SwarmOps

Numeric & Heuristic Optimization Source-Code Library for C#
The Manual
First Edition

By

Magnus Erik Hvass Pedersen

September 2009

Copyright © 2009, all rights reserved by the author.
Please see page 5 for license details.

Contents

Contents 2
Preface 3
1. Introduction 4
1.1 Installation 4
1.2 Tutorials 5
1.3 Updates 5
1.4 License 5
2. What Is Optimization? 7
2.1 Optimization 7
2.2 Meta-Optimization 9
2.3 Meta-Meta-Optimization (Advanced Topic) 13
3. Optimization Methods 16
3.1 Choosing an Optimizer 16
3.2 Mesh (MESH) 16
3.3 Gradient Descent (GD) 17
3.4 Gradient Emancipated Descent (GED) 18
3.5 Random Sampling (RND) 19
3.6 Pattern Search (PS) 19
3.7 Local Unimodal Sampling (LUS) 20
3.8 Differential Evolution (DE) 22
3.9 Particle Swarm Optimization (PSO) 23
Bibliography 25

Preface

SwarmOps is a source-code library for doing numerical optimization. This C# version features the optimizer variants that I've found through my own research to work particularly well. The C# version of SwarmOps is mainly aimed at application developers who need to optimize some mathematical problem in their programs, but wish to avoid the hassle of having to do a whole lot of research first to find out which optimizers work well. The source-code is still easy to expand though, and researchers may therefore also find it useful for experimental work. Godspeed!

M. E. H. Pedersen, Denmark, September 2009

1. Introduction

SwarmOps is a source-code library for doing numerical optimization in the C# programming language. In addition to providing several different optimization methods that have been found to work well on a range of optimization problems, SwarmOps also makes it easy to discover the behavioural or control parameters that makes an optimizer perform well. This is done by employing another overlaid optimizer, and is known here as Meta-Optimization (or Meta-Optimisation) 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 employs 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 the best behavioural parameters for an optimization method using only a fairly small number of iterations.

1.1 Installation

To install SwarmOps follow these simple steps:

1. Unpack the SwarmOps archive to a convenient directory.

2. In MS Visual Studio open the Solution in which you will use SwarmOps.
3. Add the SwarmOps project to the solution.
4. Add a reference to SwarmOps in all the projects which must use it.

Additional Requirements

SwarmOps requires a Random Number Generator (RNG) and by default uses the RandomOps library version 1 or later (1), which must be installed before SwarmOps can be used. If you wish to use another RNG, then the easiest thing is to make a wrapper for that RNG in RandomOps, so you do not have to change all the source-code of SwarmOps that uses the RNG.

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

To obtain updates to the SwarmOps source-code library or to get newer revisions of this manual, go to the library's webpage at: www.hvass-labs.org

1.4 License

Source-Code License

The SwarmOps source-code is published under the GNU Lesser General Public License (2), which means you may distribute commercial programs that link with the SwarmOps library, as well as make alterations to the library itself. There are certain

terms to be met though, please see the license included in the source-code distribution for details.

Manual 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?

This chapter describes the concepts of Optimization, Meta-Optimization, and Meta-Meta-Optimization.

2.1 Optimization

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

Fitness Function

The SwarmOps library is concerned with real-coded and single-objective optimization problems, that is, optimization problems which map 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:

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

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

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

Typically however, it is not possible to locate the exact optimum and we must be satisfied with a solution of sufficiently good quality, but perhaps not strictly optimal. In this manual we shall refer to the optimization problem f as the fitness function, but this is also known in the literature as the cost function, the objective function,

the error function, the quality measure, etc. We shall sometimes refer to candidate solutions as positions, and as the space of possible solutions (or positions) as the search-space.

Maximization

SwarmOps can also be used with maximization problems. Assume $h: \mathbb{R}^n \rightarrow \mathbb{R}$ is a maximization problem then the equivalent minimization problem f is merely:

$$f(\vec{x}) = -h(\vec{x})$$

Boundaries

SwarmOps allows for a simple type of constraints, namely 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:

$$f: [\vec{b}_{lo}, \vec{b}_{up}] \rightarrow \mathbb{R}$$

Such boundaries are typically enforced in the optimization methods by saturating or clipping candidate solutions at the boundary values, meaning that if a candidate solution steps over a boundary then the candidate solution is moved back to the boundary value.

