its own right and hence solved by an overlaid optimization method. This is known here as Meta-Optimization but is also known in the literature as Meta-Evolution, Super-Optimization, Parameter Calibration, etc. The success of SwarmOps in doing meta-optimization stems mainly from three things, first that SwarmOps features an optimization method that is particularly suitable as the overlaid meta-optimizer because it quickly discovers well performing behavioural parameters (this is the LUS method described in section 3.4 below), and second because SwarmOps employs a simple technique for reducing computational time called Pre-Emptive Fitness Evaluation, and third because SwarmOps uses the same function-interface for both optimization problems and optimization methods. A number of scientific publications use SwarmOps for meta-optimization and have more elaborate descriptions than those given here, as well as having literature surveys and experimental results, please see (1) (2) (3) (4). The concept of meta-optimization can be illustrated schematically:



Here the optimizer whose behavioural parameters are to be tuned is taken to be the DE method (described later in section 3.5). The SwarmOps framework allows for parameters to be tuned with regard to multiple optimization problems, which is sometimes necessary to make the performance of the behavioural parameters generalize better to problems other than those the parameters were specifically tuned for. In this example the DE parameters are tuned for two problems.

Fitness Normalization

Fitness functions must be non-negative to work properly with meta-optimization in SwarmOps. This is because Pre-Emptive Fitness Evaluation works by summing fitness values for several optimization runs and aborting this summation when the fitness sum becomes worse than that needed for the new candidate solution to be accepted as an improvement. This means the fitness values must be non-negative so the fitness sum is only able to grow worse and the evaluation can thus be aborted safely. SwarmOps for Java does this normalization automatically, provided you accurately implement the MinFitness field of the Problem-class. For example, you may have some fitness function f which maps to, say $[-4, \infty)$, and you would then have to set MinFitness to -4. It is best to make MinFitness accurate so that $f(\vec{x}) - MinFitness = 0$ for the optimum $\vec{x}$, that is, MinFitness should be the fitness of the optimum. You should be able to estimate a lower fitness boundary for most real-world problems, and if you are unsure what the theoretical boundary value is, you may choose some boundary fitness value of ample but not extreme magnitude.

Fitness Weights for Multiple Problems

If you are using multiple problems in meta-optimization, you may need to experiment with weights on each problem so as to make their influence on the meta-optimization process more equal.

Advice

The LUS method is generally recommended as the overlaid meta-optimizer. The tutorial source-code contains suggestions for experimental settings which have been found to work well. It is best if you can perform meta-optimization with regard to the problems you are ultimately going to use the optimization method for. However, if your fitness function is very expensive to evaluate then you may try and resort to using benchmark problems as a temporary replacement when meta-optimizing the behavioural parameters of your optimizer, provided you use multiple benchmark problems and the optimization settings are similar to the settings that are to be used for the real problem. This means you should use benchmark problems of similar dimensionality and with a similar number of optimization iterations as you would use for the actual problem you are ultimately going to optimize.

Constraints and Meta-Optimization

Two issues regarding constraints in meta-optimization should be mentioned:

1. Constraints can be made on an optimizer's control parameters in the same manner as for an optimization problem by implementing the EnforceConstraints() and Feasible() methods in the optimizer's class. This means the meta-optimizer will search for control parameters that are feasibly optimal, allowing you to search for control parameters that meet certain criteria, e.g. have certain relationships to each other such as one parameter being smaller

than another or the parameters having different signs, etc. See the source-code of the MOL optimizer for an example of this.

2. Constraint satisfaction is ignored when determining how well an optimizer performs in making up the meta-fitness measure. This is an open research topic but experiments suggest that an optimizer’s control parameters should be meta-optimized for unconstrained problems and this will yield good performance on constrained problems as well.

Meta-Meta-Optimization

In using meta-optimization to find the best performing parameters of some optimizer, one may naturally ask the question: What are then the best performing parameters for the meta-optimizer itself? It makes good sense to find the best meta-optimizer if one is going to use it often and the best parameters for the meta-optimizer can be found by employing yet another layer of optimization, which may be termed Meta-Meta-Optimization. This is supported in SwarmOps but a tutorial program is currently not included.

3. Optimization Methods

This chapter gives brief descriptions of the optimization methods that are supplied with SwarmOps and recommendations for their use.

3.1 Choosing an Optimizer

When faced with a new optimization problem the first optimizer you may want to try is the PS method from section 3.2 which is often sufficient and has the advantage of converging (or stagnating) very quickly. PS also does not have any behavioural parameters that need tuning so it either works or doesn't. If the PS method fails at optimizing your problem you may want to try the LUS method from section 3.4 which sometimes works a little better than PS (and sometimes a little worse). You may need to run PS and LUS several times as they may converge to sub-optimal solutions. If PS and LUS both fail you will want to try the DE, MOL or PSO methods and experiment with their behavioural parameters.