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:

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

Then the gradient is followed iteratively in the direction of steepest descent. 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 progress 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, and so on, but we shall generally just call it Black-Box optimization.

Terminology

We use the term *agent* synonymously to a position in the search-space and the term *swarm* or *population* as a collection of such agents; the image being that an optimization method will move its agent(s) around in the search-space in an attempt to improve fitness. Some optimization methods employ just a single agent which they move around in the search-space, while other optimization methods work by combining the effort of multiple agents. In the literature agents are sometimes also known as individuals, particles, etc.

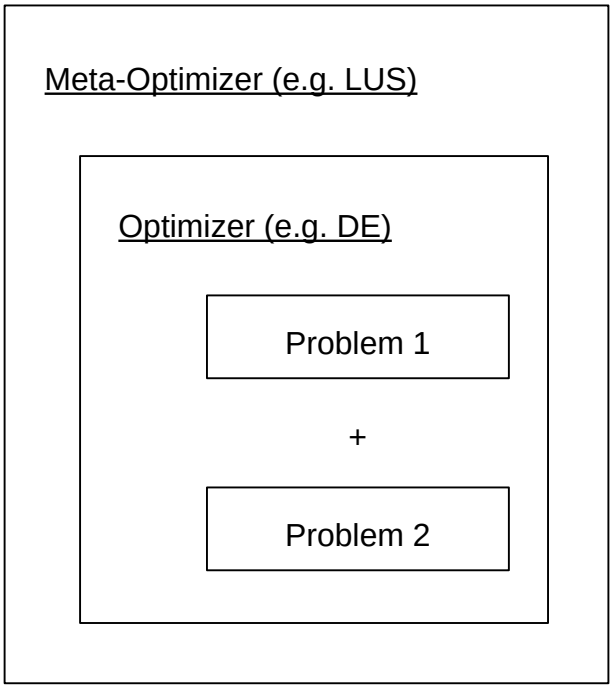
2.2 Meta-Optimization

Optimization methods usually have a number of user-defined parameters that govern the behaviour and efficacy of the optimization method. These are called behavioural

or control parameters. Finding the best 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 (3) (4) (5). 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 the best performing behavioural parameters (this is the LUS method described in section 3.6 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 (3) (4) (5) (6).

Concept Illustration

The overall 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 3.8). 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.

Choice of Meta-Optimizer

The LUS method described in section 3.6 has been found to be particularly suitable as the overlaid meta-optimizer, because it usually is rapid in finding the best performing behavioural parameters of another optimization method. This is important because meta-optimization remains a very expensive task, as each iteration of the meta-optimizer consists of performing a number of repeated runs of the optimizer.

Pre-Emptive Fitness Evaluation

A simple technique was used in (5) (4) (6) for saving computational time when doing meta-optimization. The technique is called Pre-Emptive Fitness Evaluation because it consists of aborting a meta-fitness evaluation once it becomes known that it does not lead to the discovery of improved parameters. The technique is simple to implement and yields a substantial time-saving of 50-85%.

Fitness Normalization

Fitness functions must be non-negative to work properly with meta-optimization in SwarmOps. This is because of the use of Pre-Emptive Fitness Evaluation that 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 C# 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 deduce 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

As previously mentioned, the LUS method is generally recommended as the overlaid meta-optimizer, and the DE method is recommended as the 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, and also with the same optimization settings that will eventually be used. 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. Although scientific results do not yet exist on this matter, preliminary results seem to suggest that it is possible to use benchmark problems in meta-optimization; provided you use multiple benchmark problems, and provided 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 similar optimization run-lengths as you would use for the actual problem you are ultimately going to optimize.

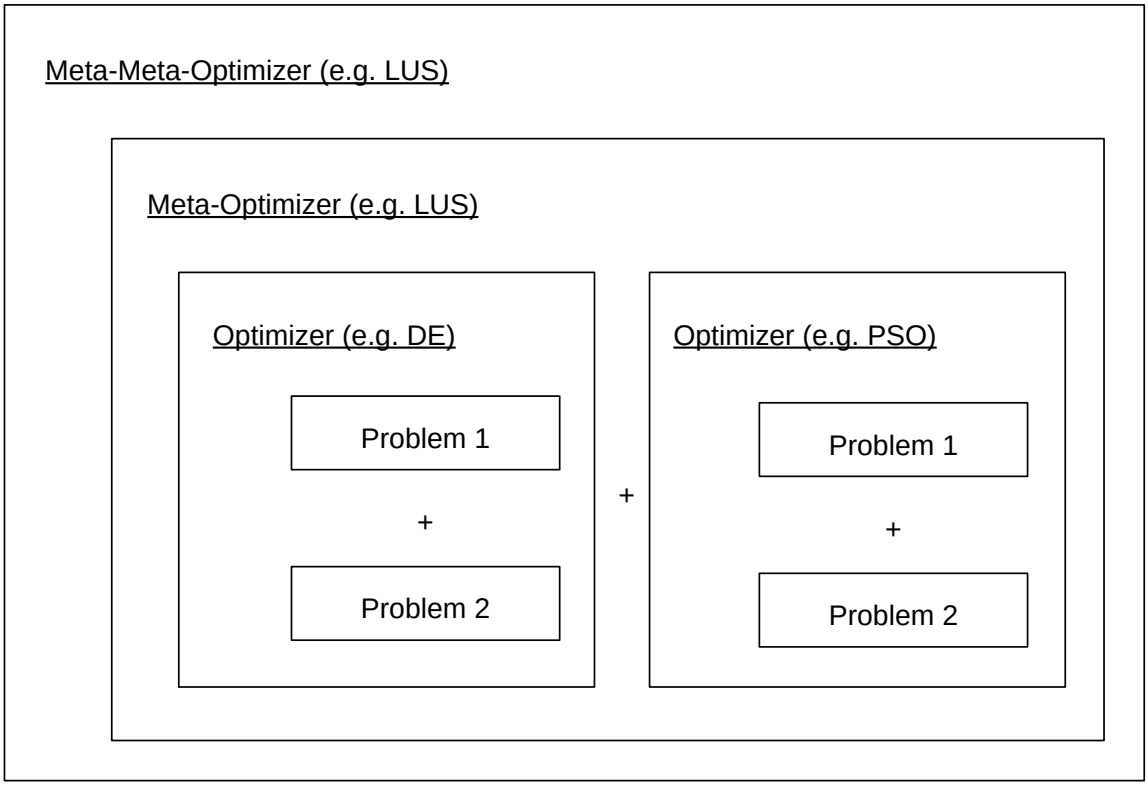
2.3 Meta-Meta-Optimization (Advanced Topic)

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 pa-

rameters 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 will be known here as Meta-Meta-Optimization (some might call it Meta-Meta-Evolution). The SwarmOps framework naturally supports meta-meta-optimization due to its use of the same interface for both optimization problems and methods, so an optimization method is considered to be an optimization problem as well. And due to the modular SwarmOps framework this also means that any number of meta-layers is supported; although it may not be that useful to go further than the Meta-Meta-layer, in part because meta-meta-optimization is already very time-consuming to execute, but also because most researchers and practitioners will be satisfied with a good meta-optimizer and do not need an optimal meta-meta-optimizer as well.

Concept Illustration

The overall concept of meta-meta-optimization can be illustrated schematically as follows. Note that the SwarmOps framework supports both the use of multiple optimization problems as well as multiple optimizers when doing meta-meta-optimization. This is useful because it allows the behavioural parameters of the meta-optimizer to be meta-meta-optimized with regard to several optimizers, and this will hopefully make the meta-optimizer work well in finding parameters for optimizers it was not specifically tuned for. The schematic drawing of meta-meta-optimization is:



Choice of Meta-Meta-Optimizer

The success of meta-meta-optimization depends on the proper choice of meta-meta-optimizer, which must be able to quickly find the best parameters for the meta-optimizer so as to keep the time-usage as low as possible. It can be expected however, that the meta-fitness landscape is fairly smooth and the LUS method therefore also appears to be suitable as the meta-meta-optimizer.