As a rule of thumb PS and LUS stagnate rather quickly, say, after $40 \cdot n$ iterations, where n is the dimensionality of the search-space, while DE, MOL and PSO require substantially more iterations, say, $500 \cdot n$ or $2000 \cdot n$ and sometimes even more.

If these optimizers fail, you either need to tune their behavioural parameters using meta-optimization or use another optimizer altogether, e.g. CMA-ES.

3.2 Gradient Descent (GD)

A classic way of minimizing some fitness function $f: \mathbb{R}^n \to \mathbb{R}$ is to repeatedly follow the gradient in the direction of steepest descent. The gradient function $\nabla f: \mathbb{R}^n \to \mathbb{R}^n$ is defined as the vector of the partial differentials of f, that is:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]$$

How it Works

The position $\vec{x}$ is first chosen randomly from the search-space and then updated iteratively according to the following formula, regardless of fitness improvement:

$$\vec{x} \leftarrow \vec{x} - d \cdot \frac{\nabla f(\vec{x})}{\|\nabla f(\vec{x})\|}$$

With $d > 0$ being the step-size. When f is a minimization problem the descent direction is followed, that is, we subtract the gradient from the current position instead of adding it as we would have done for ascending a maximization problem.

Advice

The GD method has some drawbacks, namely that it requires the gradient ∇f to be defined, that the gradient may be expensive to compute, and that GD may approach the optimum too slowly. So you may wish to try the PS method first. Other variants of GD exist for improving performance and time usage, e.g. Conjugate GD and quasi-Newton methods, but they have not been implemented in SwarmOps.

3.3 Pattern Search (PS)

The optimization method known here as Pattern Search (PS) is originally due to Fermi and Metropolis as described in (5) and a similar method is due to Hooke and Jeeves (6). The implementation presented here is the variant from (3).

How it Works

PS uses one agent / position in the search-space which is being moved around. Let the position be denoted $\vec{x} \in \mathbb{R}^n$ which is initially picked at random from the entire search-space. The initial sampling range is the entire search-space: $\vec{d} = \vec{b}_{up} - \vec{b}_{lo}$. The potential new position is denoted $\vec{y}$ and is sampled as follows. First pick an index $R \in \{1, \dots, n\}$ at random and let $y_R = x_R + d_R$ and $y_i = x_i$ for all $i \neq R$. If $\vec{y}$ improves on the fitness of $\vec{x}$ then move to $\vec{y}$. Otherwise halve and reverse the sampling range for the R'th dimension: $d_R \leftarrow -d_R/2$. Repeat this a number of times.

3.4 Local Unimodal Sampling (LUS)

The LUS optimization method performs local sampling by moving a single agent around in the search-space with a simple way of decreasing the sampling range during optimization. The LUS method was presented in (3) (7).

How it Works

The agent's current position is denoted $\vec{x} \in \mathbb{R}^n$ and is initially picked at random from the entire search-space. The potential new position is denoted $\vec{y}$ and is sampled from the neighbourhood of $\vec{x}$ by letting $\vec{y} = \vec{x} + \vec{a}$, where $\vec{a} \sim U(-\vec{d}, \vec{d})$ is a random vector picked uniformly from the range $(-\vec{d}, \vec{d})$, which is initially $\vec{d} = \vec{b}_{up} - \vec{b}_{lo}$, that is, the full range of the entire search-space defined by its upper boundaries $\vec{b}_{up}$ and its lower boundaries $\vec{b}_{lo}$. LUS moves from position $\vec{x}$ to position $\vec{y}$ in case of improvement to the fitness. Upon each failure for $\vec{y}$ to improve on the fitness of $\vec{x}$, the sampling range is decreased by multiplication with a factor q:

$$\vec{d} \leftarrow q \cdot \vec{d}$$

where the decrease factor q is defined as:

$$q = \sqrt[\gamma n]{1/2} = \left(\frac{1}{2}\right)^{1/\gamma n}$$

where n is the dimensionality of the search-space and γ is a user-defined parameter used to adjust the rate of sampling-range decrease. A value of $\gamma = 3$ has been found to work well for many optimization problems.

3.5 Differential Evolution (DE)

The multi-agent optimization method known as Differential Evolution (DE) is originally due to Storn and Price (8). Many DE variants exist and a simple one is implemented in the DE-class and a number of different DE variants are available through the DESuite class.

How it Works

DE uses a population of agents. Let $\vec{x}$ denote the position of an agent being updated and which has been picked at random from the entire population. Let $\vec{y} = [y_1, \dots, y_n]$ be its new potential position computed as follows (this is the so-called DE/rand/1/bin variant):

$$y_i = \begin{cases} a_i + F(b_i - c_i), & i = R \vee r_i < CR \\ x_i, & \text{else} \end{cases}$$

where the vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ are the positions of distinct and randomly picked agents from the population. The index $R \in \{1, \dots, n\}$ is randomly picked and $r_i \sim U(0,1)$ is also picked randomly for each dimension i. A move is made to the new position $\vec{y}$ if it improves on the fitness of $\vec{x}$. The user-defined parameters consist of the differential weight F, the crossover probability CR, and the population-size NP.