3. Optimization Methods

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

3.1 Choosing an Optimizer

The first optimizer you would want to try when faced with a new optimization problem is probably the PS method from section 3.6. Oftentimes PS is sufficient and it has the advantage of converging (or stagnating) very quickly. PS also does not have any behavioural parameters that need tuning, so either it works or it doesn't. If the PS method fails at optimizing your problem you may want to try the LUS method from section 3.7 which sometimes works a little better than PS (and sometimes a little worse). If the LUS method fails as well, you will want to try the DE method, as the variant implemented in SwarmOps is quite versatile and can have its behavioural parameters tuned to work well with specific optimization problems and settings, such as number of optimization iterations allowed. Although at first you will of course want to try using DE with its default parameters.

3.2 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 re-

turn 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.

Advice

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.

3.3 Gradient Descent (GD)

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

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

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. Note that due to the assumption that 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. The GED variant described next, offers a way to save computational time for some fitness functions. Other variants of the GD method are also in existence for improving performance and time usage, such as Conjugate GD, but they have not been implemented in SwarmOps as of yet.

3.4 Gradient Emancipated Descent (GED)

Some fitness functions are much more time consuming to evaluate than the gradient, and in such cases it may sometimes be possible to follow the gradient for a number of iterations before re-evaluating the fitness. It should be noted however, that following the gradient in steepest descent does not guarantee an improvement in fitness, whence we cannot just compute the fitness of the final position, but will have to compute the fitness during the optimization run as well. This variant of the GD method is known as Gradient Emancipated Descent (GED) and is taken from (5). The GED variant introduces a probability parameter p which determines whether the fitness should be evaluated.

Advice

If you are going to use the GD method to optimize a computationally intensive fitness function, then it may be beneficial to try and optimize the function using the GED method first. It may not work as well as the GD method, but may give you an indication of what kind of performance can be expected from the GD method, while

only using a small fraction of the computational time. It may also give you a result that you can use for seeding the GD method, so as to start its optimization run at a better position in the search-space, instead of just letting it start at a randomly chosen position.

3.5 Random Sampling (RND)

When the gradient of the function to be optimized is unavailable then we must resort to Black-Box optimization methods. The easiest way of performing black-box optimization is to pick candidate solutions from the entire search-space completely at random. This optimization method is known here as Random Sampling (RND).

Advice

As can be expected the RND method is very inefficient, and it is recommended that you only use the RND method as a measure of base-level performance in comparison with other optimization methods.

3.6 Pattern Search (PS)

In the optimization method known here as Pattern Search (PS), a single agent is being moved around in the search-space by halving its search-range one dimension at a time. The PS method is originally due to Fermi and Metropolis as described in (8), and a similar method is due to Hooke and Jeeves (9). The implementation presented here is a slight variant that was developed in (5).

Sampling

Let the current position be denoted $\vec{x} \in \mathbb{R}^n$ which is initially picked at random from the entire search-space. Let the initial sampling range be 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 $R \in \{1, \ldots, n\}$ 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: $d_R \leftarrow -d_R/2$. Iterate over this process a number of times.

Advice

The PS method works well for some optimization problems, especially problems which must be optimized within a small number of iterations. If you cannot get good optimization results with the PS method, you may wish to try using the LUS method or the DE method instead.

3.7 Local Unimodal Sampling (LUS)

The LUS 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 (5) (9).

Sampling Range Decrease

The current position is denoted $\vec{x} \in \mathbb{R}^n$ which is initially picked at random from the entire search-space. Note that the search-space is n-dimensional. The potential new position is denoted $\vec{y}$ and is sampled from the neighbourhood of $\vec{x}$ as, 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 $\vec{b}_{lo}$. 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 governing the behaviour of the LUS method. A value of $\gamma = 3$ has been found to work well for many optimization problems. Note that a parameter α or β is sometimes used in the literature, which merely equal the reciprocal γ.

Update Rule

The LUS method uses a plain deterministic update rule, meaning the LUS method only moves from position $\vec{x}$ to position $\vec{y}$ in case of improvement to the fitness.

Advice

The LUS method has been found to work well for many optimization problems. Especially problems which are fairly smooth (approaching unimodality, hence the name of the method), and which must be optimized within a small number of iterations. If you cannot get good optimization results with the LUS method, you may wish to try using the DE method.

3.8 Differential Evolution (DE)

The multi-agent optimization method known as Differential Evolution (DE) is originally due to Storn and Price (10).

Basic Variants

Several DE variants are in existence (11) (12), but the one implemented here is a simple variant that has been found to perform on par with more complex variants. It is referred to as DE/best/1/bin/simple (aka. The Joker) and was introduced in (3). Let $\vec{x}$ denote the position of the agent that is currently being updated, and which has been picked at random from the entire population. Let $\vec{y} = [y_1, \ldots, y_n]$ be its new potential position that is to be computed. The original presentation of DE computes $\vec{y}$ in several steps but these have been combined to form a single formula here. For the DE/best/1/bin/simple variant this formula is:

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

where $\vec{g}$ is the population's best known position so far, and the vectors $\vec{a}$ and $\vec{b}$ are the positions of randomly picked agents, which are chosen to be distinct from each other. The index $R \in \{1, \ldots, n\}$ is randomly picked, and a stochastic decision is made using $r_i \sim U(0,1)$ for each dimension i on whether to use x_i or a computed crossover of the other agents to determine the value of y_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.

Advice

Sometimes DE is overkill and you may only need to use PS or LUS. At other times you may need to tune the behavioural parameters of DE using meta-optimization so as to achieve the best performance; see the tutorial source-code on how to do this.

3.9 Particle Swarm Optimization (PSO)

The optimization method known as Particle Swarm Optimization (PSO) is originally due to Kennedy, Eberhart, and Shi (14) (15). It works by maintaining a swarm of agents (usually called particles), each having an associated velocity that is updated recurrently and added to the agent's current position to move it to a new position.

Velocity Update

Let $\vec{x}$ denote the current position of an agent. Then the agent'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 agent'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 dynamic range of the search-space, so an agent can not move farther than from one search-space boundary to the other in a single move, and the velocity is also not allowed to grow indefinitely.

Position Update

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

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

The agent's position is hence updated regardless of improvement to its fitness.

Advice

Empirical results suggest that the DE method may be a better choice than the PSO method (6) (4). If you must use the PSO method for one reason or another, you may also try using the MOL method instead, as it appears to offer a slight advantage over the PSO method on some optimization problems (6). Furthermore, the parameters of the MOL method appear to be slightly easier to tune using meta-optimization than the parameters of the PSO method.

Bibliography

1. **Pedersen, M.E.H.** *RandomOps - (Pseudo) Random Number Generators for C#, URL http://www.Hvass-Labs.org/.* s.l. : Hvass Laboratories, 2009.

2. **Free Software Foundation.** *GNU Lesser General Public License. URL http://www.gnu.org/copyleft/lesser.html.*

3. **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.

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

5. **Pedersen, M.E.H.** *Simplifying Swarm Optimization (PhD Thesis).* s.l. : School of Engineering Sciences, University of Southampton, United Kingdom, In preparation.

6. *Simplifying Particle Swarm Optimization.* **Pedersen, M.E.H. and Chipperfield, A.J.** s.l. : Applied Soft Computing, In print.

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

8. *"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.

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

10. *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.

11. *On the usage of differential evolution for function optimization.* **Storn, R.** Berkeley, CA, USA : Biennial Conference of the North American Fuzzy Information Processing Society (NAFIPS), 1996. pp. 519-523.

12. **Price, K., Storn, R. and Lampinen, J.** *Differential Evolution - A Practical Approach To Global Optimization.* s.l. : Springer, 2005.

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

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