3.6 Particle Swarm Optimization (PSO)

The optimization method known as Particle Swarm Optimization (PSO) is originally due to Kennedy, Eberhart, and Shi (9) (10). It works by having a swarm of candidate solutions called particles, each having a velocity that is updated recurrently and added to the particle's current position to move it to a new position.

How it Works

Let $\vec{x}$ denote the current position of a particle from the swarm. Then the particle's velocity $\vec{v}$ is updated as follows:

$$\vec{v} \leftarrow \omega\vec{v} + \varphi_p r_p(\vec{p} - \vec{x}) + \varphi_g r_g(\vec{g} - \vec{x})$$

where the user-defined parameter ω is called the inertia weight and the user-defined parameters φ_p and φ_g are weights on the attraction towards the particle's own best known position $\vec{p}$ and the swarm's best known position $\vec{g}$. These are also weighted by the random numbers $r_1, r_2 \sim U(0,1)$. In addition to this, the user also determines the swarm-size S. In the SwarmOps implementation the velocity is bounded to the full range of the search-space so an agent cannot move farther than from one search-space boundary to the other in a single move.

Once the agent's velocity has been computed it is added to the agent's position:

$$\vec{x} \leftarrow \vec{x} + \vec{v}$$

3.7 Many Optimizing Liaisons (MOL)

A simplification of PSO is called Many Optimizing Liaisons (MOL) and was originally suggested by Kennedy (11) who called it the "Social Only" PSO. The name MOL is used in (4) where more thorough studies were made. MOL differs from

PSO in that it eliminates the particle's best known position $\vec{p}$. This has been found to improve performance somewhat on some problems and also makes it easier to tune the behavioural parameters.

3.8 Mesh (MESH)

The fitness can be computed at regular intervals of the search-space using the MESH method. For increasing search-space dimensionality this incurs an exponentially increasing number of mesh-points in order to retain a similar interval-size. This phenomenon is what is known as the Curse of Dimensionality. The MESH method is used as any other optimization method in SwarmOps and will indeed return as its solution the mesh-point found to have the best fitness. The quality of this solution will depend on how coarse or fine the mesh is. The MESH method is mostly used to make plots of the fitness landscape for simpler optimization problems, or to study how different choices of behavioural parameters influence an optimization method's performance, that is, how does the meta-fitness landscape look. The MESH method is not intended to be used as an optimizer.

Bibliography

1. **Pedersen, M.E.H. and Chipperfield, A.J.** *Parameter tuning versus adaptation: Proof of principle study on differential evolution.* s.l. : Hvass Laboratories, 2008. HL0802.

2. —. *Tuning Differential Evolution for Artificial Neural Networks.* s.l. : Hvass Laboratories, 2008. HL0803.

3. **Pedersen, M.E.H.** *Tuning & Simplifying Heuristical Optimization (PhD Thesis).* s.l. : School of Engineering Sciences, University of Southampton, United Kingdom, 2010.

4. *Simplifying Particle Swarm Optimization.* **Pedersen, M.E.H. and Chipperfield, A.J.** s.l. : Applied Soft Computing, 2010, Vol. 10, pp. 618-628.

5. *Variable metric method for minimization.* **Davidon, W.C.** 1, s.l. : SIAM Journal on Optimization, 1991, Vol. 1, pp. 1-17.

6. *"Direct Search" solution for numerical and statistical problems.* **Hooke, R. and Jeeves, T.A.** 2, s.l. : Journal of the Association for Computing Machinery (ACM), 1961, Vol. 8, pp. 212-229.

7. **Pedersen, M.E.H. and Chipperfield, A.J.** *Local Unimodal Sampling.* s.l. : Hvass Laboratories, 2008. HL0801.

8. *Differential evolution - a simple and efficient heuristic for global optimization over continuous space.* **Storn, R. and Price, K.** s.l. : Journal of Global Optimization, 1997, Vol. 11, pp. 341-359.

9. *Particle Swarm Optimization.* **Kennedy, J. and Eberhart, R.** Perth, Australia : IEEE Internation Conference on Neural Networks, 1995.

10. *A Modified Particle Swarm Optimizer.* **Shi, Y. and Eberhart, R.** Anchorage, AK, USA : IEEE International Conference on Evolutionary Computation, 1998.

11. *The particle swarm: social adaptation of knowledge.* **Kennedy, J.** Indianapolis, USA : Proceedings of the IEEE International Conference on Evolutionary Computation, 1997.